Date.
Page No.

Assignment-3

Q:) what is the difference b/w DFS & BFS.? Write applications

A:) DFS & BFS both are graph traversal algorithms

- BFS stands for Breadth first search. It uses a Queue data structure. that follows First in First out

- In BFS one vertex is selected at a time when it is visited & marked → then its adjacent nodes are visited & stored in a queue. It is slower than DFS.

→ BFS applications → used in various applications such as bipartite graph & shortest path. etc.

DFS → stands for depth first search. It uses stack data structure. In this first visited vertices are pushed to stack & find if there are no vertices then visited vertices are popped.

DFS applications → It is used in various applications such as acyclic graph & topological order.

BFS	DFS
→ uses queue	→ uses stack
→ BFS considers all neighbours first & therefore not suitable for decision making trees used in games or puzzles.	→ In DFS we make a decision then explore all paths through this decision.
→ T.C → list → $O(V+E)$ matrix = $O(V^2)$	T.C → list → $O(V+E)$ matrix → $O(V^2)$
→ Here siblings are visited before children	→ Here children are visited before siblings.
→ No backtracking	→ Used backtracking since recursive

Date
Page No.

DFS applications

→ DFS on unweighted graph makes MST for all pair of shortest path tree.

→ We can detect cycles in graph using DFS.

→ Using DFS we can find path b/w 2 given vertices u&v.

→ Topological sort using DFS

BFS applications

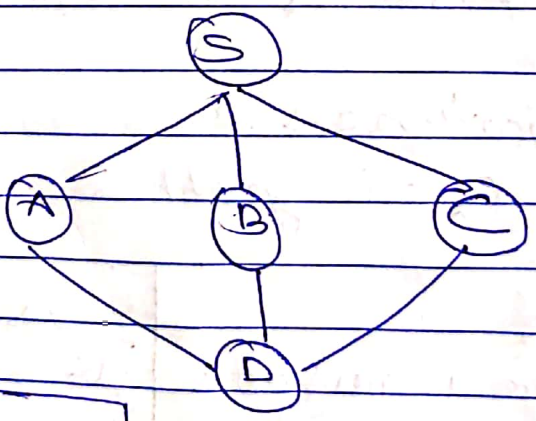
→ In peer to peer N/w BFS is used to find all neighbour nodes

→ In GPS " " " "

→

Q) Data structure used to implement BFS & DFS algo

→ For BFS we use Queue since we visit every neighbour of the node first and then check for the neighbour of neighbour nodes & if there are any we mark them visited & push them in queue.

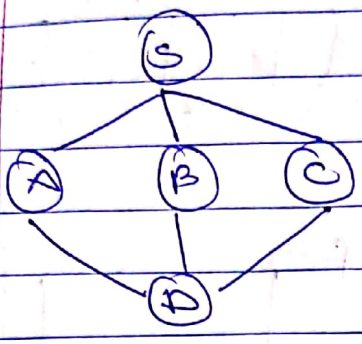


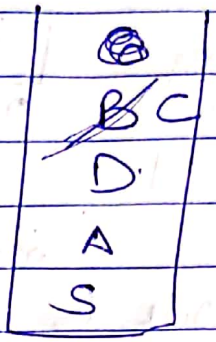
A	B	C	D

node	S	A	B	C	D
Parent	-	S	S	S	A

Date.
Page No.

In DFS we use stack because it traverses the graph in depthward motion & uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



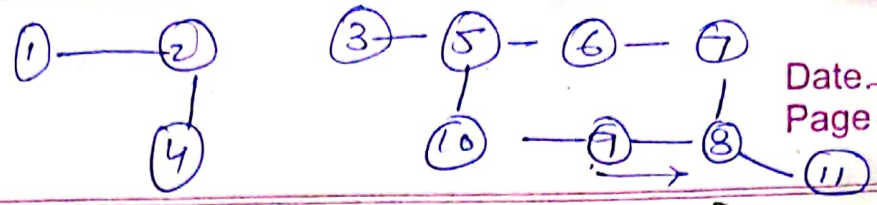


Q2-
A queue (FIFO first in first out) data structure is used by BFS. You mark any node in graph as root & start traversing the data from it. BFS traverses all the nodes in the graph & keeps marking them as completed. BFS visits an adjacent unvisited node, marks it as visited & inserts it into queue.

Since DFS algo traverses graph in depthward motion and uses a stack to remember to get the next vertex in start a search, when a dead end occurs in any iteration.

Q3. Sparse Graph:- A graph in which the number of edges is much less than the possible number of edges.
for sparse graph use adjacency list.

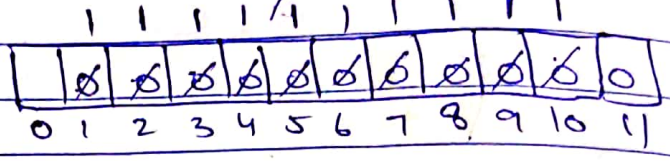
Dense Graph → A graph in which the no° of edges is close to the maximal no° of edges.
for dense use adjacency matrix.
→



Date
Page No.

