# Design & Analysis of Algorithms

## Tutorial - 5

Name => MOHAK KALA

Section => D

Class R. No. => 29

University R. No. => 2014727

## Answers

① Using Breadth first Search (BFS), we can find the min. no. of nodes b/w a source node and destination node, while using Depth First Search, we can find if a path exists b/w 2 nodes.

Applications of BFS => - To detect cycles in a graph.
- Minimum distance Comparison

Applications of DFS => - To detect cycles in a graph
- To detect & compare multiple paths

② DFS : We use stack data structure to implement Depth First Search because order doesn't have much importance.

BFS : We use queue data structure to implement Breadth First Search because order matters in this case

③ Sparse Graph : No. of edges is close to minimal no. of edges

Dense Graph : The no. of edges is close to the maximal no. of edges.

## ④ Cycle Detection in BFS =>

1. Compute in-degree (No of incoming edges) for each of the vertex present in graph and count of nodes = 0
2. Pick all the vertices with in degree as 0 & add them to queue
3. Remove a vertex from the queue, then
   - Increment count by 1
   - Decrease in-degree by 1 for all neighbours
   - If in-degree of a neighbouring node is = 0 add to queue
4. Repeat step 3 until queue is empty
5. If no. of visited nodes is not equal to no. of nodes then graph has a cycle.

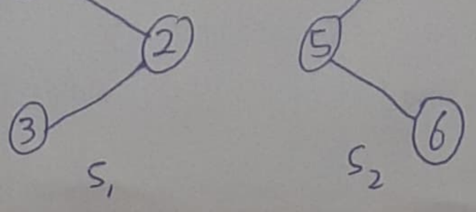
### Cycle detection using DFS

A similar process is done in DFS as well, but in DFS, we have the option of doing recursive calls, for vertices which are adjacent to the current node & are not yet ~~ver~~ visited. If recursive functions ~~node~~ return false, then graph does not have a cycle

## ⑤ Disjoint Set Data Structure =>

It is a data structure that is used in various aspects of cycle detection. This is literally a grouping of 2 or more disjoint sets

Eg =>



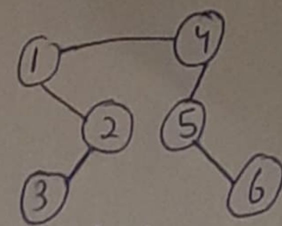

$S_1 = \{1, 2, 3\}$

$S_2 = \{4, 5, 6\}$

Operations =>

1. **Union** => Merge 2 sets when edge is added

$S1 \cup S2 = S3$ →



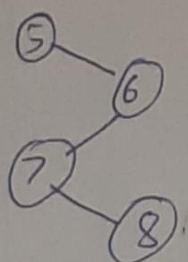

2. **Find ()** tells which element belongs to which set.

Find (1) = $S_1$ | Find (4) = $S_2$

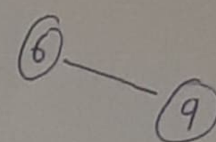
3. **Intersection** → Output another set as common elements.

