Graph — 1

- Madhan Kumar M S
- Balaji S K
- Bhaveshkumar
- Burhan
- Gagan Kumar S
- Hemant Kumar
- Nikhil Pandey
- Purusharth A
- Rajat Sharma
- Rajendra
- Sanket Giri
- Saurabh Ruikar
- Shani Jaiswal
- Shradha Srivastava
- Sridhar Hissaria
- Sumit Adwani
- Suyash Gupta

Content

- Introduction to Graphs
- Classification of graphs
- Storing a graph
- Traversal on a graph.

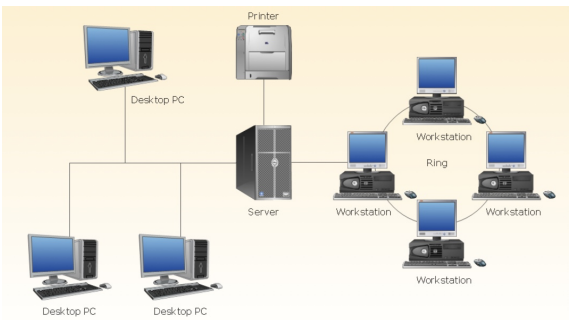
PSP

↓

61% ⟶ 70%

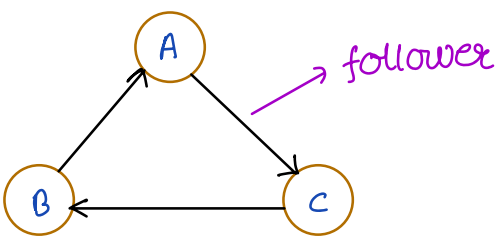
What is graphs ?

It is simply nothing but collection of nodes, connected to each other wing edges.

