TUTORIAL - 5

Ans (1)

BFS	DFS
→ BFS stands for breadth first search.	→ DFS stands for depth first search.
→ BFS uses queue to find the shortest path.	→ DFS uses stack to find the shortest path
→ BFS is better when target is closer to source	→ DFS is better when target is far from source.
→ As BFS considers all neighbours so it is not suitable for decision tree used in puzzle games	→ DFS is more suitable for decision, we need to traverse further to argument the decision. If we needs the conclusion
→ BFS is slower than DFS	→ DFS is faster than BFS
→ TC of BFS = O(V+E) where V is vertices and E is edges	→ TC of DFS is also O(V+E) where V is vertices and E is edges

Applications of DFS :-

→ If we perform DFS on unweighted graph, then it will create minimum spanning tree for all pair shortest path tree.

→ We can detect cycles in a graph using DFS. If we get one back-edge during BFS, then there must be one cycle.

→ Using DFS we can find path between two given vertices u and v.

→ We can perform topological sorting is used to scheduling jobs from given dependencies among jobs. Topological sorting can be done using DFS algorithm.

→ Using DFS, we can find strongly Connected components of a graph. If there is a path from each vertex to every other vertex, that is Strongly Connected

Application of BFS

→ Like DFS, BFS may also used for detecting cycles in a graph
→ Finding shortest path and minimal spanning tree in unweighted graph
→ Finding a route through GPS navigation system with minimum number of crossings
→ In networking finding a route for packet transmission
→ In building the index by search engine crawler.
→ In peer to peer networking BFS is the find neighbouring node
→ In garbage Collection BFS is used for copying garbage

Ans(a) BFS (Breadth first search) uses queue data structure for finding the shortest path
→ DFS (Depth first search) uses stack data structure
→ A queue (FIFO - First in First Out) data structure is used by BFS. You mark any node in the graph as root and start traversing the data from it. BFS traverses all the nodes in the graph and keeps dropping them as Completed. BFS visited an adjacent unvisited node, marks it as done, and inserts it into a queue.
→ DFS algorithm traverse a graph in a depth ward and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

• Sparse Graph :- A graph in which the number of edges is much less than the possible number of edges.

• Dense Graph :- A dense graph is a graph in which the number of edges is close to the maximal number of edges.

→ If the edge graph is sparse, we should store it as a list of edge. Alternatively, if the graph is dense, we should store it as adjacency matrix.

The existence of a cycle in directed and undirected graphs can be determined by whether depth-first search (DFS) finds an edge that points to an ancestors of the current vertex (it contains a back edge). All the back edges which DFS skips ones are part of cycles.

* Detect cycles in a directed Graph

→ DFS can be used to detect a cycle in a graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge that is from a node to itself (self-loop) one of its ancestors in the tree produced by DFS.

→ For a disconnected graph, get the DFS forest as output to detect cycle check for a cycle in individual by checking back edge. To detect a back edge, keep track of vertices currently in the recursion stack of function for DFS traversal. If a vertex is reached that is already in the recursion stack, then there is a cycle in the tree. Use stack [] array to keep track of vertices in recursion stack.

Detect cycle in an undirected graph

→ Run a DFS from any unvisited node. DFS can be used to detect a cycle in a graph. DFS for a connected graph produced a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is joining a node to itself or one of its ancestors in the tree produced by DFS. To find the back edge to any of its ancestor in the tree keep a visited array, if there is a back edge to any visited node then there is a loop and return true.

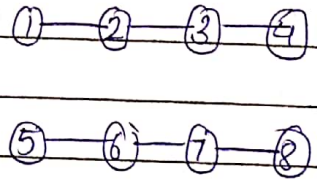
Vol(5) Disjoint Set Data Structure

→ It allows to find out whether the two elements are in the same set or not efficiently

→ The disjoint set can be defined as the subsets where there is no common elements b/w the two sets

Eg $S_1 = \{1,2,3,4\}$

$S_2 = \{5,6,7,8\}$



Operation performed

(i) find :- can be implemented by recursively traverses the parent array until one hit node who is parent to itself int find(int i)

```
{
    if (parent[i] == i)
        return i;
    else
        return find(parent[i]);
}
```

