Date.
Page No.

Tutorial-5

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 consider all neighbour so it is not suitable for decision tree used in puzzle games.	• DFS is more suitable for decision tree. As with one decision, we need to traverse further to argument the decision. If we reach the conclusion, we won.
• BFS is slower than DFS.	• DFS is faster than BFS.
• TC of BFS = O(V+E) where V is vertices & E is edges	• TC of DFS is also O(V+E) where V is vertices & E is edges.

Date.
Page No.

⇒ Application 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 & v.
- We can perform topological sorting is used to scheduling jobs from given dependencies among jobs. Topological sorting can be done using DFS algorithm.

⇒ 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 no. of crossings.
- In networking finding a route for packet transmission.

Date.
Page No.

- In building the index by search engine crawlers.
- In peer-to-peer networking, BFS is to find neighbouring node.

2. 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 traverse all the nodes in the graph & keeps dropping them as completed. BFS visits an adjacent unvisited node, marks it as done, & inserts it into a queue.

→ DFS algorithm traverse a graph in a depthward motion & uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Date
Page No.

3. 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 no. of edges is close to the maximal no. of edges.
- If the graph is sparse, we should store it as a list of edges. Alternatively, if the graph is dense, we should store it as a adjacency matrix.

4. The existence of a cycle in directed & 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 over are part of cycles.
* Detect cycle 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

Date.
Page No.

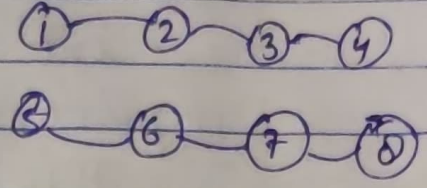
itself (self-loop) are one of its ancestors in the tree produced by DFS.

* Detect cycle in a undirected graph - Run a DFS from every unvisited node. DFS can be used to detect a cycle in a graph. DFS for a connected graph produce 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 / self-loop or one of its ancestor in the tree produced by DFS. To find the back edge to any of its ancestor keep a visited array & if there is a back edge to any visited node then there is a loop & return true.

5. Disjoint set data structure - It allows us 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 element b/w the

Date.
Page No.

two sets.
eg. S1 = {1,2,3,4}
S2 = {5,6,7,8}



Operations performed:

(i) Find: can be implement by recursively traverse the parent array until we hit a 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 finds the representatives of their sets using the find operation & finally puts either one of the trees (representing the set) under the root node of the other tree, effectively merging the trees & the sets.

Date.
Page No.

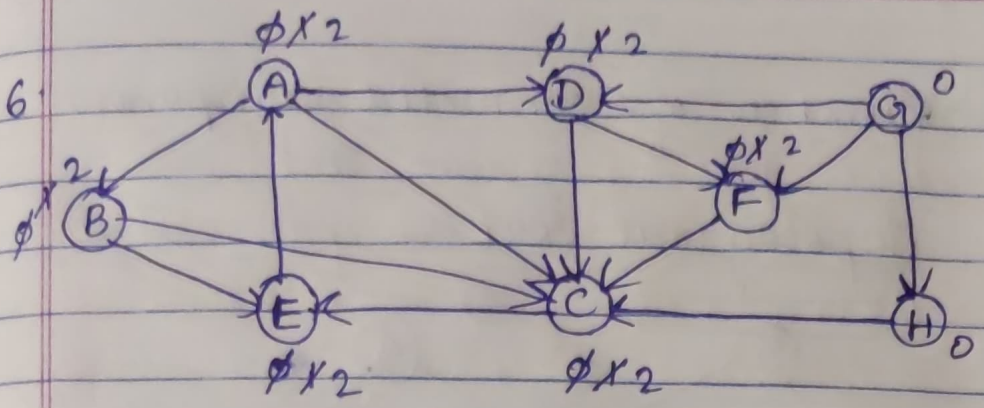
```
void union (int i, int j) {
    int irep = this.Find(i);
    int jrep = this.Find(j);
    this.Parent[irep] = jrep;
}
```

(iii) Path compression (Modifications to find()):- It speeds up the data structure by compressing the height of the tree. It can be acheived 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;
    }
}
```

Date.
Page No.

6.



BFS:- Node B E C A D F
Parent - B B E A D
Unvisited nodes - G and H
Path = B → E → A → D → F

DFS:-
node processed: B B C E A D F
Stack: B CE EE AE DE FE E
Path: B → C → E → A → D → F

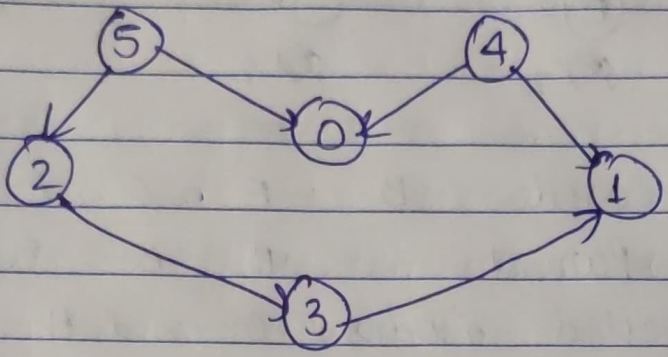
7. V = {a} {b} {c} {d} {e} {f} {g} {h} {i} {j}
E = {a,b}, {a,c}, {b,c}, {b,d}, {e,f}, {e,g}, {h,i}, {j}

(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}

Date.
Page No.

Number of connected components = 3

Q. Topological sort -



Adjacent list 0 →
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)

Step1 - Topological sort (0) visited [0] = true list is empty, no more recursion call

stack [0]

Step2 - Topological sort (1), visited [1] = true list is empty. no more recursion call.

stack [0 | 1]

Date.
Page No.

Step3. Topological sort (2), visited [2] = true

↓

Topological sort (3), visited [3] = true
'1' is already visited. no more recursion call stack

0	1	3	2

Step 4. Topological sort (4), visited [4] = true
'0', '1' are already visited. no more recursion call stack

0	1	3	2	4

Step5. Topological sort (5), visited [5] = true

↓

'2', '0' are already visited. no more recursion call
stack

0	1	3	2	4	5

Step6. Print all elements of stack from top to bottom

5, 4, 2, 3, 1, 0

9. We can use heaps to implement the priority queue. It will take O(log N) time to insert & delete each element in the priority queue. Based on heap structure priority queue also has two types - max priority & min priority queue.

Date.
Page No.

Some algorithms where we need to use priority queue are:

(i) Dijkstra's shortest path algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used extract minimum efficiently when implementing Dijstra's algorithm.

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

(iii) Data compression: It is used in Huffman's code which is used to compresses data.

10.

Min Heap	Max Heap
• In a min-heap the key present at the root must be less than or equal to among the keys present at all of 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.

Date.
Page No.

• The minimum key element present at the root.	• The maximum key element present at the root.
• Uses the ascending priority.	• Uses descending priority.
• In the construction of a min-heap, the smallest element has priority.	• In the construction, the largest element has priority.
• The smaller element is the first to be popped from the heap.	• The largest element is the first to be popped from the heap.