Tutorial - 5

Piyush Kumar
Roll No. - 41
Sec - H

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

- Applications -

  BFS - To detect cycles in graph, min distance comparison, gps navigation.

  DFS - To detect a cycle in a graph.

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

BFS - We use queue data structure 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 -

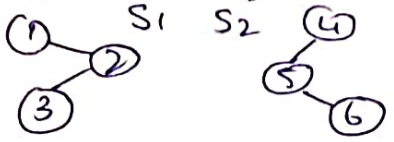
1. Compute in degree (no. of incoming edges) for each of the vertex present in graph & count no. of nodes = 0.
2. Pick all the vertices with indegree as 0 & add them to queue.
3. Remove a vertex from the queue, then
   - increment count by 1.
   - Decrease indegree by 1 for all neighbours.
   - If in-degree 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 function returns false, then graph does not have a cycle.

Q15) 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.

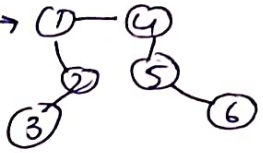
e.g : 

$S1 = \{1, 2, 3\}$

$S2 = \{4, 5, 6\}$

Operations :-

① Union - Merge two sets when edge is added

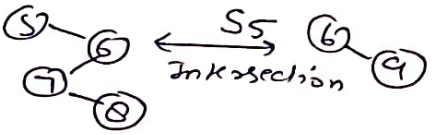
$S_1 \cup S_2 = S_3 \rightarrow$ 

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

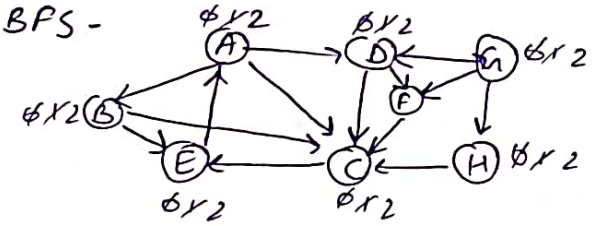
find(1) = S_1 find(4) = S_2

③ Intersection - outputs another set is common elements.

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

S_4 - 

Q16) BFS -

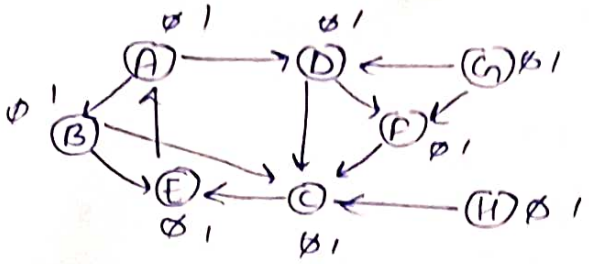


Node	G	H	F	D	C	E	A	B
Parent		G	G	G	H	C	E	A

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

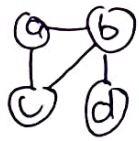
• DFS



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

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

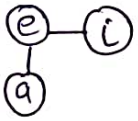
Sol 7 (1)



No. (V) = 4

No. (CC) = 1

(2)



No. (V) = 3

No. (CC) = 1

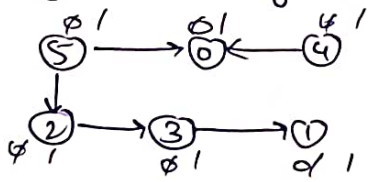
(3)

(J)

No. (V) = 3

No. (CC) = 2.

Sol 8 Topological Sorting



Adjacent 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 |

DFS → 5 → 2 → 3 → 1 → 0 → 4

Sol9) Applications of Priority Queue -

1) Dijkstra's algo : we need to use a priority queue here so that minimal edges can have higher priority.

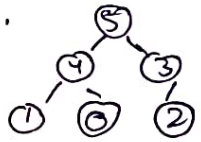
2) Load Balancing : Load balancing can be done from branches of higher priority to those of lower priority.

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

4) Huffman Code - For data compression in Huffman code.

Sol10 Max Heap → where parent is bigger than with children.

e.g.



Min Heap → where parent is smaller than both children.

e.g.