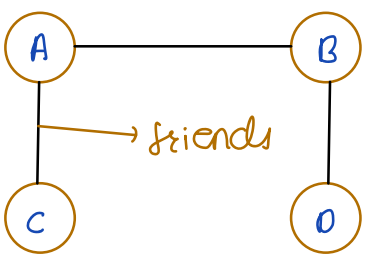


— Classification of Graph

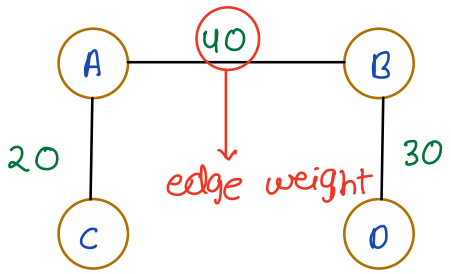
① Directed Graph



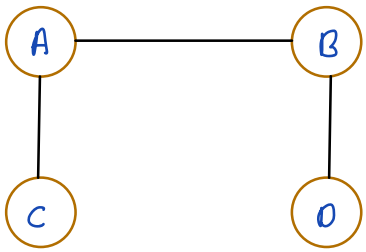
Undirected graph



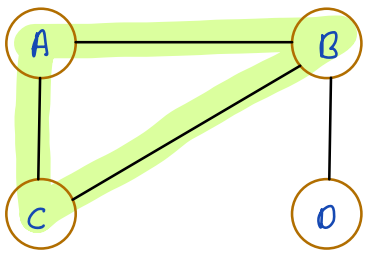
② Weighted Graph



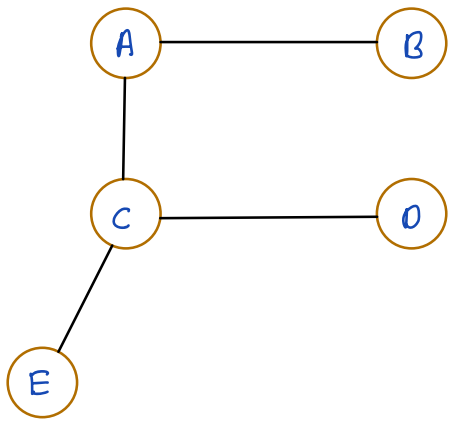
Unweighted



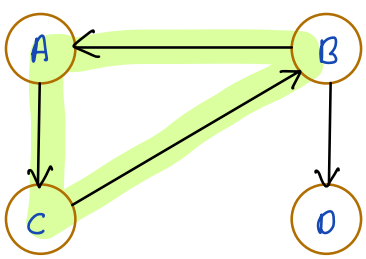
③ Cyclic Graph



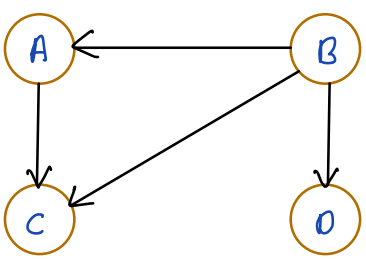
Acyclic



In a cycle no. of node must be atleast 3 {undirected graph}



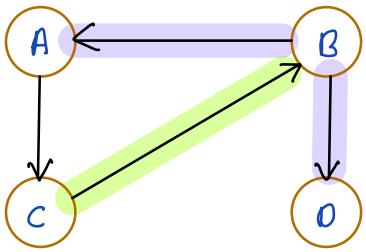
Cyclic Graph



Acyclic

Indegree

No. of incoming edges



Outdegree

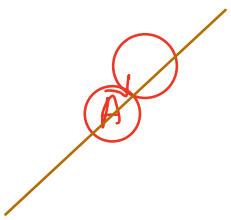
No. of outgoing edges

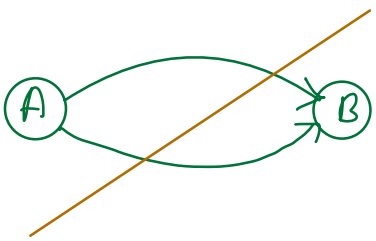
B 1

B 2

Simple Graph

⟶ No self loops or multiple edges b/w same node





Storing a graph { Nodes + Edges }

nodes N | M edges

5 7

start | end

1 4
2 5
3 2
4 3
2 4
3 5
1 2

* Adjacency Matrix

1 ⟶ edge
0 ⟶ no edge

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

edges [7] [2]

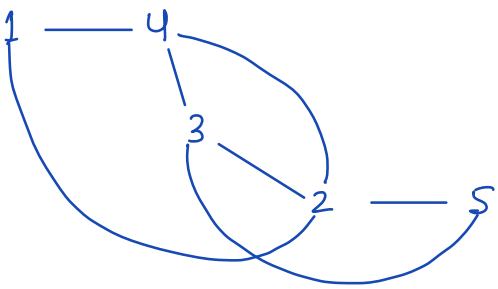
AM [N+1] [N+1] SC : $O(N^2)$

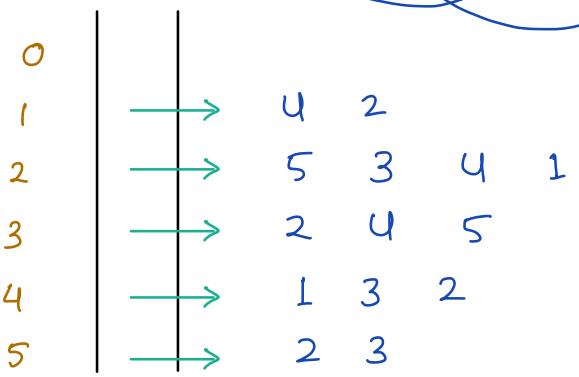
```
for i ⟶ 0 to M-1 {
    start = edges [i][0]
    end   = edges [i] [1]

    AM[start] [end] = 1
    AM[end] [start] = 1
}
```

* Adjacency List

V E
N M
5 7

start end
1 4
2 5
3 2
4 3
2 4
3 5
1 2





How to store

Map < Int , AL of Int >

AL of AL

```
AL = new ArrayList<>()
for i ——→ 0 to N {
|     AL.add ( new ArrayList<>() )
}
```