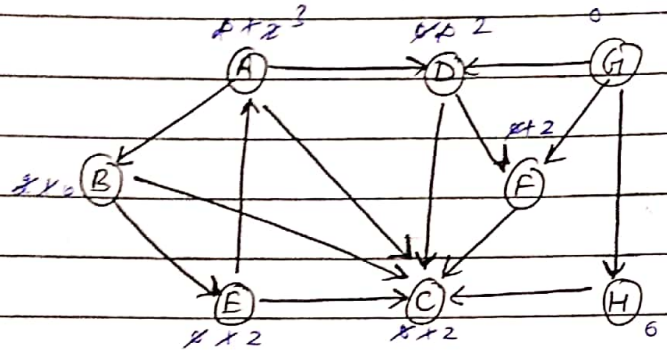
(ii) Union :- It takes as input, two elements and find the representation of their sets using the find operation and finally puts either one of the trees under root node of other tree, effectively merge the tree and set

```
Void union(int i, int j)
{
    int irep = this.find(i);
    int j_rep = this.find(j);
    this.parent[i_rep] = j_rep;
}
```

(iii) Path Compression :- It speeds up the data structure by compressing the height of the tree. It can be achieved by inserting a small caching mechanism into find operation.

```
int find (int i)
{
    if (Parent[i] == i)
    {
        return i;
    }
    else
    {
        int result = find (Parent[i]);
        Parent[i] = result;
        return result;
    }
}
```

Sol (6)



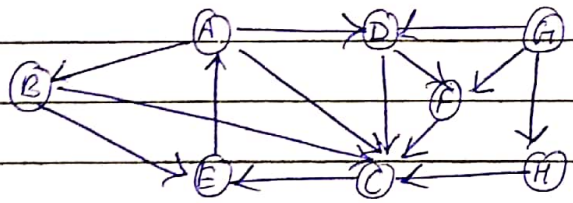
BFS :- Node - B E C A D F

Parent - B B E A D

Unvisited node :- G & H

Path :- B → E → A → D → F

DFS :



Node processed B B C E A D F

Stack B CE FE AE DF FE F

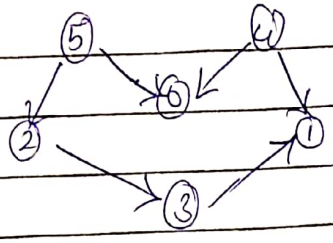
Path : B → C → F → A → D → F

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

Number of connected Components = 3

Topological Sort :-



Adjacent list 0 → 0
1 → 1
2 → 3
3 → 1
4 → 0,1
5 → 2,0

Visited :-

false	false	false	false	false	false
0	1	2	3	4	5

Stack (empty)

Step 1 :- Topological Sort (0), Visited [0] = true
list is empty, no more recursion call.
stack [0]

Step 2 :- Topological Sort (1), visited [1] = true
list is empty. No more recursion call
stack [0|1]

step 2 :- Topological sort (2), visited [2] = true
Topological sort (2), visited [3] = true
'4' is already visited, no more recursive call
stack 0 1 3 2

step 4 :- Topological sort (4), visited [4] = true
'0', '1' are already visited, no more recursive call
stack 0 1 3 2 4 1

step 5 :- Topological sort (5), visited [5] = true
'2' '0' are already visited. No more recursive call
stack 0 1 3 2 4 5

step 6 :- Print all elements of stack from top to bottom
5, 4, 2, 3, 1, 0

Ans) We can use heap to implement the priority queue.
It will take o(log n) time to insert and delete each element in the priority queue. Based on heap structure, priority queue is also two types - max priority and min priority queue.

Some algorithms where we need to use priority queue are-

(i) Dijkstra's :- Shortest path algo. uses priority queue when the graph is stored in the form of adjacency list or matrix. priority queue can be used to extract minimum efficiently when implementing Dijkstra's Algorithm.

(ii) Prim's Algorithm :- It is used to implement Prim's Algorithm to store keys of nodes and extract minimum key node at every step.

(iii) Data Compression :- It is used in Huffman's Code which is used to compress data

Ans (10)

Min Heap	Max Heap
→ In a min heap the key present at the root must be than or equal to among the keys present at are its Children	→ In a max heap the key present at the root node must be greater than or equal to among the keys present at all of its children
→ The minimum key element present at the root uses the ascending priority	→ the maximum key element present at the root uses descending priority
→ In a Construction of min heap the smallest element has the priority	→ In the construction the largest element has priority
→ The smallest element is the first to be poped from the heap	→ the largest element is the first to be popped from the heaps