SRISHTI '18

# GRAPH THEORY

Programming And Algorithms Group

PAG

## Topics that we will cover in this lecture

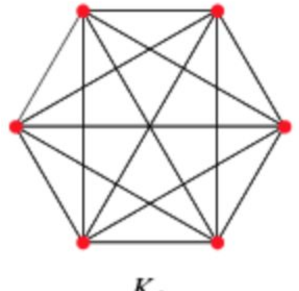
- Introduction to Graphs and Basic Terminology
- Representation of Graphs
- Standard Template Libraries in C++/JAVA
- Graph Searching Algorithms
- Dijkstra’s Algorithm

## Basic Terminologies in Graph Theory

- **Node:** A vertex (plural vertices) or **node** is the fundamental unit of which graphs are formed.
- **Edge**: An edge can be referred to as the 2-element subsets of the set of Vertices. It means that the two elements in the subset are connected.
- **Graph:** A graph **G=(V,E)** consists of a finite set V (set of vertices or nodes) and a list E of pairs of elements from V.

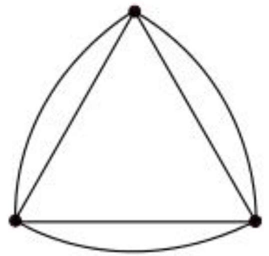
# Basic Terminologies in Graph Theory

- **Complete Graphs:** A complete graph is a graph in which each pair of graph vertices is connected by an edge. A complete graph of n vertices is denoted by $K_n$.



$K_6$

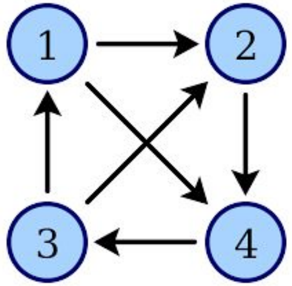
- **Multi Graphs:** A graph having multiple edges between same pair of nodes.



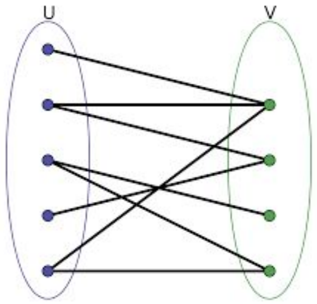
- **Pseudo Graphs:** A graph in which loops are also allowed.

# Basic Terminologies in Graph Theory

- **<u>Directed Graphs</u>:** Graphs with directional edges.



- **<u>Bipartite Graph</u>:** A graph in which the set of vertices V can be decomposed in 2(k for k-partite) disjoint sets having no edge among the elements of a set.



# Some Terminologies

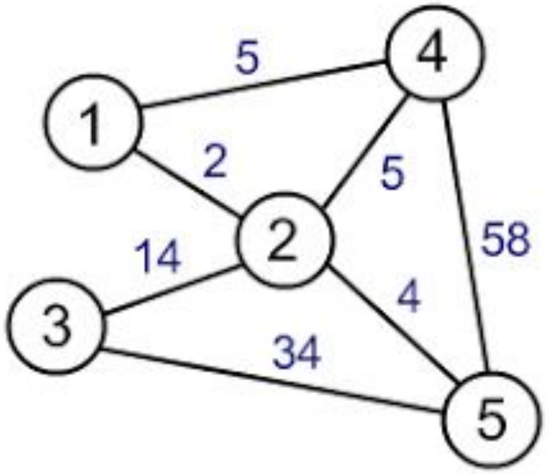
- ❖ **Walk**: A ***walk*** is a list of vertices and edges ( $v_0$ $e_1$ $v_1$ $e_2$ . . . . $e_k$ $v_k$ ), such that for 1<=i<=k, the edge $e_i$ connects $v_{i-1}$ and $v_i$.
- ❖ **Trail:** A ***Trail*** is a **walk** with no repeated edge.
- ❖ **Path**: A Simple ***Path*** is a **trail** with no repeated vertices (hence no repeated edges).
- ❖ **Cycle**: A ***Cycle*** is a **path** in which no vertex is repeated, except the first and the last vertex, which are same.