O(N) space

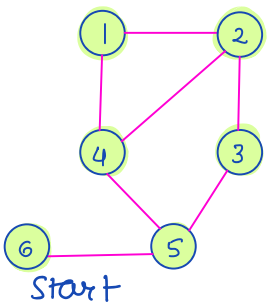
```
for i ——→ 0 to M-1 {
|     start = edges[i][0]
|     end   = edges[i][1]
|     AL.get(start).add(end)
|     AL.get(end).add(start)
}
```

SC : O(N+M)

SC : O(V+E)

Graph Traversal — DFS ⟶ Depth first search

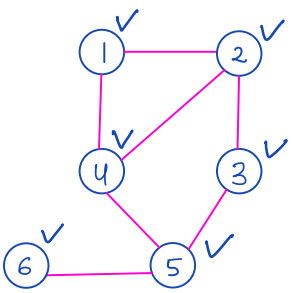
{Inorder}
{Preorder}
{Postorder}



N	M
6	7

start	end
1	2 ✓
1	4 ✓
2	4 ✓
2	3 ✓
3	5 ✓
5	6 ✓
4	5 ✓

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



vis

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

Hash set
Hashmap
Array

Pseudocode

```
// given input N, M
                    -> construct adjacency list.
list < list <int>> graph  // Adjacency list
visited []

                                          V times
for node ---> 1 to N {
    if (!visited [node])  dfs (node)
}

void dfs ( node, ) {
    // mark the node as visited
    visited [node] = true

    // traverse to all unvisited nei of node
    for ( nei : graph.get(node) ) {
        if (! visited [nei]) {
            dfs(nei)
        }
    }
}
```

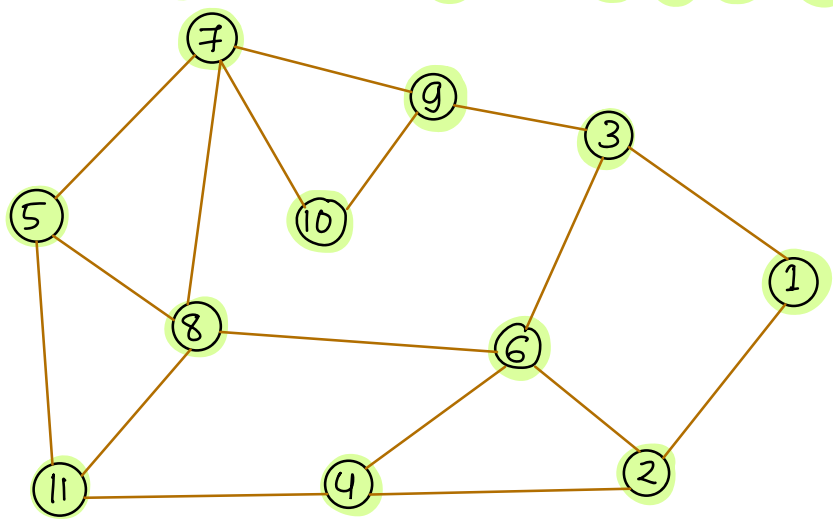
TC: O(V+E)

SC: O(V+E)

visited

0 1 2 3 4 5 6 7 8 9 10 11

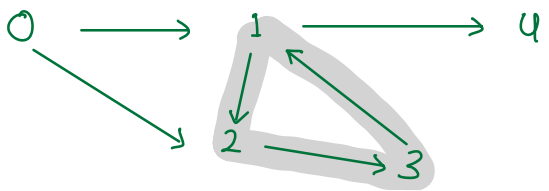
1 → 3 2
2 → 6 1 4
3 → 9 6 1
4 → 11 6 2
5 → 11 8 7
6 → 8 3 4 2
7 → 5 8 9 10
8 → 5 7 11 6
9 → 10 7 3
10 → 7 9
11 → 5 8 4