Q4. Detecting cycle using DFS or BFS ?

A) Using BFS & DFS we can detect cycle. In BFS if any of the adjacent node has been visited previously then there is a cycle.

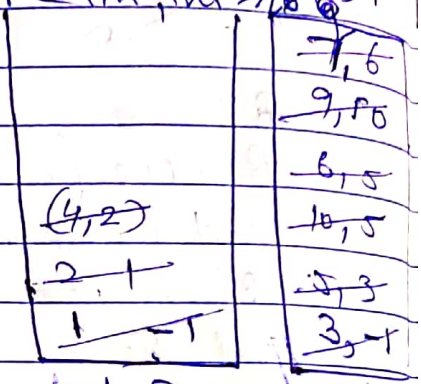


making a visited array

queue < pair < int, int > > 8, 9

for (7,6) → 8 node is already visited but it is not the previous node.

Since 8 has already been mark visited when when the 9 node 9 was pushed to queue with its adjacent nodes marked 10,8 visited



∴ there is a cycle.

```
bool check for cycle (vector<int> adj[], int source, bool vis[])
{
    queue<pair<int,int>> q;
    visited[source] = true;
    q.push_front(source, -1);
    while(!q.empty())
    {
        int node = q.front().first(); int par = q.front().second();
        q.pop_front();

        for (auto child : adj[node])
        {
            if(!vis[child])
            {
                vis[child] = true;
                q.push_front(child, node);
            }
            else if (child != par)
                return true;
        }
    }
    return false
}
```

Date.
Page No.

Q5) Disjoint set data structure?

A→ It allows to find whether the 2 elements are in same set or not.

→ The disjoint set can be defined as subsets where there is no common element b/w the 2 sets.

Eg = S1 = {1, 2, 3, 4} S2 = {5, 6, 7, 8}
S1 ∩ S2 = {ϕ}.

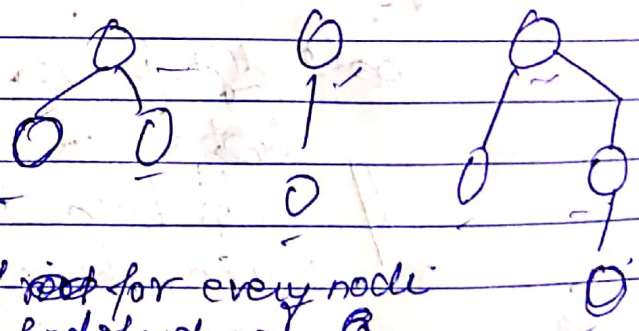
Operations that can be performed.

```
void make (int v)
{
    Parent[v] = v;      stores parent.
    size [v] = 1 ;      stores size of each grp
}

int find (int v)            // find parent.
{
    if (v== Parent[v]) return v;
    // for path compression.
    return parent[v] = find(Parent[v]);
}
                left     right
void union (int a, int b)
{
    a= find (a);
    b= find (b);

    if (a! = b)
// now union by their size means adding small tree to large tree. (right)
// basically we want to join small to large tree.      Eg. fig
```

Date.
Page No.

T.C = O(α(n)) reverse Ackermann fn.

```
if (sizeof(a) < sizeof(b))
{
    Parent[a] = b;
    size of[b] += size of[a]
}
if (size [b] < size [a])
{
    Parent[b] = a;
    sizeof[a] += size[b];
}

int main() {
    int n, k          // no of nodes, k edges
    cin >> n >> k
    for(int i=1; i<=n; i++) {
        make(i);
    }
    while (k--)
    { int u v;
      cin >> u >> v;
      union(u,v)
    }
    int connected_ct = 0;
    for(int i=1; i<=n; i++)
    { if find[i] = [i]   // if root for every node
                         find of node == root node
                         then they are connected
      connected_ct ++;
    }
```



Date
Page No.

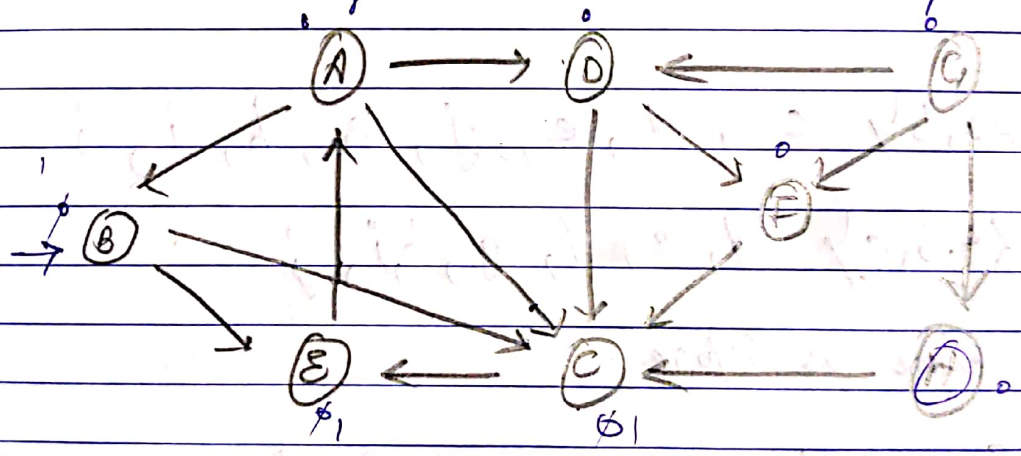
→ Find → Can be implemented by recursively traversing the parent array until we hit a node who is parent to itself.