## Degree of a Vertex

- The ***degree of a vertex*** v is the number of edges whose one of the end is vertex v i.e., the number of edges that are incident on v.
- The sum of degree of all the vertices is twice the number of vertices.
- **The Handshaking Theorem:** In a finite graph, the number of vertices with odd degree will be even.

## Weighted Graph

If each edge has some weight associated with it, it is called a ***Weighted Graph***.



## Representation of Graphs

To represent a graph, all we need is the set of vertices, and for each vertex the neighbors of the vertex (vertices directly connected to it by an edge) and if its a weighted graph, the weight associated with each edge. Graphs are most commonly represented in either of the following two ways -

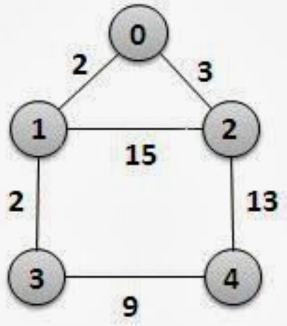
- **Adjacency List Representation**
- **Adjacency Matrix Representation**

# Adjacency Matrix Representation

In this representation, we construct a *nxn* matrix A where n is the number of vertices. If there is an edge from a vertex *i* to vertex *j* , then the corresponding element of A, $a_{i,j}$ = 1, otherwise $a_{i,j}$= 0. Note, even if the graph on 100 vertices contains only 1 edge, we still have to have a 100x100 matrix with lots of zeroes. For a weighted graph, instead of 1s and 0s, we can store the weight of the edge. In that case, to indicate that there is no edge we can put a safely large value (we can call it INF (infinity) ).

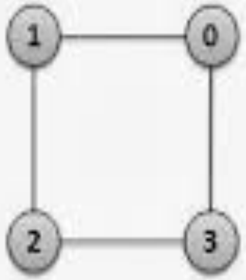
This representation can be quite useful sometimes.For a simple undirected graph,$A^K$ will give the number of walks of length K.

**Practice problem:** http://codeforces.com/problemset/problem/166/E



| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 3 | 0 | 0 |
| 1 | 2 | 0 | 15 | 2 | 0 |
| 2 | 3 | 15 | 0 | 0 | 13 |
| 3 | 0 | 2 | 0 | 0 | 9 |
| 4 | 0 | 0 | 13 | 9 | 0 |

Adjacency Matrix Representation of Weighted Graph



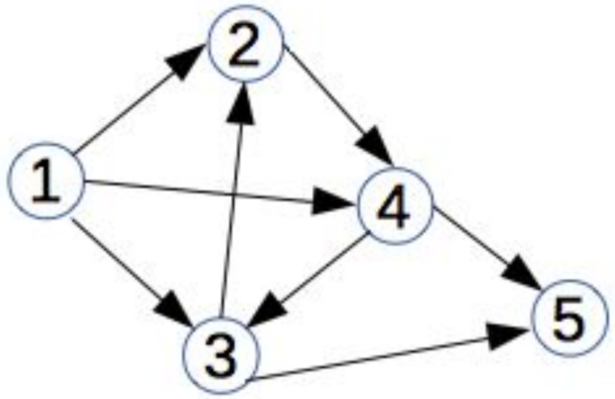
| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 1 | 0 | 1 | 0 |

