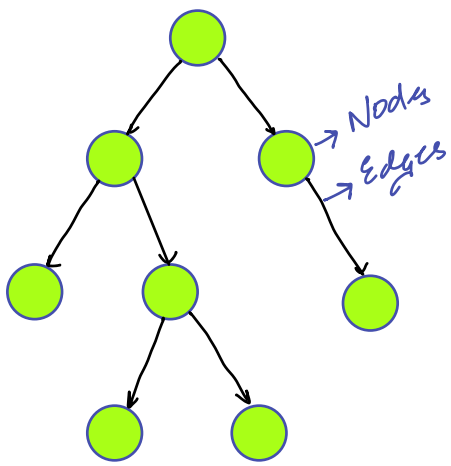
Introduction to graphs

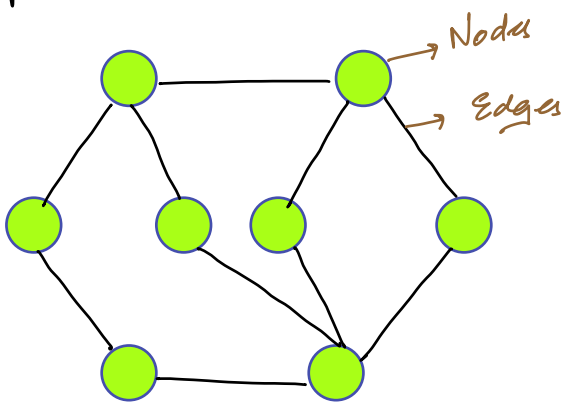
Graph : It is simply nothing but collection of nodes, connected to each other using edges.

Eg1: Tree



N = 8 E = 7

Graph



N = 8 E = 10

Main differences between tree & Graphs

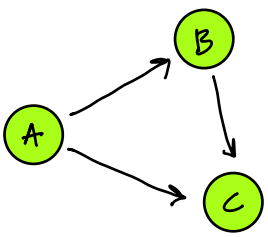
→ Tree is Hierarchial Data Structure

→ In a Tree with Nodes Edges : {N-1}

Classification of Graphs:

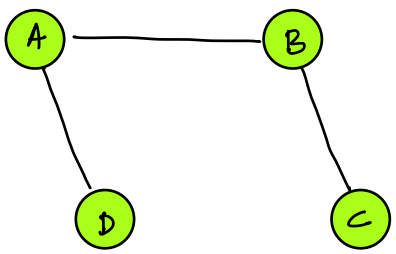
Case-I:

directed: (All Edges)



vs

Undirected graph: (All Edges)



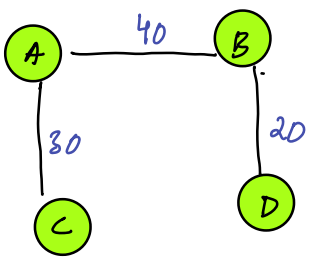
Eg:

Facebook:

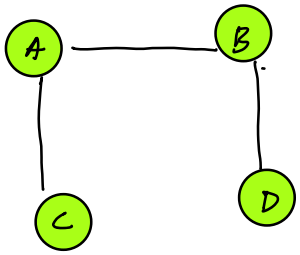
Instagram

Case-II

Weighted



unweighted

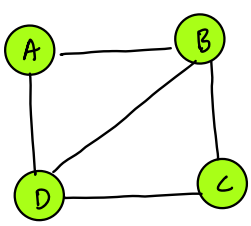


Eg: Google maps

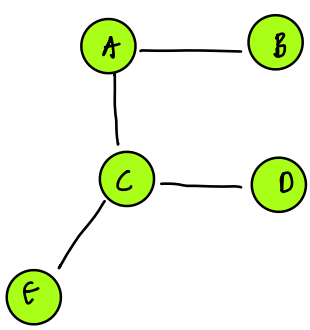
Weight: { dist, time, Money }

Case-III

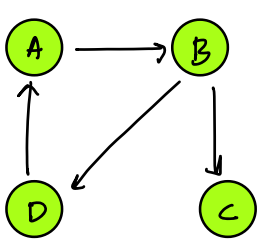
undirected Cyclic graph



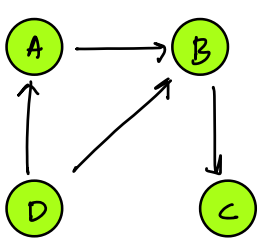
Undirected Acyclic graph



directed Cyclic graph



directed Acyclic graph



How Graph is Given as Input?