→ Here we can perform path compression too. as it speeds up the data structure by compressing the height of tree.

line return Parent[v] = find (Parent[v])
does Path comp...

→ Union → It takes as input, 2 elements. And finds the representation of their sets using the find operation, & puts either one of the tree (representing the set) under the root node of other tree, effectively merging the trees & sets according to smaller & larger.

Q6)



BFS

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

Path from B → F

B → E → A → D → F

G, H nodes can't be visited

Date.
Page No.

DFS:

node Processed	stack
—	B
B	C
C	E
E	A
A	D
D	F
F	-

Path →
B → C → E → A → D → F

Q.) Find no of connected components & vertices in each component using disjoint set data structure?

A) V = {a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

E = {a,b} (a,c) (b c) (b,d) (e f) (e g) (h i) (j

(a,b) → {a,b} c, d, e, f, g, h, i, j

(a,c) {a,b,c} d, e, f, g, h, i, j

(b c) same as before.

(b d) {a,b,c,d} e, f, g, h, i, j

(e f) {a,b,c,d} {e,f} g, h, i, j

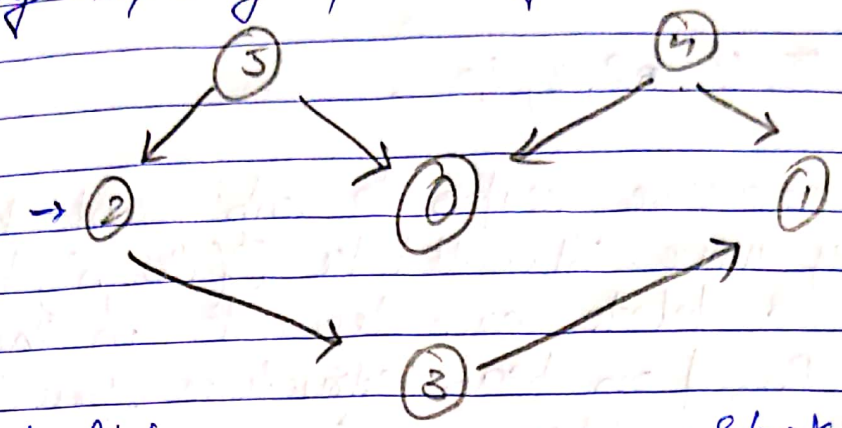
(e g) {a,b,c,d} {e,f,g} h, i, j

(h i) {a,b,c,d} {e,f,g} {h,i} {j}

no of connected component = 3.

Date
Page No.

Q) Apply Topological sorting & DFS on graph



adj list

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 0, 2

stack

step ① Topo_s (0) visited[0] = true;
list, No more recursion call.
stack [0]

② Topo_s (1) vis[1] = true.
no more recursion call
stack [0, 1]

③ Topo_s (2) vis[2] = true
Topo_s (3) vis[3] = true.
no more recursion call.
stack [0, 1, 3, 2]

④ Topo_s (4), vis[4] = true; (0,1) already visited
[0, 1, 3, 2, 4] ,no more recursion call.

⑤ Topo_s (5) vis[5] = true (0, 2) already visited.
no more recu call.
[0, 1, 3, 2, 4, 5]

Date.
Page No.

now popping from top to bottom:

- 5, 4, 2, 3, 1, 0

Ans → We can use heaps to implement the priority queue. It will take O(logn) time to insert & delete each element in the priority queue. Based on heap structure, priority queue also has 2 types max-priority queue & min-priority queue.

Some algo. where we need to use priority queue

① Dijkstra's algorithm shortest path using priority queue: when the graph is sorted in form of list or matrix, PQ can be used to extract minimum efficiently.

② Prim's algorithm → It is used to implement prim algorithm to store keys of nodes & extract minimum key node at every step.

③ Data compression → Used in huffman coding which is used to compress data.

Date
Page No.

Q10→

Min-heap	Max heap.
① The key present at root node is less than or equal to among the keys present at all children	① The key present at root node is greater than or equal to children keys
② The minimum key at root	② Maximum key at root
③ Uses ascending priority	③ Uses descending priority
④ The smallest element is the first to be popped from heap.	④ The largest element is the first to be popped from heap.
⑤ In construction min heap smallest element has priority	⑤ Largest element has priority.