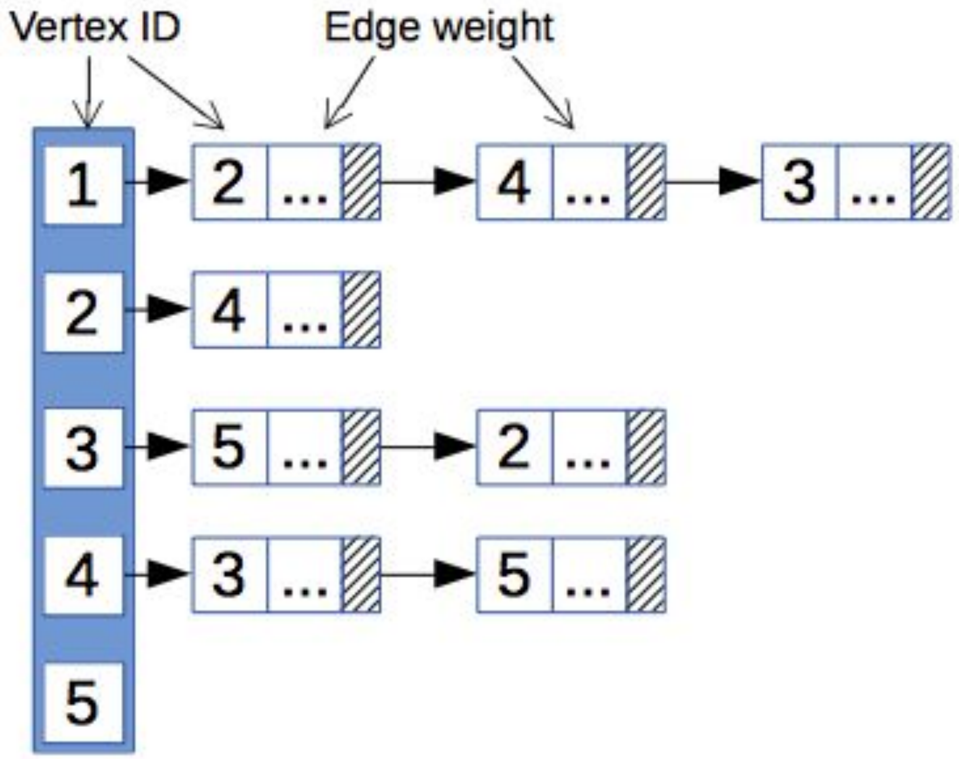
Adjacency Matrix Representation of Undirected Graph

## Adjacency List Representation

In this representation, for each node in the graph we maintain the list of its neighbors. We have an array of vertices, indexed by the vertex number and for each vertex *v*, the corresponding array element points to a singly-linked list of neighbors of *v* .

# Adjacency List Representation of a given Graph





## Adjacency Matrix vs Adjacency List

While the adjacency list saves a lot of space for sparse graphs, we cannot instantly access an edge. While we can access any edge instantly in adjacency matrix, it takes a lot of space (space is fine for graphs even with n = 1000 ) and to visit all the neighbors of a vertex, we have to traverse all the vertices in the graph ( the whole row corresponding to the vertex ), which takes quite some time.

# Standard Template Library (STL) - C++

- ❑ **Vector -** *vector<int> v;*
- ❑ **Stack -** *stack<int> s;*
- ❑ **Queue -** *queue<int> q;*
- ❑ **Pair -** *pair<int,int> p;*
- ❑ **Set -** *set<int> s;*
- ❑ **Multiset -** *multiset<int> m;*
- ❑ **Map -** *map<int,int> d;*
- ❑ **Priority Queue -** *priority_queue<int> p;*

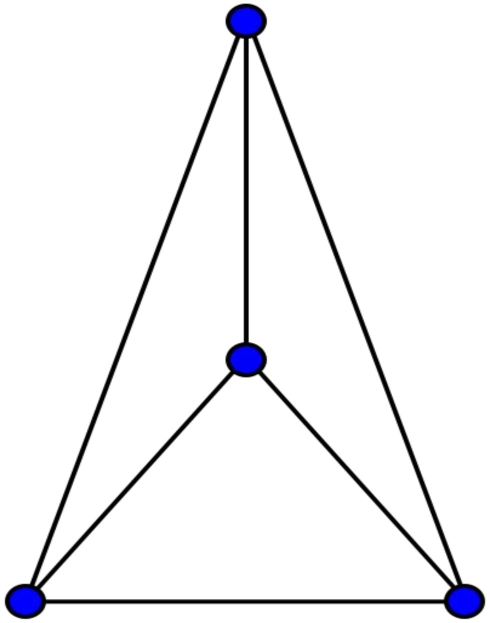
Link for syntax - *https://www.geeksforgeeks.org/the-c-standard-template-library-stl/*

