Tutorial - 5

Name : Karan Maurya
section : F
Class Roll no : 11

Ans.1.

BFS

-> BFS stands for breadth first search.

-> It uses Queue DS for its implementation.

-> Use to find single source shortest path in unweighted graph.

-> More suitable for searching vertices which are closer to the given source.

-> It consider all neighbors first & therefore not suitable for decision making trees used in games or puzzles.

-> Time Complexity when adjacency list is use : O(V+E) & when matrix is used then it become $O(V^2)$.

-> No backtracking.

-> Application : bipartite graph & shortest path, etc.

DFS

-> DFS stands for Depth first search.

-> It uses stack DS for its implementation.

-> Use to transverse ~~the~~ ~~graph~~ all the reachable vertices.

-> DFS is more suitable when there is a solution away from source.

-> DFS is more suitable for game or puzzle problems we make a decision, then explore all paths through this decision.

-> Time Complexity if adjacency list is used O(V+E) & if matrix is used then $O(V^2)$.

-> Backtracking

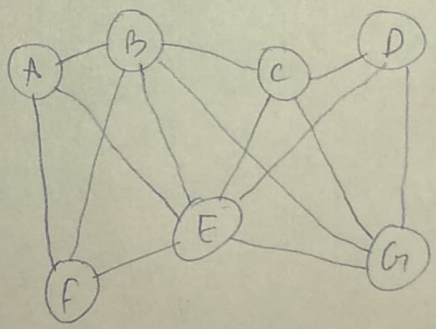
-> Topological sort, Acyclic graph, etc

Ans. 2.

-> As in DFS, we go towards the depth of the g & then after reaching the end, it backtracks to its previous nodes. As for backtracks stack is used. So, DFS uses stack for its implementation.

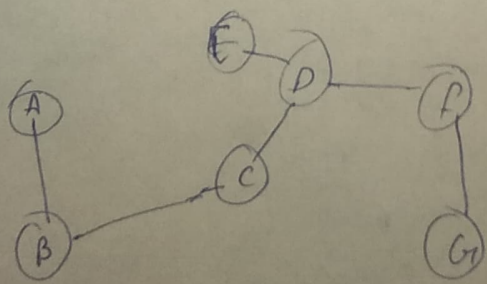
-> As in BFS, all the neighbour nodes are traversed first ~~then~~ the ~~next~~ i.e traversing occur in a level order. So, to make it possible, queue DS is used.

Ans 3. -> Dense graphs has many edges b/w nodes.



dense graph

-> Sparse graph has few edges b/w nodes.



sparse graph

-> If the graph has n vertices -> Maximum no. of edges = n^2

-> In dense graph no. of edges is close to n^2.

-> In sparse graph no. of edges is close to n.

Detecting Cycle by Using DFS :

```
bool isCyclic( int v, vector<int> adj[])
{
    vector<bool> visited (v, false);
    bool FLAG = false;
    for( int i=0; i<v; i++)
    {
        visited[i] = true;
        for( int j = 0; j< adj[i].size(); j++)
        {
            FLAG = isCyclic_util ( adj, visited, adj[i][j]);
            if(FLAG == true)
                return true;
        }
        visited[i] = false;
    }
    return false;
}
bool isCyclic_util ( vector<int> adj[], vector<bool> visited, int curr)
{
    if( visited[curr] == true)
        return true;
    visited[curr] = true;
    bool FLAG = false;
    for( int i=0; i< adj[curr].size(); i++)
    {
        FLAG = isCyclic_util (adj, visited, adj[curr][i]);
        if (FLAG == true)
            return true;
    }
    return false;
}
```

finds

BFS

1. Compare in degree (no. of incoming edges) for each of two vertex present in graph & count no. of nodes.
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.
   - decrement in degree by 1 for all neighbour.
   - If in degree of a neighbouring node is = 0 add it 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.

~~DFS~~

Ans. 5 Disjoint set Datad structure:

→ It is a DS that is used in various aspects of cycle detection. This is litterally grouping of two or more disjoint set.

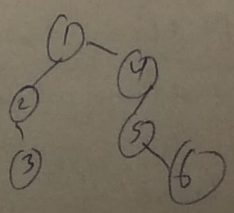
→ Operation available:

(1). Union: Merge two set when edge is added.

