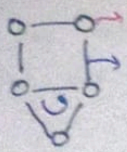
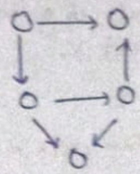
Graph_Day_1

Introduction to graphs: → Not only enclosed figures



{undirected graph: cyclic:



edges are unidirected

Directed graph: acyclic:

Degree of graph:
undirected graph
:- No. of edges attached

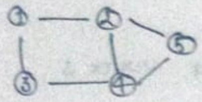
| Total degree = 2 × Edges |

• Directed graph
→ Indegree :- Incoming N
~ Outdegree :- Outgoing N

• Path: Contains a lot of nodes and each of them are reachable
↳ can-not appear twice Node

for ex: 1 2 3 5 → path
1 2 3 2 1 → Not path.

⇒ Graph Representation / Java



Input:
N no. of Nodes ⇒ 5
M, no. → edges ⇒ 6

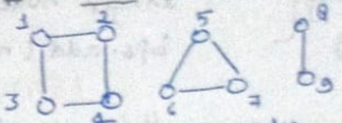
m lines
1 2
1 3
3 4
2 4
2 5
4 5

Store:
→ Matrix {Adjacent} ↳ O(N×N) Method is not used
→ List (Adjacency)
↓
| ArrayList < ArrayList<> > adj |

Space: O(2M) * ← Edges:

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

Connected Components.



N = 10 nodes
8 - edges

There are component of graph

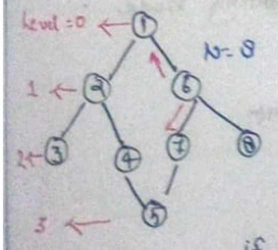
Vis = | 0 | 0 | 0 | 0 - - - | 0 | N = 11
Visited: 0 1 2 - - - 10

for (i = 1 → 10)
{ if (! vis[i]
traversal (i);
}

In one traversal it only visits the connected part rest is untouched.
We need again to call traversal.

⇒ BFS of a graph: → Traversal Technique

Breadth first Search
↳ Level wise:



→ $\underbrace{1}\ \underbrace{2\ 6}\ \underbrace{3\ 4\ 7\ 8}\ \underbrace{5}$

only one node can be at level = 0

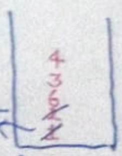
if starting node = 6

We are storing graph in Adjacency list: ↳ data structure.

$\underbrace{6}_{0}\ \underbrace{1\ 7\ 8}_{1}\ \underbrace{2\ 5}_{2}\ \underbrace{3\ 4}_{3}$
Level

Starting Node = 1   Initial

→ 1 2
Who are your Neighbours.



If once visited we need not to do anything.

Space ⇒ O(3N) *

Time ⇒ O(N) + O(2E) → for loop
while:

Vis ⇒ array.

0	1	2	3	4	5	6	7	8
0	1	~~0~~ 1	~~0~~ 1	~~0~~ 1	0	~~0~~ 1	0	0

already in Queue

Code: BFS

```
// function to return Breadth First Traversal of given graph
public ArrayList < Integer> bfsOfGraph ( int V,
ArrayList < ArrayList < Integer>> adj) {

    ArrayList < Integer > bfs = new ArrayList < > ( );
    boolean vis[] = new boolean [V]
    Queue < Integer> q = new LinkedList < > ( );

adding in     { q.add (0);
queue         { vis[0] = true;

              { while ( !q.isEmpty ( )) {
   Traversing {     Integer node = q.poll ( );
and returning {     bfs.add (node);

                    // get all adjacent vertex of the dequeued vertex
                    // if a adjacent has not visited then mark it
                    // and enque it.

getting neighbours  for ( Integer it : adj.get(node)){
of each and adding {     if (vis[it] == false){
in queue.          {         vis[it] = True;
                             q.add (it); }
                    }

              }
            return bfs.
```

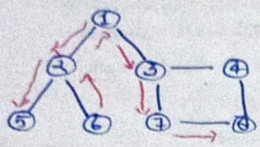
BFS (using deque)

```
From collection import deque
def bfs ( graph, start):
    visited = set()
    queue = deque([start]) # initialize queue with the starting node

    while queue:
        vertex = queue.pop(left) # Remove the first element
        if vertex not in visited :
            print (vertex, end=" ")
            visited.add (vertex)
            # Add all unvisited neighbour to the queue
            For neighbor in graph[vertex] :
                if neighbor not in visited :
                    queue.append (neighbor)

    return bfs
```

DFS Traversal in a Graph. → Recurssion

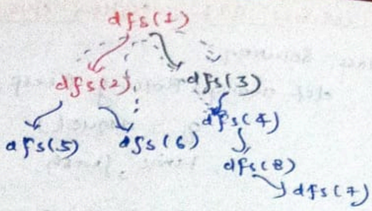


Starting Node = 1

→ 1 2 5 6 3 7 8 4.

changing starting node = 3

→ 3 4 8 7 1 2 5 6.



Till depth:

adj List :
This is how graph is stored

1 → {2,3}
2 → {1,5,6}
3 → {1,4,7}
4 → {3,8}
5 → {2}
6 → {2}
7 → {3,8}
8 → {4,7}

SN = 1 Vis:-

0	1	2	3	4	5	6	7	8
0	1	1 0	1 0	0	0	0	0	0

```
dfs (node)
{  vis[node] = 1
   list.add (node)
}
for (auto it : adj [node])
{ if(!vis[it]
    dfs(it);
}
```

Space Complexity
O(N) + O(N) + O(N)
≈ O(N) ★

Time Complexity
O(N) + (2×E)
★

code: :

```
public static void dfs (int node, boolean vis[],
ArrayList <ArrayList <Integer>> adj , ArrayList <Integer> ls)
{
    // marking current node is visited
      vis [node] = true;
      ls.add (node);
    // getting neighbour nodes
      for (Integer it : adj.get (node)) {
          if ( vis[it] == false ){
              dfs (it, vis, adj, ls);
          }
      }
   }
   // function to return a list containing the DFS traversal of graph
   public ArrayList <Integer> dfs of Graph (int V,
    Array list <Array List <Integer>> adj) {
       // Boolean array to keep track of visited vertices
         boolean vis[] = new boolean [V+1];
         vis[0] = true;
         ArrayList <Integer> ls = new ArrayList <> ();
         dfs (0, vis, adj, ls);
         return ls;
   }
```

PAGE No. DATE

→ Recursive DFS (using function call stack):

```
def dfs_recursive(graph, node, visited):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbor in graph[node]:
            dfs_recursive(graph, neighbor, visited)
```

→ Iterative DFS

```
def dfs_iterative(graph, start)
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)

            # Push neighbors in reverse to maintain order
            stack.extend(reversed(graph[node]))
```