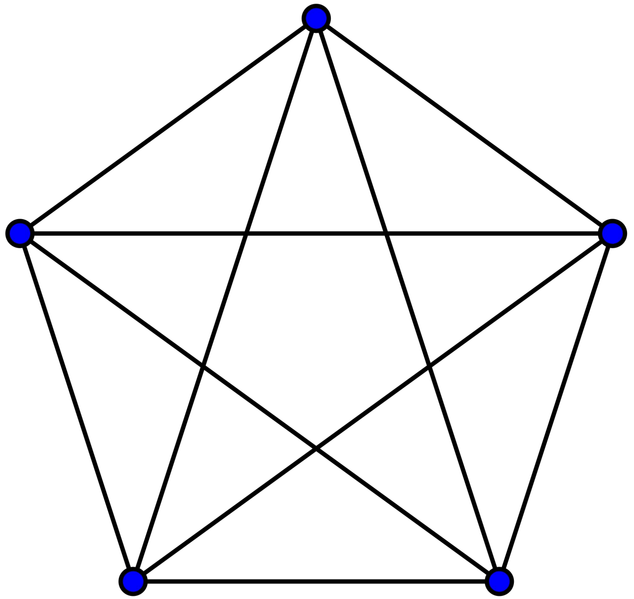
## Planar Graphs

In **graph** theory, a **planar graph** is a **graph** that can be embedded in the **plane**, i.e., it can be drawn on the **plane** in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other.

For a planar graph $\mathbf{e \leq 3}\boldsymbol{v}\mathbf{ - 6}$ always holds true, where **e** denotes the number of edges and **v** the number of vertices.

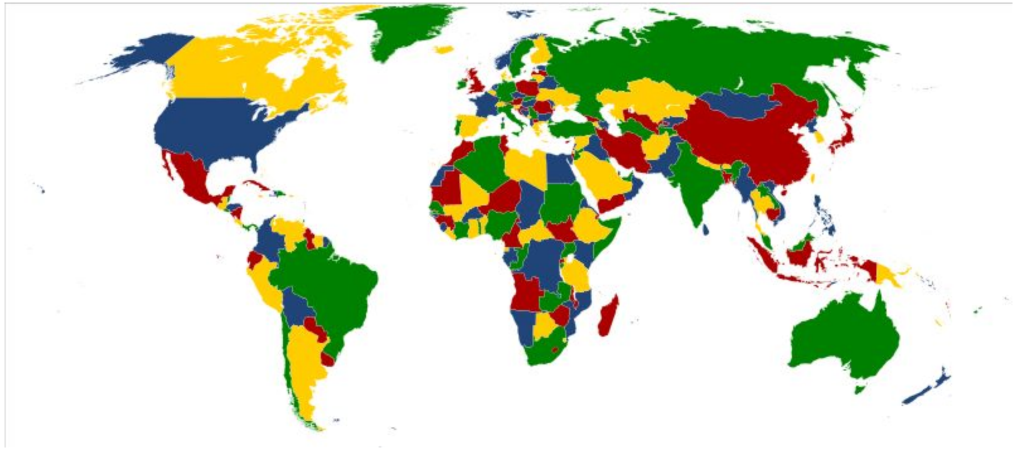
# Example of Planar | Nonplanar Graph


PLANAR ($K_4$)


NONPLANAR ($K_5$)

# Four Color Theorem

The **four color theorem** states that any map--a division of the plane into any number of regions--can be colored using no more than four colors in such a way that no two adjacent regions share the same color.



## Graph Search Algorithms

Most of the time it is required to search for nodes in the graph or traverse the graph and compute something during the traversal.

