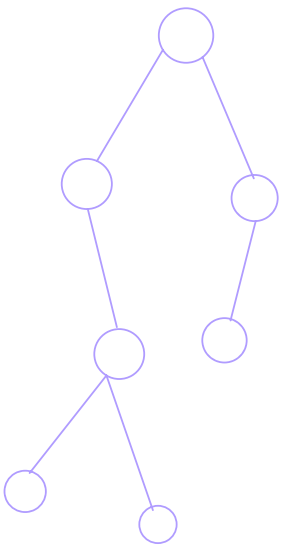
Lecture :- Graph 1

Agenda

- Introduction to graphs
- classification and inputs
- Store a graph
- Bfs and Dfs.

Graph: Collection of nodes connected to each other using edges.

[Tree] → Graph



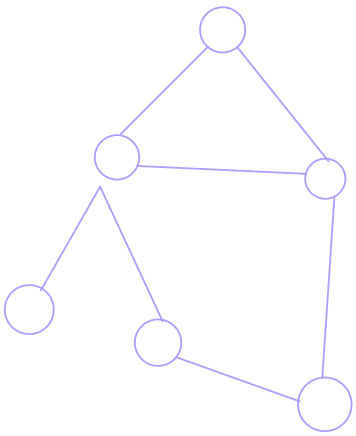
n = 7

edges = 6

↓

No of edges = n - 1

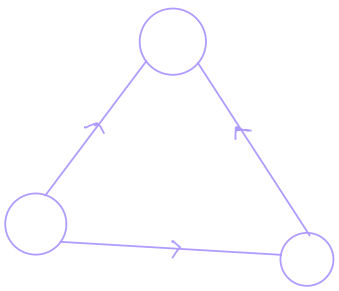
Graph



n = 6

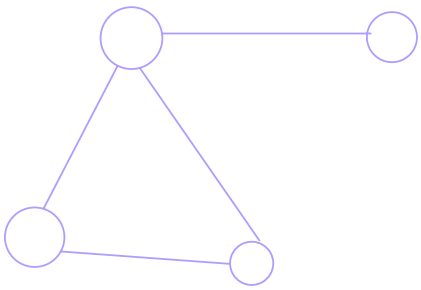
edges = 7

Classification of graphs



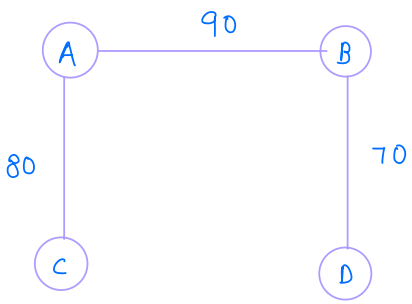
Directed graph

Instagram



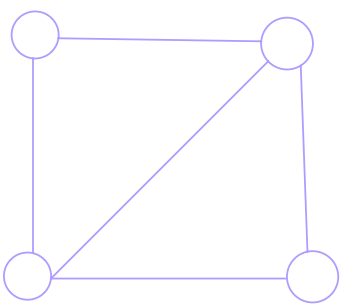
Undirected graph

facebook

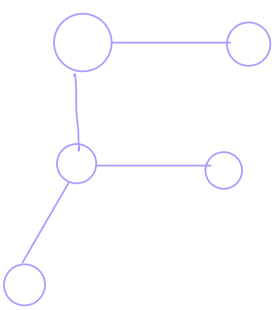


Weighted undirected.

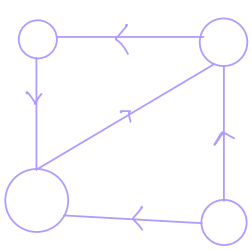
Maps



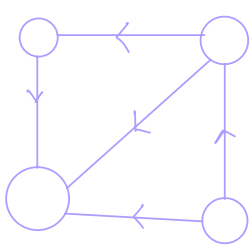
Undirected cyclic graph



Undirected Acyclic graph



Directed cyclic



Directed acyclic

Graph input

Qu. Given an undirected graph with n nodes and m edges.