$S_1 \cap S_2 = \{\phi\}$ , $S_4 \cap S5 = \{6\}$

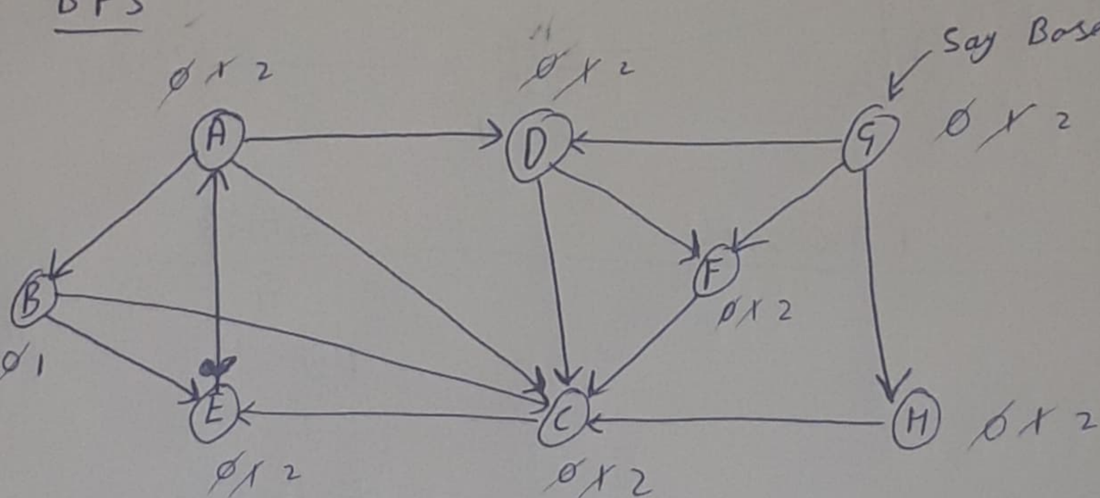
$S_4$




$S_5$ =>




(6) **BFS**




**Node** => G H F D C E A B

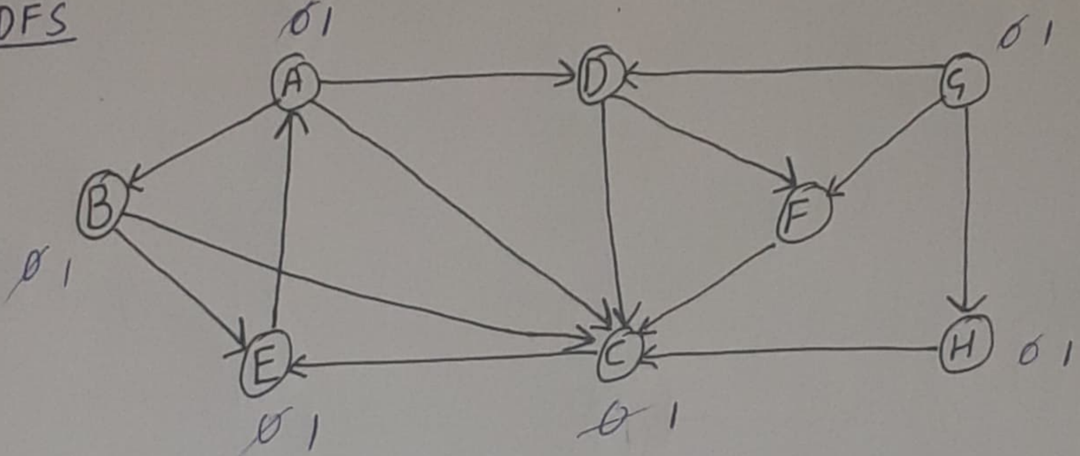
**Parent** => G G G H C E A

All nodes visited from source G

(4)

| Source | Destination | Path |
|---|---|---|
| G | A | G → H → C → E → A |
| G | B | G → H → C → E → A → B |
| G | C | G → H → C |
| G | D | G → D |
| G | E | G → H → C → E |
| G | F | G → F |
| G | H | G → H |

(6) DFS

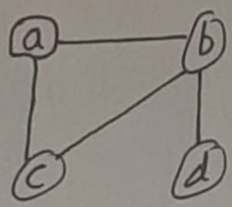



| Node | Processed | Stack |
|---|---|---|
| | | G |
| G | | D F H |
| D | | C F H |
| C | | E F H |
| E | | A F H |
| A | | B F H |
| B | | |
| F | | F H |
| H | | H |

Only to empty the stack

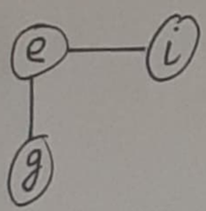
| Source | Destination | Path |
|---|---|---|
| G | A | G → D → C → E → A |
| G | B | G → D → C → E → A → B |
| G | C | G → D → C |
| G | D | G → D |
| G | E | G → D → C → E |
| G | F | G → F |
| G | H | G → H |
| G | | |

Ans-7 (i)



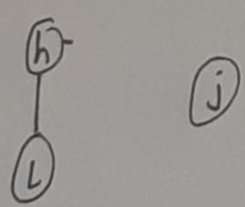
No (V) = 4
No. (CC) = 1

(ii)



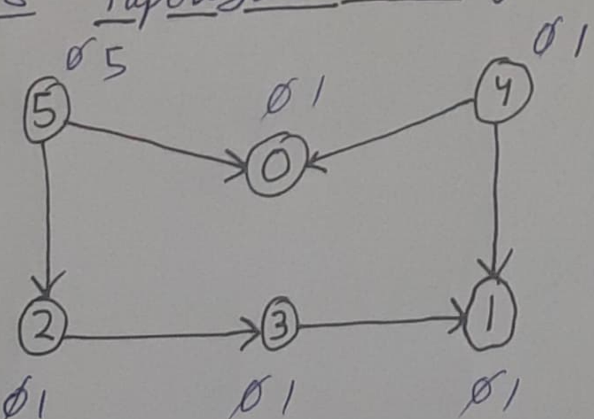
No. (V) = 3
No. (CC) = 1

(iii)



No. (V) = 3
No (CC) = 2

Ans-8 Topological Sorting



Adjacency list

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

Stack

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 3 | 2 | 4 | 5 |

Head →

Topological Sort ⇒
5 4 2 3 1 0

DFS ⇒ Stack → | 4 | 0 | 1 | 3 | 2 | 5 | Head ⟶

DFS → 5 → 2 → 3 → 1 → 0 → 4

(9) Application of Priority Queue ⇒

(i) Dijkstra's Algorithm ⇒ We need to use a priority queue here so that minimal edges can have higher priority.

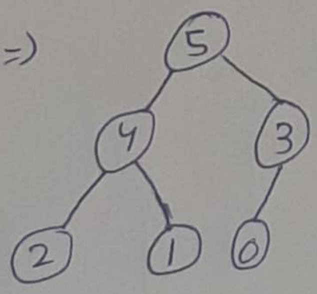
(ii) Load Balancing ⇒ Load balancing can be done from branches of higher priority to those of lower priority.

(iii) Interrupt Handling ⇒ To provide proper numerical priority to more important interupts.

(iv) Huffman Code → For data compresession in Huffman code

(10) Max. Heap where parent is bigger than both childs

Eg ⇒




Min. Heap : Where parent is smaller than both childs.

Eg ⇒