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

add edge 1 & 4

$S_1 \cup S_2 =$



Finds tell which element belongs to which set.

find(1) = S_1

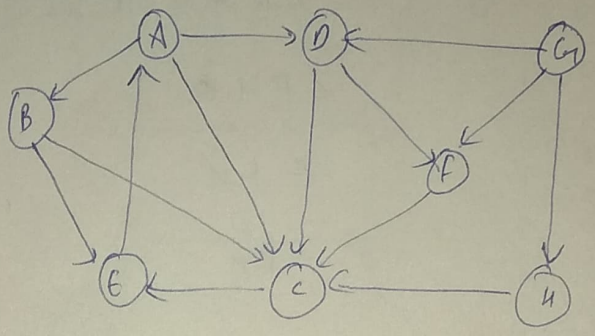
find(4) = S_2

(3). Intersection : Output the common elements b/w two sets.

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

$S_1 \cap S_2 = \phi$.

Ans. 6

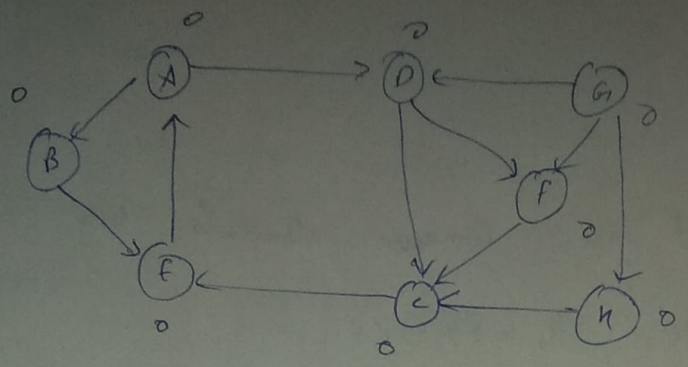


BFS

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

All visited nodes from source G :

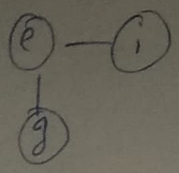
source	destination	Path
G	A	G → H → C → E → A
G	B	G → H → C → E → A → B
G	C	G → H → C
G	D	G → D
G	E	G → H → C → E
G	F	G → F
G	H	G → H



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

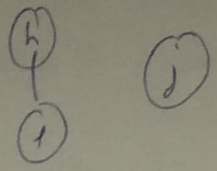
(2)



No. of (V) = 3

No. of (cc) = 1.

(3).

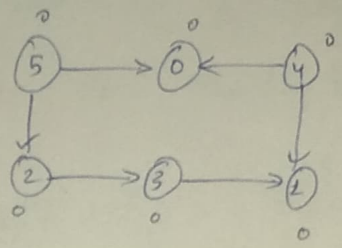


No. (V) = 3

No. of (cc) = 2

Ans. 8

Topological sort :



List

0

1

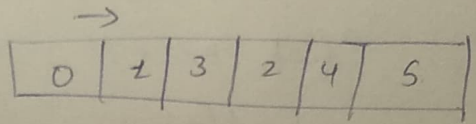
2 → 3

3 → 1

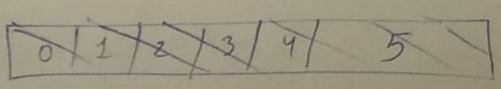
4 → 0 → 1

5 → 0 → 2

stack :



DFS :



if 0 is source out : 0

if 5 is source out : 5, 0, 2, 3, 1

if 4 is source out : 4, 0, 1,

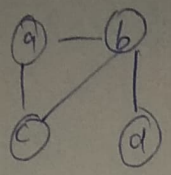
if 3 is source out : 3, 1

if 2 is source out : 2, 3

if 1 is source out : 1

Ans. 7.

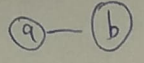
(1)



$V = \{a, b, c, d\}$

add edge a,b

find(a) = 0 , find(b) = 0



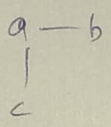
, $S = \{a, b\}$

add edge a,c

find(a) = S,

find(c) = 0

$S = \{a, b, c\}$



add edge c,b

find(c) = S

find(b) = S

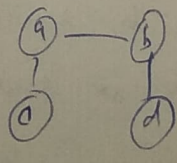
cycle detected

add edge b,d

find(b) = S

find(d) = 0

$S = \{a, b, c, d\}$



No. of vertices : 4

No. of connected component : 1.

Ans. 9: Application of heap

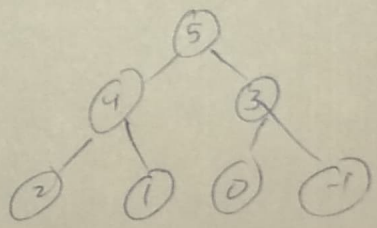
1. Dijkstra's algo: we need to use a priority queue here so that minimal edges can have higher priority.

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 Handling imp. interrupt.

4. Huffman code: for data comparison in huffman code.

Ans. 10: Max Heap: where parent is bigger than their children.



Min Heap: where parent is smaller than children.