Input: 1st line : n and m.
↓ no of nodes ↓ no of edges

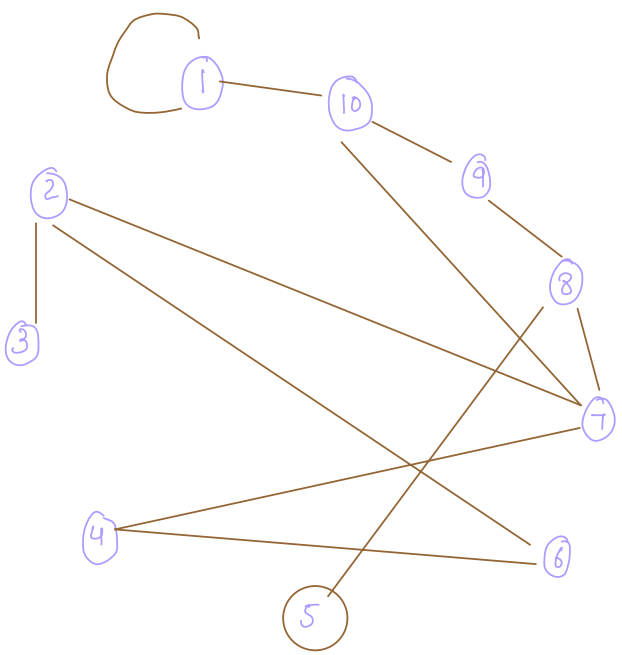
followed by m lines:
Each line contains u and v indicating an edge from u to v.

Eg:

n	m
10	12

u	v
2	3
4	7
8	9
2	7
7	8
10	1
4	6
5	8
2	6
10	9
7	10
1	1

Visualisation

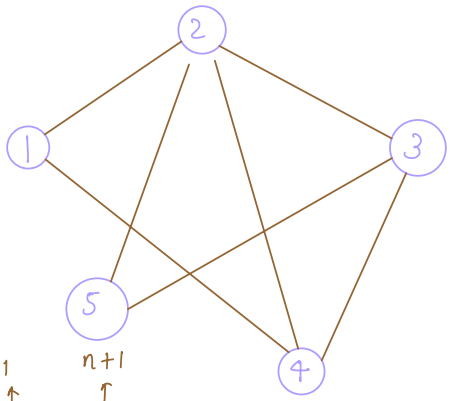


Approach 1 — Adjacency matrix

ip n = 5 , m = 7

Edges:

u	v
1	2
1	4
2	3
2	4
2	5
3	4
3	5



n+1 n+1

mat[6][6]

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	~~0~~ 1	0	~~0~~ 1	0
2	0	~~0~~ 1	0	~~0~~ 1	~~0~~ 1	~~0~~ 1
3	0	0	~~0~~ 1	0	~~0~~ 1	~~0~~ 1
4	0	~~0~~ 1	~~0~~ 1	~~0~~ 1	0	0
5	0	0	~~0~~ 1	~~0~~ 1	0	0

↳ mat[i][j] = 1 { edge b/w i and j }

u	v
1	2
1	4
2	3
2	4
2	5
3	4
3	5

	Unweighted	Weighted
Undirected	mat[u][v] = 1 mat[v][u] = 1	mat[u][v] = wt mat[v][u] = wt
Directed	mat[u][v] = 1	mat[u][v] = wt

Code:

```
int[][] buildGraph(n, m, edges[][]) {
        mat[n+1][n+1];
        for(i = 0; i < m ; i++) {
            u = edges[i][0];
            v = edges[i][1];
            mat[u][v] = 1;
            mat[v][u] = 1;
        }
    return mat;
}
```

TC: O(edges)
SC: O(n*n) ≃ O(n²)
└ Space wasted

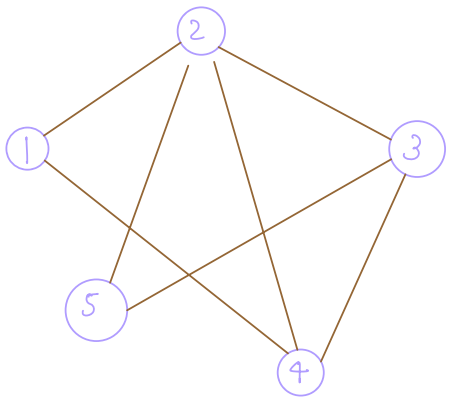
Confusion: mat[u][v] = wt
├ u & v have an edge
└ wt b/w u and v is wt.