→ Any graph is collection of Nodes & Edges

Q1) Given a undirected graph with N Nodes & M Edges

N Nodes   M Edges

// Input ↑ ↑

1st line: N & M

Followed by M lines

Each line contains

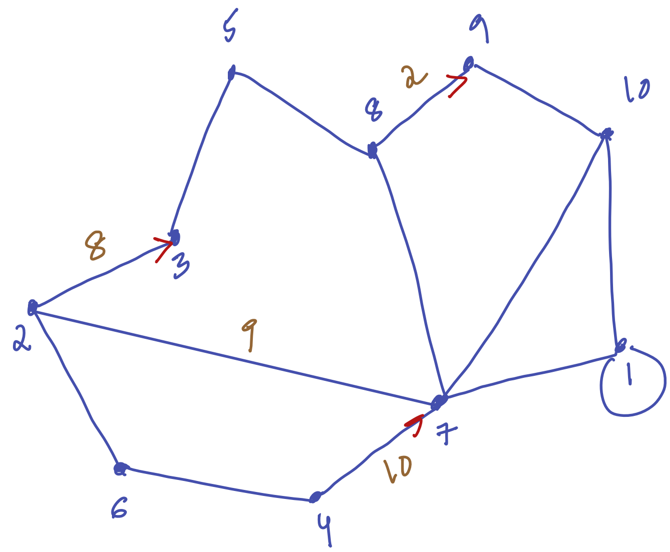
u v

Indicates Edge Between u & v
(nodes, nodes)

N	E	14 Edges
10	14	
u	v	w
2	3	8
4	7	10
8	9	2
2	7	9
7	8	.
10	1	.
4	6	
5	8	
2	6	
10	9	
7	10	
3	5	
7	1	
1	1	

If directed There is Edge from u→v

If weighted graph u, v, w

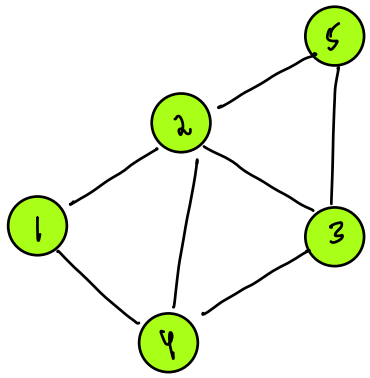


Any graph:

→ undirected vs directed

→ weighted vs unweighted

Storing a graph



Input:

N	t
5	6
1	4
2	5
3	2
4	3
2	4
3	5

App:1 → { Adj Matrix }

int mat[6][6] = { 1 Based indexing }

	0	1	2	3	4	5
0						
1					✓	
2				✓	✓	✓
3			✓		✓	✓
4		✓	✓	✓		
5			✓	✓		

SC: $O(N^2)$
To store adj matrix

// Space Wastage

// int mat[N+1][N+1] = {-1}

mat[u][v] = w

w not only tells weight, it only indicates presence of Edge Between u & v

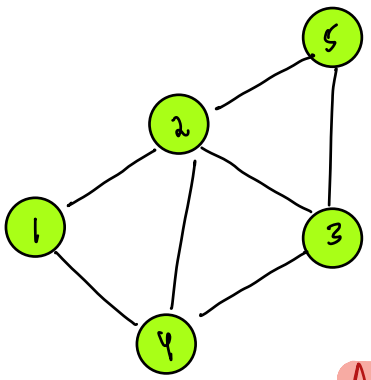
// given N & M mat[N+1][N+1]

// u, v

	unweighted	weighted
undirected	mat[u][v] = 1 mat[v][u] = 1	mat[u][v] = w mat[v][u] = w
directed	mat[u][v] = 1	mat[u][v] = w

App:2 Undirected ⟶

{Mostly for all graph algorithms list is used}



↳ Adj List

App:2 ↳ Dynamic array

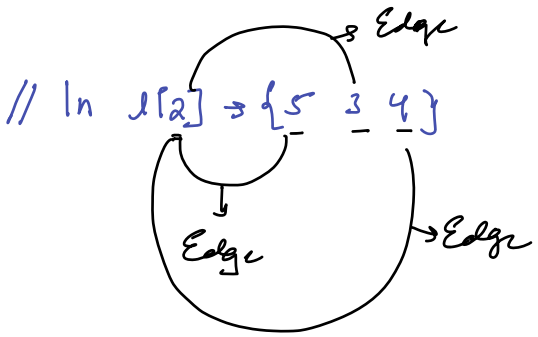
list<int> l[N+1] → SC: 2E → O(E)

