Tutorial Sheet - 5

Mridul Gupta
H - 38

Sol 1 Using BFS, we can find the min 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 comparisons, gps navigator.
DFS : to detect & compare multiple paths, detect cycle in a graph.

Sol 2:- DFS : We use stack to implement DFS because "order doesn't has much importance".
BFS : we use queue D.S to implement BFS because "order matters in this case".

Sol 3: Sparse graph → No of edges is close to minimal no. of edges.
Dense graph : No of edges is close to maximal no. of edges.

Sol 4: Cycle Detection in BFS :
1) Compute in degree (no. of incoming edges) for each of the vertex present in 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 in degree of a neighbouring node is = 0, add, to queue

4) Repeat 3 until queue is empty

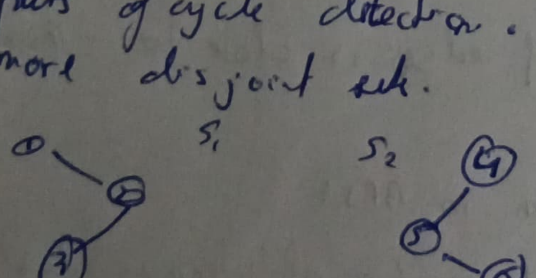
5) If no. of visited nodes is not equal to no. of nodes then graph has a cycle.

Cycle detection 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 funcn return false, then graph does not have a cycle.

Sol 5: Disjoint set DS:- It is a DS that is used in various aspects of cycle detection. This is literally grouping of two or more disjoint sets.

eg:

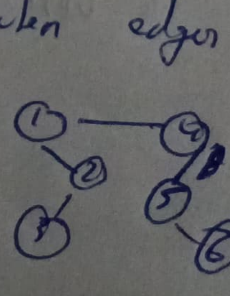



$S_1 = \{1, 2, 3\}$
$S_2 = \{4, 5, 6\}$

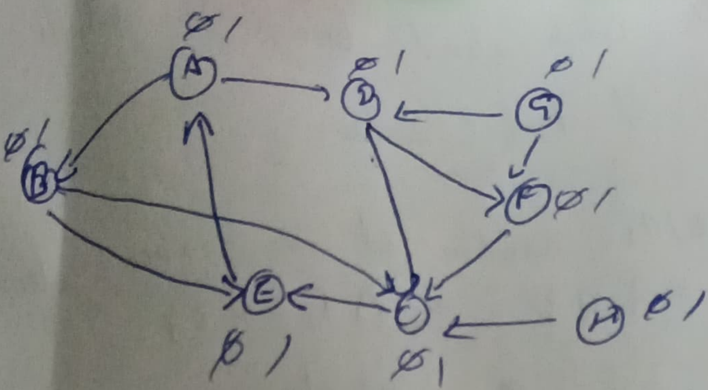
Operations.

① Union :- merge two sets when edges is added

$S_1 \cup S_2 = S_3$ →



# DFS




| Nodes 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 H |

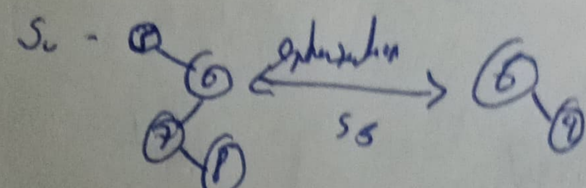
| Source | Destination | Path |
| --- | --- | --- |
| G | A | G→D→C→E→A |
| G | B | G→D→C→E→~~A~~A→B |
| G | C | G→D→C |
| G | D | G→D |
| G | E | G→D→C→E |
| G | F | G→F |
| G | H | G→H |

③ Find() - tells which element belongs to which set

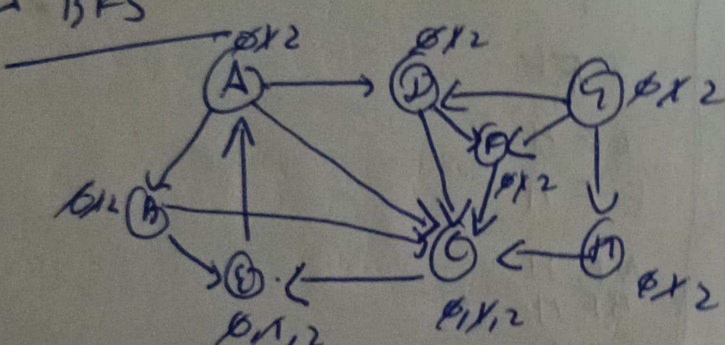
Find (1) = $S_1$ / Find (4) = $S_2$

④ Intersection - O/Ps another set as common elements

$S_1 \cap S_2 = \{\Phi\}$ . $S_4 \cap S_5 = \{6\}$




Sol(6) BFS



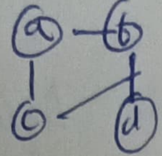
| node | G | H | F | D | C | E | A | B |
|---|---|---|---|---|---|---|---|---|
| Parent | | G | G | G | H | C | E | A |

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 |

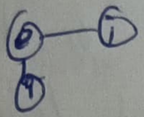
Sol 7

(1)



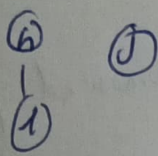
No(V) = 4
No(C.C) = 1

(2)



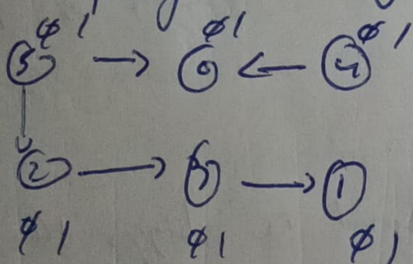
No(V) = 3
No(CC) = 1

(3)



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

Sol 8 Tropological Sorting



Adjacency list

0 →
1 →
2 → 3
4 → 0, 1
5 → 2, 0

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

tropological = 5 4 2 3 1 0

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

DFS → 5 → 2 → 3 → 1 → 0 → 4

Sol 9) Appⁿ of Priority Queue ->

① Dijkstra algo -> We need to use a priority queue here so that minimal edges can have higher priority

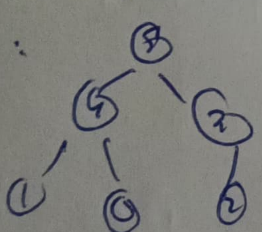
② Load Balancing => load balancing can be done from branches of higher priority to those of lower priority

③ Interrupt Handling -> to provide proper numerical priority to m imp interrupt.

④ Huffman codes for data compression in Huffman co-

Sol 10) max heap => where parent is bigger than both child

eg:



min heap -> where parent is smaller than both chil

eg