Tutorial 5

(DAA)

Q1)

DFS	BFS
• DFS stands for Depth First Search	→ BFS stands for Breadth First Search
• DFS uses Stack data structure	→ BFS uses Queue data structure for finding the shortest path
• DFS algorithm is a recursive algo. that uses the idea of backtracking.	→ In BFS there is no concept of backtracking.
• DFS is used in various application such as acyclic graph and topological order etc.	→ BFS is used in various application such as bipartite graph, and shortest path etc.
• DFS is faster than BFS and requires less memory.	→ BFS is slower than DFS and requires more memory.

* Application of BFS :-

→ Finding shortest path and minimum spanning tree in unweighted graph.

→ like DFS, BFS can also be used for detecting cycles in a graph.

→ Finding a route through GPS navigation system with minimum of crossing.

→ In networking finding a route for packet transmission

* Applications of DFS :-

→ DFS of a graph produce the minimum spanning tree and pair of all shortest-path trees.

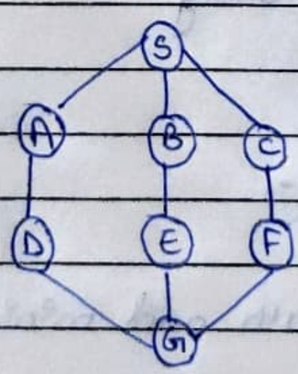
→ Detects a cycle in a graph

→ To find the path between two vertices

→ It is used in Topological sorting and checks if a graph is bipartite or not.

Q2) Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

For example;



In this DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C.

whereas in Breadth First Search (BFS) algorithm we traverse a graph in breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Q3) **Sparse Graph :-** It's the opposite of dense Graph. If a graph has only a few edges (the no. of edges is close to maximum number of edges), then it is a sparse graph. There is no distinction between the sparse and the dense tree.

Dense Graph :- If the number of edges is close to the maximum number of edges in a graph, then that graph is called dense graph.
In dense graph every pair of vertices is connected by one edge.

For sparse graph, adjacency list is good and usually preferred.
And for dense graph, adjacency matrices are the suitable ones as in Big-O terms they don't take up more space.

Q4) To detect cycles in a graph using BFS:-

1) No. of incoming edges for each of the vertex present in graph and initialize the count of visited nodes as 0.
2) Pick all the vertices with in-degree as 0 and add them into a queue (enqueue operation).

3. Remove a vertex from the queue (dequeue operation) and then: increment count of visited nodes by 1, decrease in degree by 1 for all its neighbouring nodes and, if in-degree of a neighbouring nodes is reduced to 0 then add it to the Queue.

4. Repeat steps until Queue is empty.
5. If the count of visited nodes is not equal to the no. of nodes in the graph has cycle, otherwise not.

→ Using DFS :-

1) Create the graph using given number of edges and vertices.

2) Create a recursive function that initializes the current index or vertex, visited and recursion stack.

3) Mark the current node as visited and also mark the index to recursion stack.

4) Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices if the recursive function returns true.

5) If the adjacent vertices are already marked in the recursion stack then return true.

6) Create a wrapper class that calls the recursive function for all the vertices and if any function returns true

, return true else if for all vertices and if any function returns false return false.

Q5) A disjoint - set data structure also called a union find data structure ~~or~~ a merge - find set is a data structure that stores a collection of disjoint sets. It also stores a portion of a set into disjoint subsets.

Operations :-

1) Making new sets - The make set operation adds a new element into a new set containing only the new element and the new set is added to the data structure.

2) Merging ~~two~~ sets - The operation union (x, y) replace the set containing x and set containing y with their union.
Union first uses find to determine the root of the tree containing x & y.

3) Finding set Representatives :- Find operation follows the chain of parent pointer from a specified query ~~node~~ x until it reaches a new element. This root element represents the set to which x belongs and may be x itself. find return the root element it reaches.

Q6) A be source Node and F is destination

⇒ For BFS :-
(queue)

A	Visited
{B, C, D}	{A}
{D, C, E}	{A, B}
{C, E, F}	{A, B, D}
{E, F}	{A, B, D, C}
{F}	{A, B, D, C, E}
	{A, B, D, C, E, F}

⇒ For DFS :-
(stack)

visited	A :	B	C	D	E F
Stack	~~B~~	E,	F	~~E~~	F
	~~D~~	~~D~~	~~C~~	F	
	~~C~~	~~C~~			

⇒ {A, B, D, C, E, F}

Q7) In Disjoint set union algorithm, there are two main functions i.e., connect() and root() function

connect() connects an edge and Root() recursively determine the topmost parent of a given edge.

For each edge {a, b}, check if 'a' is connected to 'b' or not, if found to be false connect them by appending their top parents. After completing the above step for every edge, print total no. of distinct top-most parents for each Vertex. as in this example the output will be = 3.

Q8) Class Graph

```
{
  int v;
  list <int>* adj;
  void topologicalsortutil (int v, bool visited[], stack
            <int> & stack);

public:
  Graph (int v);
  void add Edge (int v, int w);
      void topological sort ();
};
Graph :: Graph (int v)
{
    this -> V = v;
    adj = new list <int> [v];
}
```

```
void Graph :: addEdge (int v, int w)
{
    adj [v]. push_back (w);
}
void Graph :: topological sort util (int v, bool visited [],
                                      stack <int > & stack)
{
  visited [v] = true;
  list <int > :: iterator i;
  for (i = adj [v]. begin ();
       i != adj [v]. end (); ++i)

    if (!visited [*i]
       topological sort util (*i, visited, stack);
          stack.push (v);
     }
   void Graph :: topological sort ()
    {
      Stack <int > Stack;
          bool * visited = new bool [v];
          for (int i = 0; i < v; i++)
            Visited [i] = false;
        for (int i = 0; i < v; i++)
          if (visited [i] == false)
        topological sort util (i, visited, stack);
           while (stack. empty () == false)
           {
            cout << stack. top () << " ";
                 Stack. pop ();
           }
     }
```

Q9) Yes, heap datastructure can be used to implement priority queue.
Heap datastructure provides an efficient implementation of priority queue.

→ Few graph algorithms where priority Queue is used are:-

1) Dijkstra's algo → when graph is stored in the adjacency matrix a list, priority Queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.

2) Prims Algo → to store keys of node and extract minimum key node at every step.

3) A* search algo → It is used to find the shortest path b/w two vertices of a weighted graph

Q10)

	Min Heap	Max Heap
1)	In min heap the key present at root node must be less than or equal to among the keys present at all of its children.	In max heap the key present at root node must be greater than or equal to among the keys present at all of children.
2)	The min. element is present at the root	The maximum element is present at root.
3)	It uses the ascending priority	It uses descending priority

4)	In construction of min heap the smallest element has priority.	In the construction of max heap the largest element has priority.
5)	The smallest element is the first to be popped from the heap.	The largest element is the first to be popped from the heap.