Tutorial-5

Soln:-

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 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.
→ T.C 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.

- 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 & v.

→ we can perform topological storting 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.

• Applications of BFS:

→ Like DFS, BFS may also used for detecting cycles in a graph.

→ Finding shortest path and minimal spanning trees 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 crawlers.
→ In peer-to-peer networking, BFS is to find neighbouring node.
→ In garbage collection BFS is used for copying garbage.

Sol2:- 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 visits an adjacent unvisited node, marks it as done, and inserts it into a queue.

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

Sol3:- • 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 graph is sparse, we should store it as a list of edges. Alternatively, if the graph is dense, we should store it as as adjacency matrix.

Sol4:-
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 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 itself (self-loop) or one of its ancestors in the tree produced by DFS. In the following
→ For a disconnected graph, Get the DFS forest as output
To detect cycle, check for a cycle in individual trees by checking back edges.
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. The edge that

connects the current vertex to the vertex in the recursion stack is a back edge. Use recStack[] array to keep track of vertices in the recursion stack.

• Detect cycle in an undirected graph.

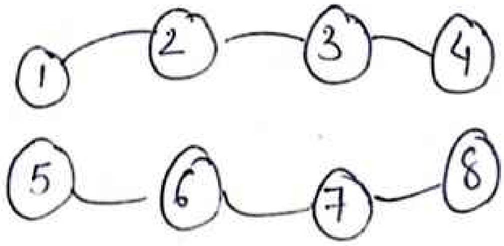
→ Run a DFS from every unvisited node. DFS can be used to detect a cycle in a graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is joining a node to itself (self-loop) or one of its ancestor in 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.

Sol5:- 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 element b/w the two sets.

Eg:- $S1 = \{1, 2, 3, 4\}$
$S2 = \{5, 6, 7, 8\}$



Operations performed :

(i) Find : can be implemented by recursively traversing the parent array until we hit a node who is parent to itself.

~~int find~~ (~~parent~~

```
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, and finally puts either one of the trees (representing the set) under the root node of the other tree, effectively merging the trees & the sets.

