Sol 1) Using BFS we can find the minimum no. of nodes b/w a source node & destination node, while using DFS we can find if a path exists b/w two nodes.

Applications:-

BFS - To detect cycles in a graph, min distance comparison, eg ps navigators.

DFS - To detect & compare multiple paths, detect cycles in a graph.

Sol 2) - DFS - We use stack to implement DFS because order doesn't has much importance.

BFS - We use queue Datastructure to implement BFS because order matters in this case.

Sol 3) → Sparsegraph - No. of edges is close to minimal no. of edges.

Dense graph - No. of edges is close maximal no. of edges.

Sol 4) - Cycle Detection in BFS -

1) Compute indegree (no. of incoming edges) for each of the vertect present in a graph & count no. of nodes = 0

2) Pick all the vertices with indegree as 0 & add them to queue

3) Remove a vertex from the queue, then.
- Increment count by 1
- Decrease indegree by 1 for all neighbours.
- If indegree of neighbouring node is = 0, add to queue.

4) Repeat 3 until queue is empty.

5) If no of ~~the~~ visited nodes is not equal to no of nodes, then graph has a cycle.

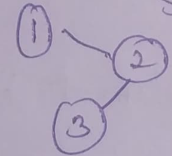
Cycle detection in DFS-

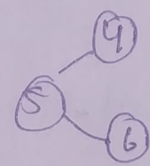
- A similar process is done in DFS as well, but in DFS we have the option of doing recursive calls for vertices which are adjacent to the current node & are not yet visited. If recursive function returns false, then graph does not have a cycle.

<u>Sol 5)</u> Disjoint set Data structure-

It is a DS that is used in various aspects of cycle detection. This is literally grouping of two or more disjoint sets.

eg: S_1 S_2



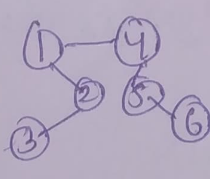


$S_1 = \{1, 2, 3\}$
$S_2 = \{4, 5, 6\}$

<u>operations</u>-

① Union - Merge two sets when edge is added.

$S_1 \cup S_2 = S_3 \rightarrow$

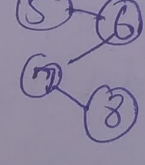


③ Find() tells which element belongs to which set
$Find(1) = S_1$ | $Find(4) = S_2$

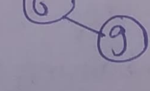
③ Intersection outputs another set is common elements.

$S_1 \cap S_2 = \{\phi\}$ $S_4 \cap S_5 = \{6\}$

S_4 -

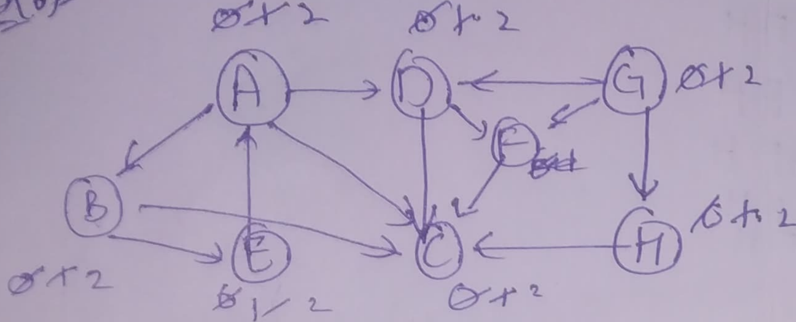


$\xleftarrow{S_5}$
Intersection



BFS

Sol(6):

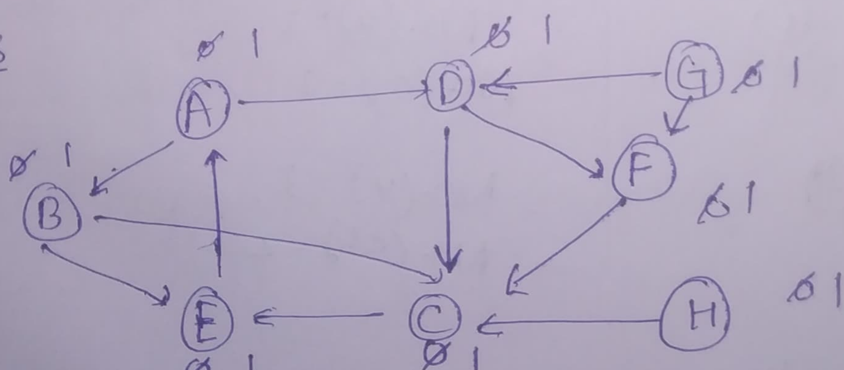


Nodes	G	H	F	D	C	E	A	B
Parent		G	G	G	H	C	E	A

All visited from Source G

Source	Destination	Path
G	A	G → H → C → E → A
G	B	G → H → C → 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
G		

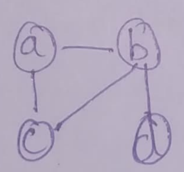
DFS



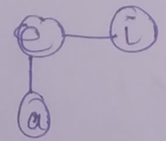
Nodes Processed	Stack
G	G
D	DFH
C	CFH
E	EFH
A	AFH
B	BFH
	FH

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.

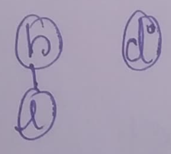
A

Sol 7) → ① 

No(v) = 4
No(cc) = 1

② 

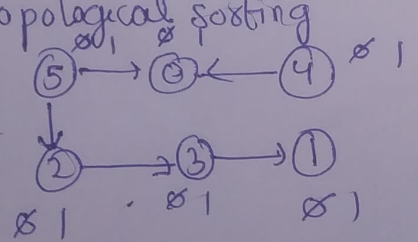
No.(v) = 3
No.(cc) = 1

③ 

No.(v) = 3
No.(cc) = 2

At

Sol 8) → Topological sorting



Adjacency List

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

Stack | 0 | 1 | 3 | 2 | 4 | 5 |

Topological = 5 4 2 3 1 0

DFS Stack → | 4 | 0 | 1 | 3 | 2 | 5 | ← Head →

DFS → 5 → 2 → 3 → 1 → 0 → 4

Sol 9) Applications of priority Queue

(1) Dijkstra's Algo - We need to use a priority queue there so that minimal edges can have higher priority.

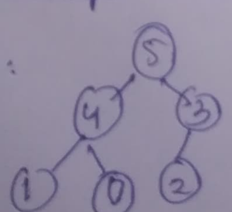
(2) Load Balancing - Load balancing can be done from branches of higher priority of those of lower priority.

(3) Interrupt - To provide proper numerical priority to Handling imp. interrupt.

(4) Huffman code - Data compress in huffman code.

Sol 10) - Max heap - Where parent is bigger than both children.

eg:



Heap - Where parent is smaller than both children

eg -