2 of the commonly used graph traversal methods are:

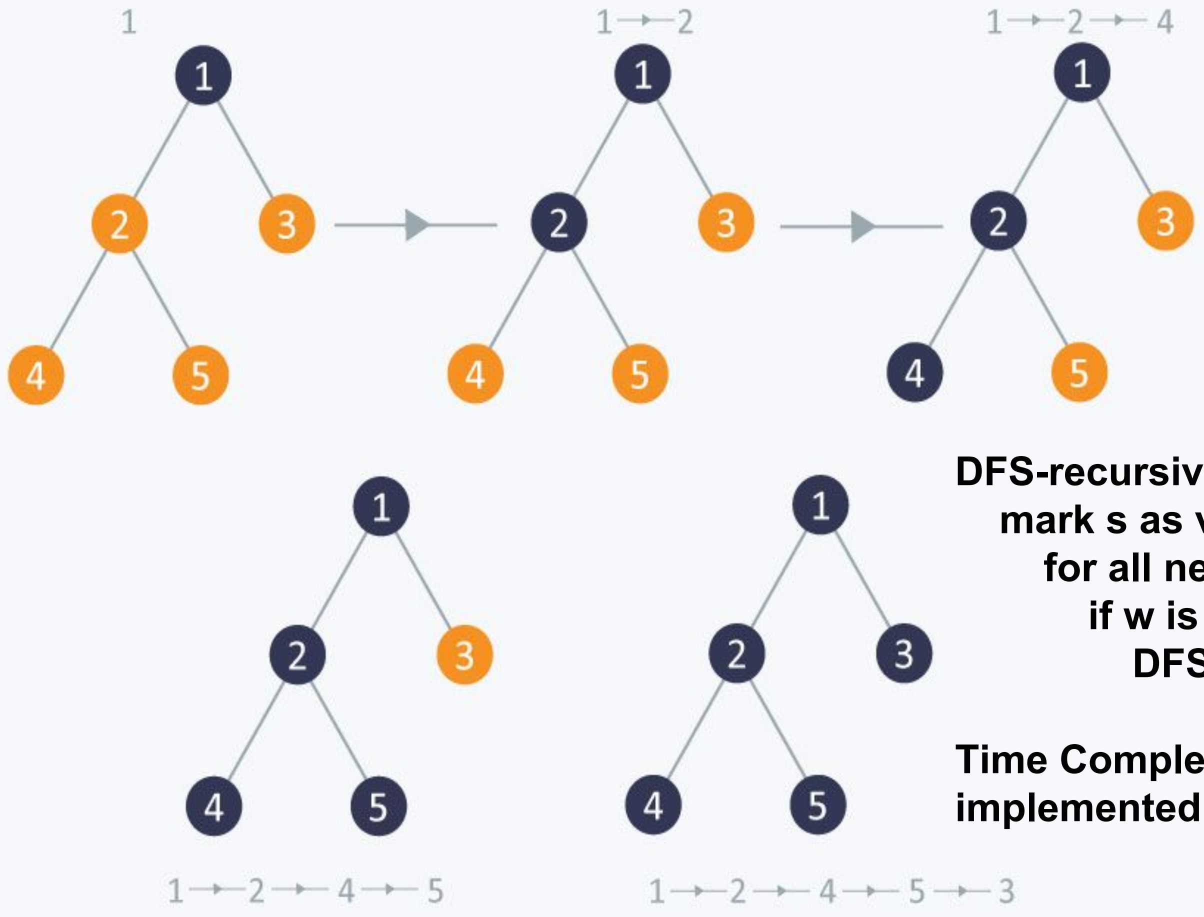
- **Depth First Search (DFS)**
- **Breadth First Search (BFS)**

These are one of the basic and highly useful algorithms in graph theory.

# Depth First Search (DFS)

Depth-first search algorithm searches deeper in graph whenever possible. In this, edges are explored out of the most recently visited vertex that still has unexplored edges leaving it. When all the adjacent vertices of that vertex have been explored, the search goes backtracks to explore edges leaving the vertex from which a vertex was recently discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. It works on both directed and undirected graphs.