Approach2 — Adjacency list

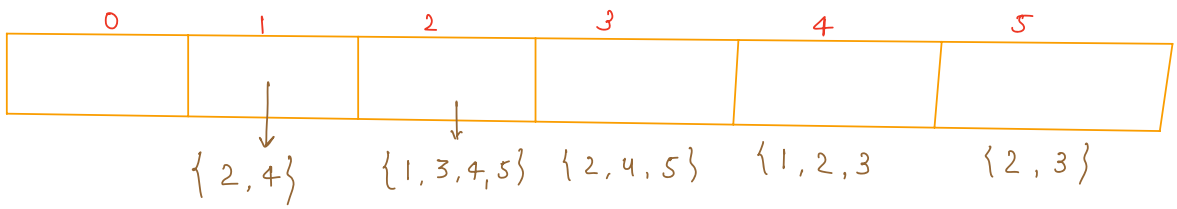
ip n = 5 , m = 7

Edges:

u	v
1	2
1	4
2	3
2	4
2	5
3	4
3	5



List<Integer> l[n+1]



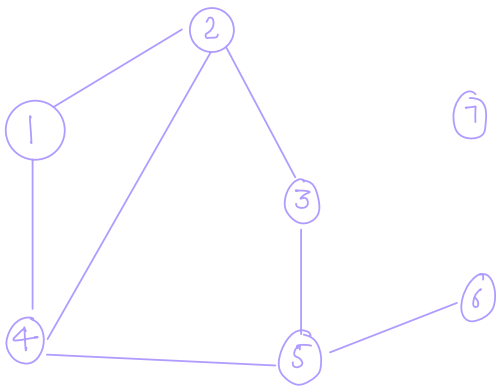
	Unweighted	Weighted
Undirected	l[u].add(v) l[v].add(u)	l[u].add(Pair(v,wt)) l[v].add(Pair(u,wt))
Directed	l[u].add(v)	l[u].add(Pair(v,wt))

code:

```
List<Integer>[]  buildGraph(n, m, edges[][]) {
        List<Integer>[n+1] l;
        for(i=0; i<  m  ;  i++) {
            u = edges[i][0];
            v = edges[i][1];
            l[u].add(v);
            l[v].add(u);
        }
    return l;
}
```

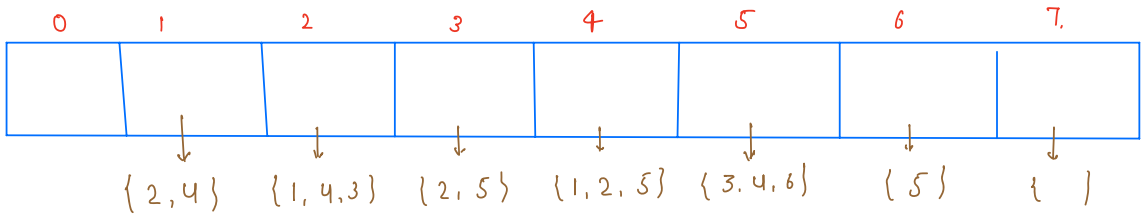
BfS

Qu: Given an undirected graph and source node and destination node. Check if destination node can be visited from source node.



s	d	
1	6	true
1	7	false

Graph:



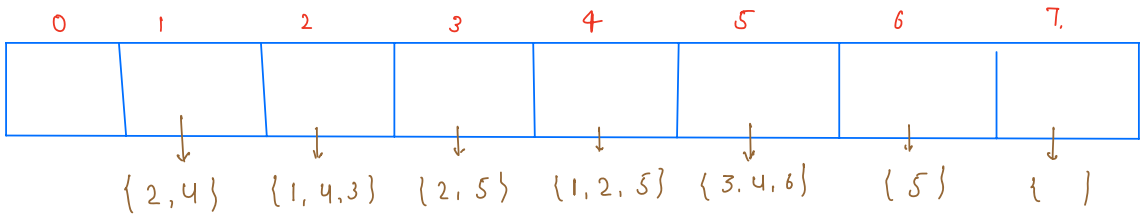
Approach:

source = 1 , destination = 6

Queue:

visited[] =

0	1	2	3	4	5	6	7
f	~~f~~ t	~~f~~ t	~~f~~ t	~~f~~ t	~~f~~ t	~~f~~ t	f



Code:

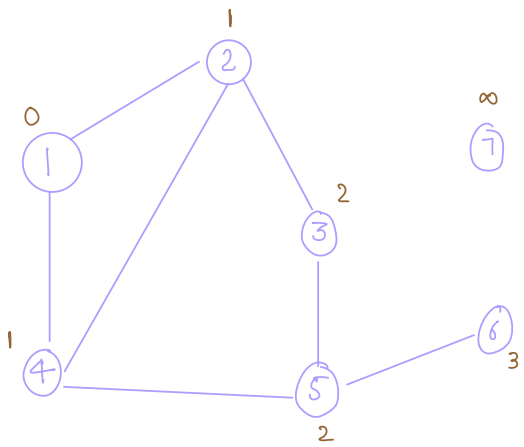
```
boolean bfs(n, m, edges[][], src, dest) {
    List<Integer>[] graph = buildGraph(n, m, edges);
    boolean[] vis = new boolean[n+1];
    Queue<Integer> q;
    q.add(src);
    vis[src] = true;
    while( !q.isEmpty()) {
        curr = q.poll();
        List<Integer> nghbrs = graph[curr];
        for(int el: nghbrs) {
            if(vis[el] == false) {
                q.add(el);
                vis[el] = true;
            }
        }
    }
    return vis[dest];
}
```

TC: $O(n + e)$ ≃ Linear

Sc: $O(n + n + e)$

- n: Queue size
- $n + e$: Adjacency list

Qu: find distance from source node to destination node



s	d	distance
1	6	3
1	7	∞

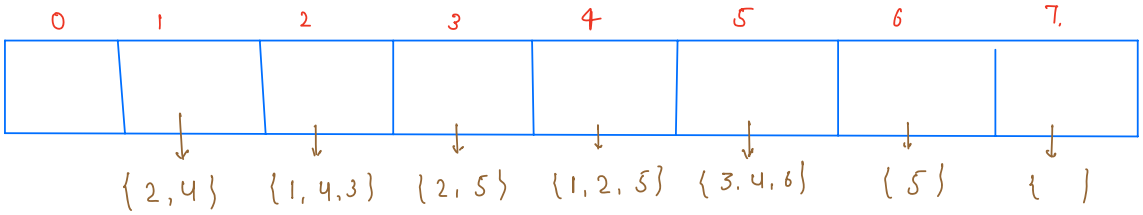
Queue:

visited[] =

0	1	2	3	4	5	6	7
f	~~f~~ t	~~f~~ t	~~f~~ t	~~f~~ t	~~f~~ t	~~f~~ t	f

dist[] =

0	1	2	3	4	5	6	7
∞	~~∞~~ 0	~~∞~~ 1	~~∞~~ 2	~~∞~~ 1	~~∞~~ 2	~~∞~~ 3	∞



Code:

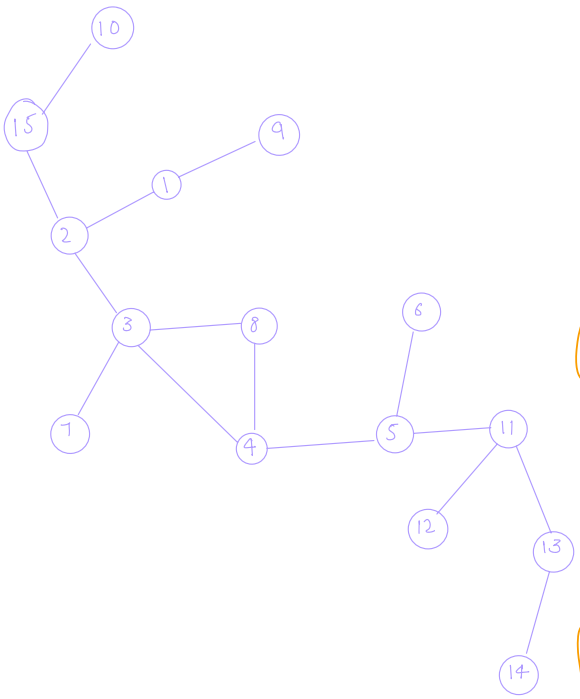
```
int getDistance(n, m, edges[][], src, dest) {

    List<Integer>[] graph = buildGraph(n, m, edges);

    boolean[] vis = new boolean[n+1];
    int[] dis = new int[n+1]; // Initialise as ∞.
    Queue<Integer> q;

    q.add(src);

    vis[src] = true;
    dis[src] = 0;
    while( !q.isEmpty()) {

        curr = q.poll();

        List<Integer> nghbrs = graph[curr];

        for(int el: nghbrs) {
            if(vis[el] == false) {

                q.add(el);

                vis[el] = true;
                dis[el] = dis[curr] + 1;
            }
        }
    }

    return dis[dest];
}
```