Input:

N	t
5	6
1	4 ✓
2	5 ✓
3	2 ✓
4	3 ✓
2	4
3	5

l[]

0
1 → {4}
2 → {5 3 4}
3 → {2 4 5}
4 → {1, 3, 2}
5 → {2 3}



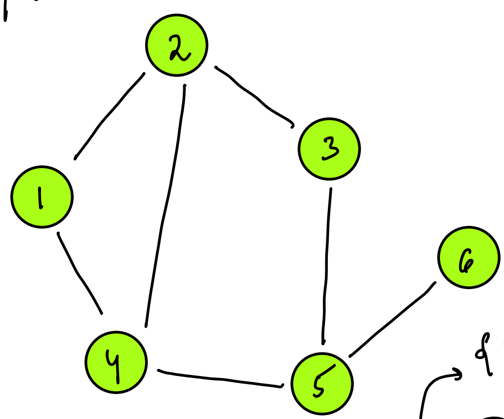
10:20 → 10:30pm

list< pair<int, int> > l[N+1]

	unweighted	weighted {u, v, w}
undirected	l[u].add(v) l[v].add(u) } 2E value	l[u].add(pair{v, w}) l[v].add(pair{u, w})
directed	l[u].add(v) } E value	l[u].add(pair{v, w})

1Q) Given a undirected graph & Source Node & Dest Node, Check if node can be visited from Source Node?

Graph:



S → D
1 → 6

{ ArrayList<ArrayList<>> / Array of Array List → { Syntax for this in your language & code }

Input:

N	E
6	7
1 _ 2	
1 _ 4	
2 _ 4	
2 _ 3	
3 _ 5	
5 _ 6	
4 5	

Adj List → { Array of (Dynamic Arrays) }

list<int> g[7];

0		
1	→ {2, 4}	: 2
2	→ {1, 4, 3}	: 3
3	→ {2, 5}	: 2
4	→ {1, 2, 5}	: 3
5	→ {3, 6, 4}	: 3
6	→ {5}	: 1

Total = 14

undirected → 2E

directed → E

idea: ↦ once a node is added it should not be added in again

2 → bool vis[N+1]

vis[7]:

0	1	2	3	4	5	6
	T	T	T	T	T	T

3) Delete front ele from que, add all unvisited nodes, connected to ele to que

Queue ← 1 | 2 | 4 | 3 | 5 | 6

vis[6] = T { That means we can reach from S → 6 }

Pseudo Code:

// given input N, M, Edges → using input

construct an own list<int> g[N+1]

```
bool BFS ( list<int> g[N+1], int s, int d ){
    bool v[N+1] = {F}
    int dis[N+1] = {N+1}
    Queue<int> q;
    q.insert(s); vis[s] = T, dis[s] = 0
    while( q.size() > 0){
        int u = q.front(u)
        q.delete();
        // for node u we need to node connected to u?
        for( i=0 ; i< g[u].size(); i++){
            int v = g[u][i] , dis[v] = dis[u]+1
            if( vis[v] == False){
                q.insert(v)
                vis[v] = True
            }
        }
    }
    return vis[d] / return dis[d] /
}
```

TC: { We pop every node once → { iterate in it's connected nodes } }

Total iterat: { 2N + 2E }

TC: O(N+E)

SC: O(N + N + E)