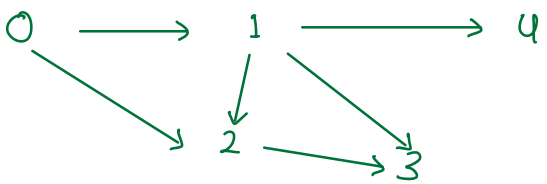
10 7 9 3 6 2 1 4 11 5 8

Break : 22:40

Q1 > Detecting cycle in a directed graph

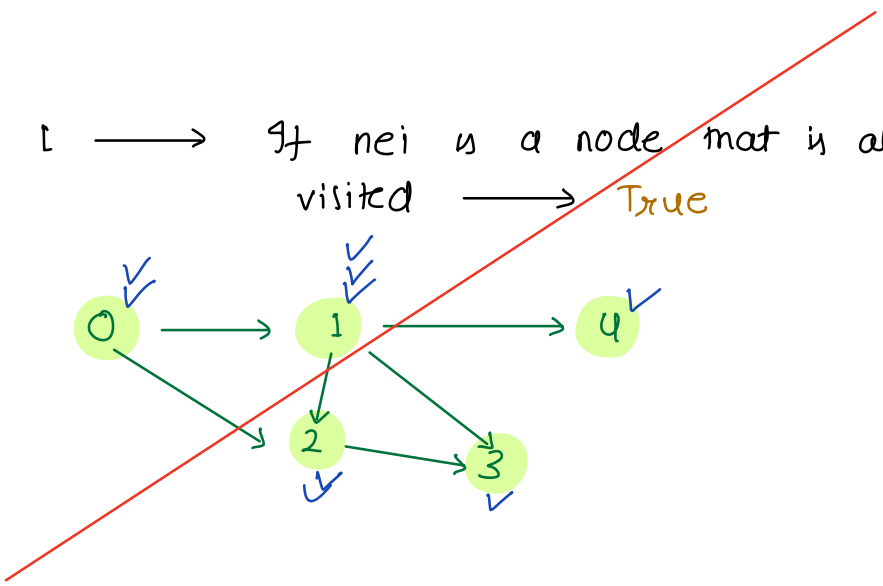


ans = true



ans = false

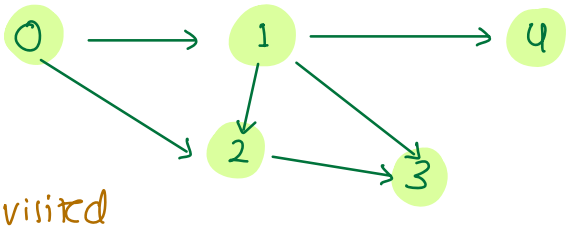
Approach I ⟶ If nei is a node that is already visited ⟶ True



Idea 2 ⟶ Maintain current path

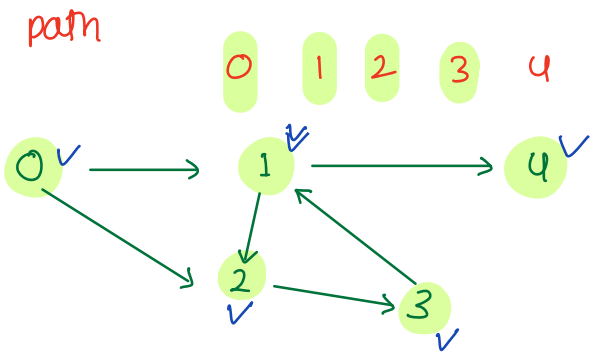
path = [F, F.....]

isCycle = False




```
void dfs ( node ) {
    // mark the node as visited
    visited [node] = true
    path [node] = true          // do
    // traverse to all unvisited nei of node
    for ( nei : graph.get(node) ) {
        if ( path [nei] )  isCycle = true
        if (! visited [nei] ) {
            dfs(nei)
        }
    }
    path [node] = false         // undo
}
```

path F F F F F
0 1 2 3 4



TC : O(V+E)

SC : O(V+E)

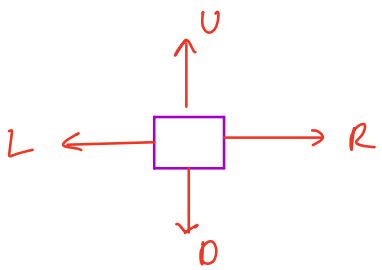
If graph ie AL rep is given as input SC : O(V)