# DFS



```
DFS-recursive(G, s):
    mark s as visited
        for all neighbours w of s in Graph G:
            if w is not visited:
                DFS-recursive(G, w)
```

Time Complexity : O(E+V) when implemented using adjacency list

# Pseudo Code for DFS

```
DFS-iterative (G, s):  // Where G is graph and s is source vertex
    let S be stack
    S.push( s )           // Inserting s in stack
    mark s as visited
    while ( S is not empty):
    // Pop a vertex from stack to visit next
    v  =  S.top( )
    S.pop( )
    // Push all the neighbours of v in stack that are not visited
    for all neighbours w of v in Graph G:
        if w is not visited :
            S.push( w )
            mark w as visited
```

## Problem 1: Unreachable Nodes

You have been given a graph consisting of *N* nodes and *M* edges. The nodes in this graph are enumerated from *1* to *N* . The graph can consist of self-loops as well as multiple edges. This graph consists of a special node called the head node. You need to consider this as the entry point of this graph. You need to find and print the number of nodes that are unreachable from this head node.

Constraints : 1<=N,M<=10^5

# Solution: Unreachable Nodes

```
Unreachable-nodes(Graph G, Node special node)
    Initialise all nodes as unvisited
    dfs(G,special node)
    Return number of unvisited nodes
```

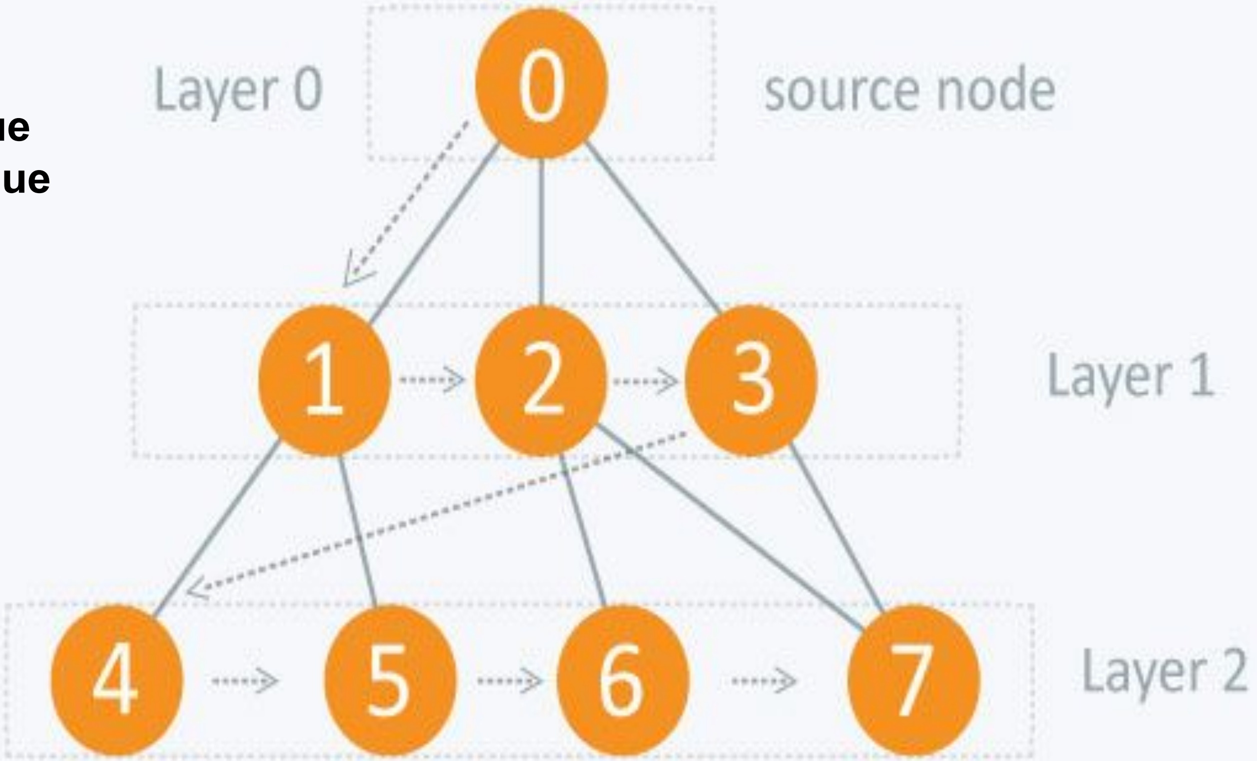
## Breadth First Search (BFS)