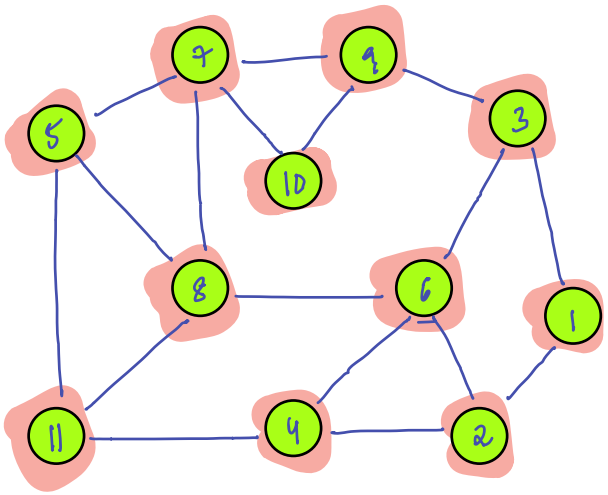
visited nodes

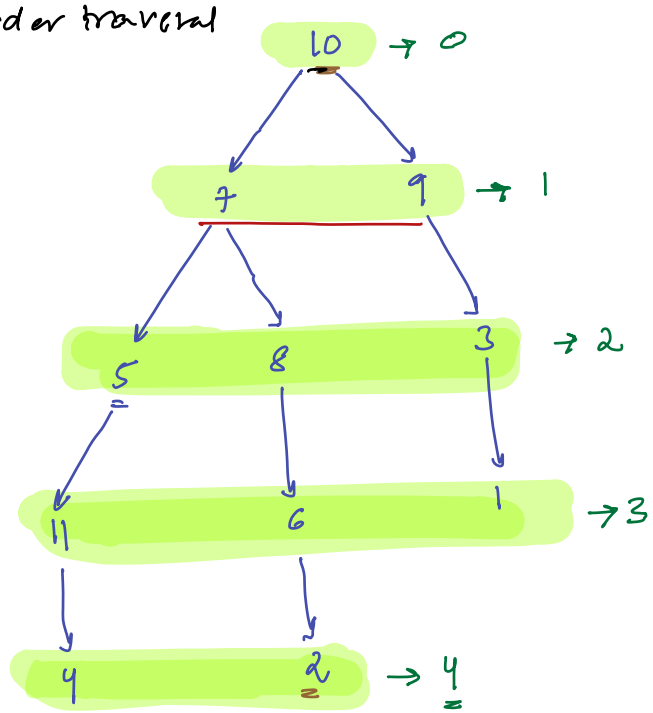
{ Adj list }

Queue Space

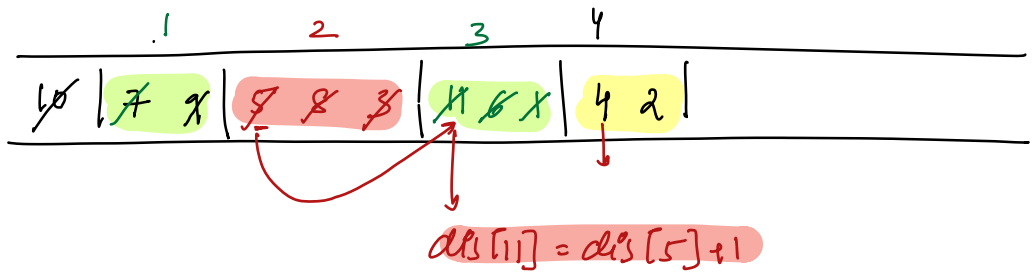
using BFS we can also get min path length from S → D

Ex: S = 10, D = 2 Idea: level order traversal





// length of shortest path from S → D



// Tomorrow → { more Problems on BFS }

// DFS → { Depth first Search }

Doubts:

Nodes:

{ 6 8 14 3 2 9 25 }

Sort nodes

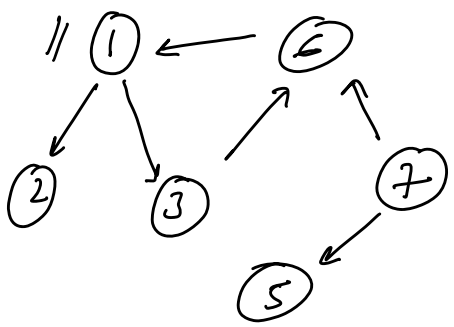
2 3 6 8 9 14 25

Map:

{2:1} {6:3} {9:5} {25:7}

{3:2} {8:4} {14:6}

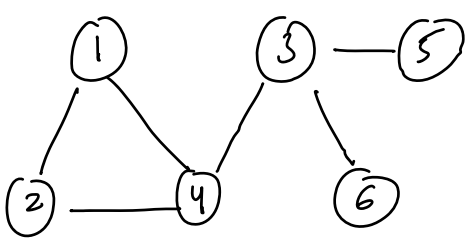
Node number → {1 6}



s=1, d=7

1	2 3	6

Eg:



6 nodes