Q 2> No. of Islands

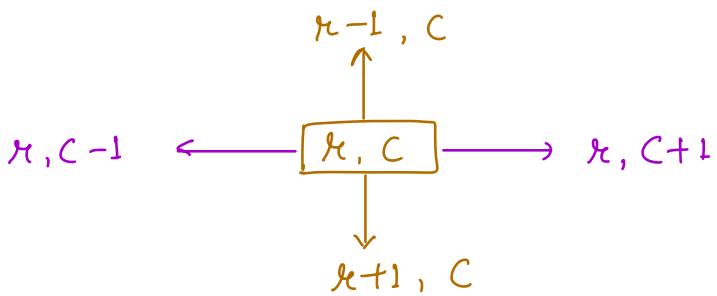
Given 2D grid of 1s (land) 0s (water)

Find the no. of islands in the grid

	0	1	2	3	4
0	1	1	0	0	1
1	0	1	0	1	0
2	1	0	0	1	1
3	1	1	0	0	0
4	1	0	1	1	1



ans = 5



R // no. of rows

C // no. of cols

visited [R][C] // false

islands = 0

LAND = 1

WATER = 0

```
for r ⟶ 0 to R-1
    for c ⟶ 0 to C-1
        val = A[r][c]
        if ( val == LAND &&
```

visited [r][c] == false) {

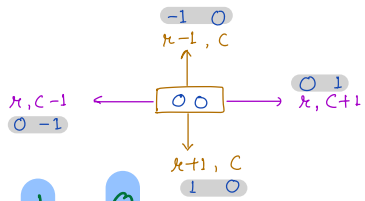
dfs (r , c)

islands ++

}

}

}



	0	1	2	3	4
0	1	1	0	0	1
1	0	1	0	1	0
2	1	0	0	1	1
3	1	1	0	0	0
4	1	0	1	1	1

dirX = −1 0 1 0

dirY = 0 1 0 −1

UP Right Down Left

```
void dfs ( r , c ) {
    visited [r][c] == true

    // go to all the neighbors.
    for i ⟶ 0 to 3 {
        nr = r + dirX [i]
        nc = c + dirY [i]

        if ( 0 <= nr < R && 0 <= nc < C
             !visited [nr] [nc] && A[nr][nc] == LAND )
            dfs ( nr , nc )
    }
}
```

TC : O(R*C)

SC : O(RC)

SC : O(RC)

using the matrix doesn't help

LinkedIn Maximising the Reach

In the LinkedIn ecosystem, understanding the structure of professional networks is crucial for both users and the platform itself.

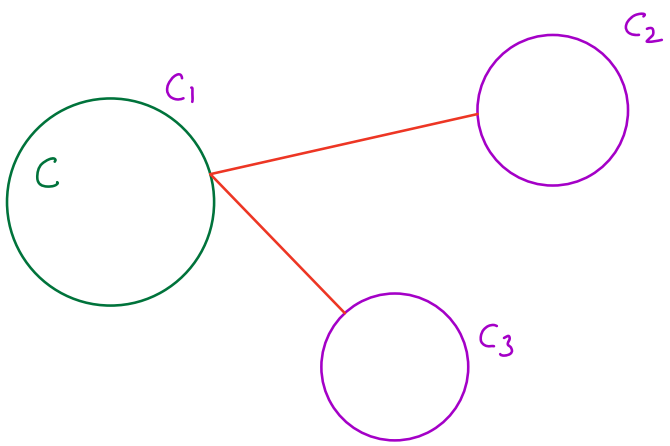
A new feature is being developed to visualize the network of a user in terms of "**clusters**". These clusters represent groups of LinkedIn users who are all connected to each other, either directly or through mutual connections, but do not have connections to users outside their cluster. This visualization aims to help users to increase their **Reach**.

Given **A** denoting the total number of people and matrix B of size Mx2, denoting the bidirectional connections between users, and an Integer C denoting the user ID of each connection joins two users by their user IDs, the target person, find out the number of connections that this person should make in order to connect with all the networks.

A ⟶ total no. of people

B[M][2] ⟶ edges

C ⟶ user id of the person who needs to make connections.



ans = 2.

Idea — Find out no. of clusters

qry = no. of clusters − 1