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

```
bfs-recursive(G,s)
    for all neighbours w of s in Graph G:
        if w is not visited:
            Visit w
            Insert w in queue
    For all vertices v in queue
        bfs-recursive(G,v)
```



# Pseudo Code for BFS

```
BFS (G, s)                    //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.
    mark s as visited.
    while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be visited now
        v  =  Q.dequeue( )
       //processing all the neighbours of v
       for all neighbours w of v in Graph G
           if w is not visited
                Q.enqueue( w )            //Stores w in Q to further visit its neighbour
                mark w as visited.
```

# Problem 2: Level Nodes

You have been given a Tree consisting of *N* nodes. A tree is a fully-connected graph consisting of *N* nodes and N-1 edges. The nodes in this tree are indexed from *1* to *N*. Consider node indexed *1* to be the root node of this tree. The root node lies at level one in the tree. You shall be given the tree and a single integer *x* . You need to find out the number of nodes lying on level *x*.

The first line consists of a single integer *N* denoting the number of nodes in the tree. Each of the next

Next N-1 lines consists of 2 integers *a* and *b* denoting an undirected edge between node *a* and node *b*. The next line consists of a single integer *x*.

Constraints: 1<=N<=10^5

# Solution: Level Nodes

- Use breadth first search to traverse the tree.
- Start with level=1
- Increment the level after completion of each level.
- Return the number of nodes on level x.

```
levelnodes(Tree T,level x)
  Queue q
  q.enqueue(1)
  level=2
  answer=0
  q.enqueue(N+1) // representing level is finished
  while(q is not empty)
    v=q.dequeue
    if(v>N)
        level++
    else
         for all neighbours w of v in Graph G
           if w is not visited
               Visit w
               Q.enqueue( w )
                   if(level==x)
                       answer++
 return answer
```

## Problem 3: Prime Path (PPATH)

You are given two 4 digit prime numbers a and b. In one step you can change one of the digit in a to create a new number say a1. You can only move from a to a1 if a1 is also a prime number. You have to tell the minimum number of steps that are required to move from a to b. If it is impossible to do so print -1. Number of test cases at most 100.

**Input:**

1

1033 8179

**Output:**

6

**Explanation:**

**1033 -> 1733 -> 3733 -> 3739 -> 3779 -> 8779 -> 8179**

## Solution: Prime Path (PPATH)

The question asks us to find the shortest route between the two prime numbers. What we can do is change each digit of the number and check whether the new number is prime. If its prime then we again apply the above process and continue this process until we reach the destination. The forming of new number and traversing to new one can be easily done using Breadth First Search.

## Dijkstra’s Algorithm - Introduction

Applied on a weighted graph, Dijkstra’s Algorithm is used to find the minimum distance/ sum of weight of edges between any two given vertices. It has many variants  but the most common one is to find the minimum distance from the source vertex to all other vertices in the graph.

## Dijkstra’s Algorithm - Algorithm

- 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.

# References

You can refer to these links to learn more about graphs and to practice problems.

➢ https://www.hackerearth.com/practice/algorithms/graphs
➢ https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/
➢ https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/
➢ https://www.hackerrank.com/domains/core-cs/graph-theory
➢ https://a2oj.com/category?ID=13
➢ https://a2oj.com/category?ID=101
➢ http://codeforces.com/problemset/tags/graphs
➢ http://www.spoj.com/problems/tag/graph-theory

**Feel free to post any queries you have on PAG IITR.**