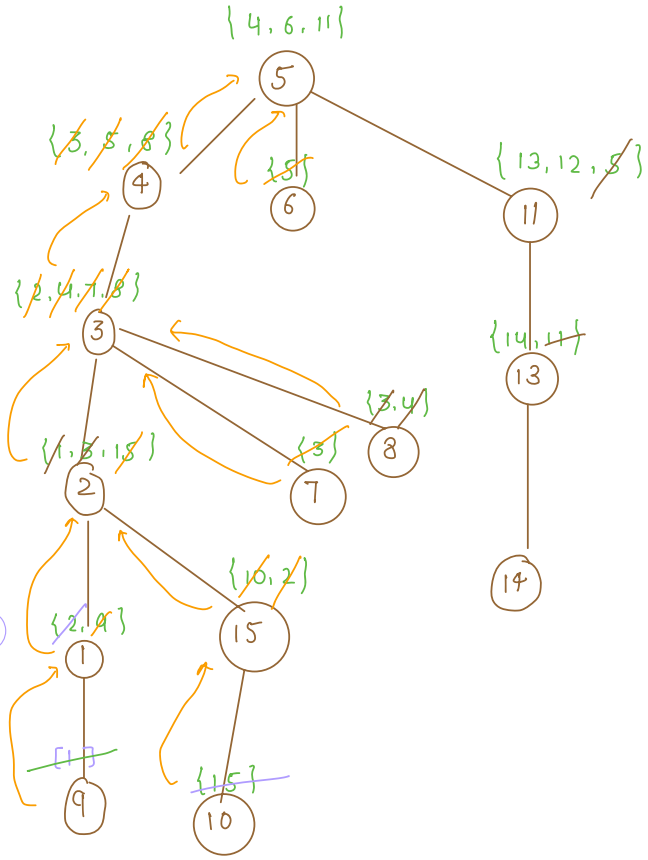
Break: 8:30 - 8:40

Dfs [Depth first search]

src = 5

dest = 14





Code:

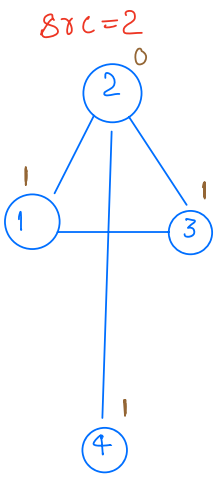
```
boolean isVisited(n, m, edges[][], src, dest) {
        List<Integer>[] graph = buildGraph(n, m, edges);
        boolean[] vis = new boolean[n+1];
        dfs(src, graph, vis);
        return vis[dest];
    }

    void dfs(s, graph, vis) {
        if(vis[s] == true) {
            return;
        }
        vis[s] = true;
        List<Integer> nghbrs = graph[s];
        for(int el: nghbrs) {
            if(vis[el] == false) {
                dfs(el, graph, vis);
            }
        }
    }
}
```

TC: O(n+e)

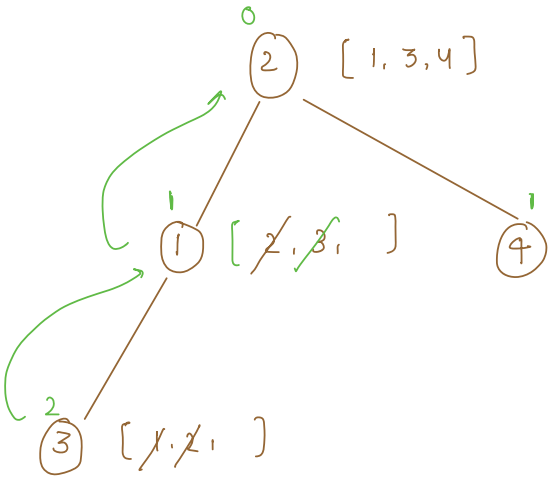
SC: O(n+e) + O(n)

- O(n+e): Adjacency list
- +: stack space
- O(n): vis[]

Shortest path from src to destination



[Bfs]



[Dfs]

Observation

1. Shortest path ⟶ Bfs
2. " " ⟶ cannot be calculated using Dfs

Thankyou ☺