```
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 trees. 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;
```

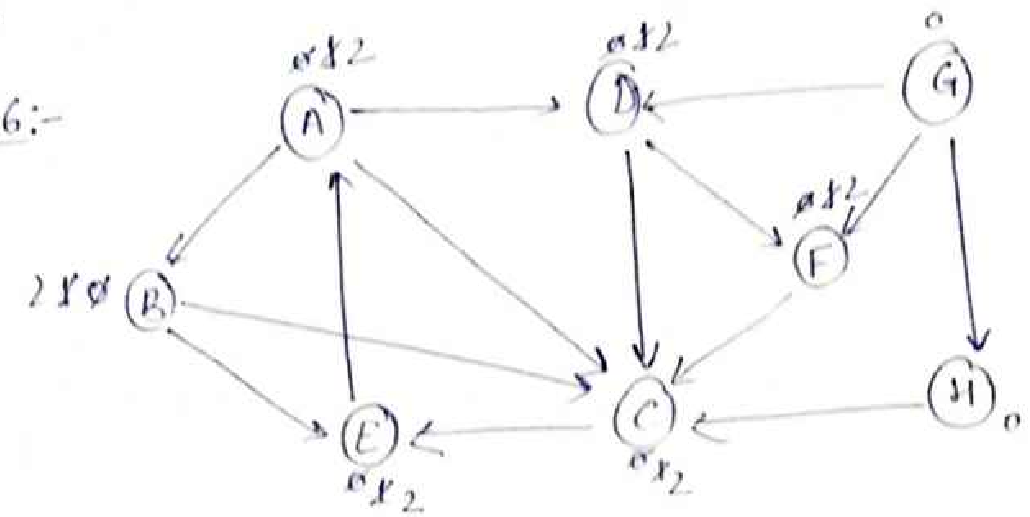
Tut-5

return result;

}

}

Sol 6:-



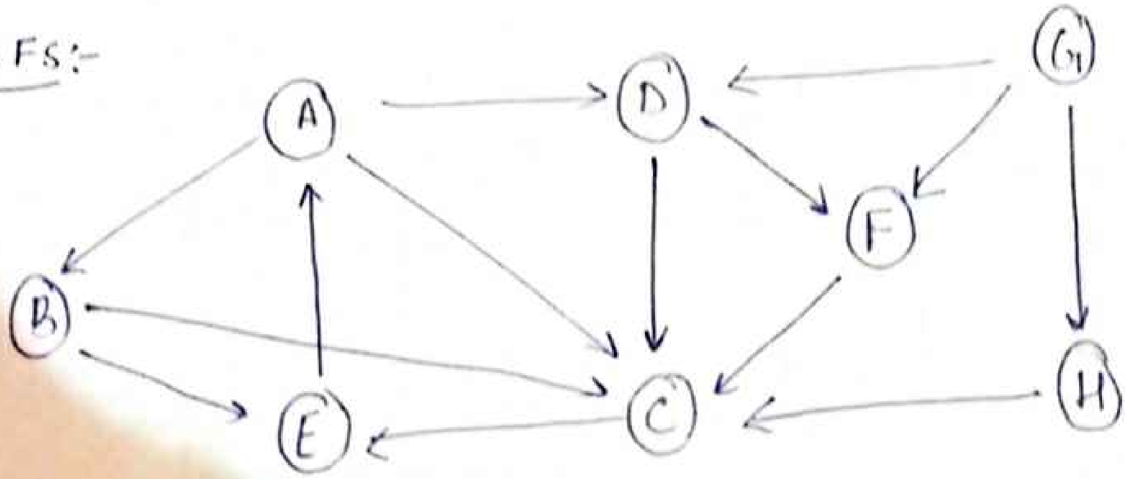
Bfs:- Node (B) (E) (C) (A) (D) (F)

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

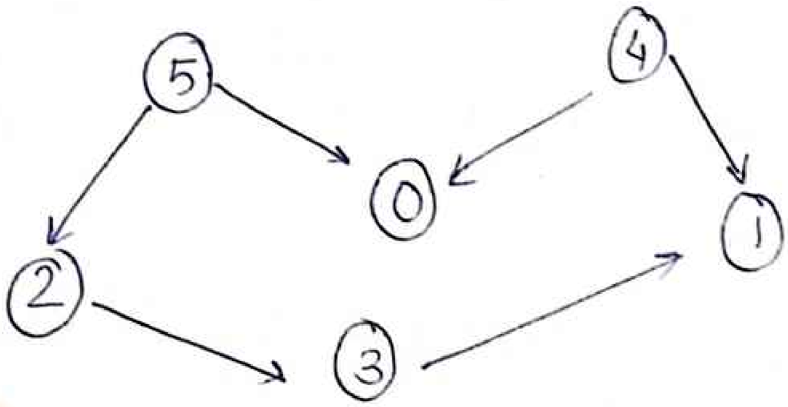
Sol 7 :- V = {a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

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}

Number of connected components = 3 ans.

Sol 8 :- topological sort :-



Adjacent list

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0.

Visited :-

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

stack (empty)

step 1:- Topological sort(0) visited[0] = true
list is empty, No more recursion call

stack $\boxed{0}$

step 2:- Topological sort (1), visited[1] = true
list is empty. No more recursion call.

stack $\boxed{0|1}$

step 3:- Topological sort(2), visited [2] = true
↓
Topological sort(3), visited [3] = true
↓
'1' is already visited. No more recursion call

stack $\boxed{0|1|3|2}$

step 4:- Topological sort (4), visited [4] = true
↓
'0','1' are already visited. No more recursion call

stack $\boxed{0|1|3|2|4}$

step 5:- Topological Sort (5), visited [5] = true
↓
'2','0' are already visited. No more recursion call

stack $\boxed{0|1|3|2|4|5}$.

step 6:- Print all elements of stack from top to bottom

5, 4, 2, 3, 1, 0. ans.

Sol 9:- We can use heaps 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 has also has two types - max priority and min-priority queue.

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 Dijkstra'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.

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