TUTORIAL- 05

Name- Harshita Mishra
Section- CST-SPL-1
University Roll no- 2017498

Ques 1 :→ What is difference between DFS and BFS. Please write the algorithm of both the algorithms ?

Ans :→

DFS	BFS
① It stands for Depth First Search.	① It stands for Breadth First Search.
② DFS uses stack to find the Shortest Path.	② BFS uses Queue to find the Shortest Path.
③ It finds the shortest path/ Possible to the destination.	③ It finds all the ~~possible~~ shortest path to the destination.
④ It is implemented using LIFO List.	④ It is implemented using FIFO List.
⑤ Backtracking is used in DFS.	⑤ No Backtracking is required.

Their Applications :→

① **BFS :→**
- ① It is used for detecting cycle in a graph.
- ② Finding a route to GPS navigation System with minimum no. of crossing.
- ③ In networking finding a route for packet transmission.

② **DFS :→**
- ① For detecting cycles in a graph.
- ② It is used in the Generation of Topological sorting.
- ③ We can find Strongly connected components of graph.

Ques 2. :→ Which data Structure are used to implement BFS & DFS and why ?

Ans → In DFS, Stack data Structure is used means implemented as LIFO List. Whereas, In BFS, Queue data Structure is used means implemented as FIFO List.

DFS algorithm transverse 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.

BFS algorithm transverse a graph in a breadthwise motion and uses a queue to get the next search as it is based on First in First out basis.

Ques 3 :- What do you mean by sparse and dense graph? which representation of graph is better for sparse and dense graph?

Ans :- Sparse graph is defined as a graph in which the number of edges is much less than the possible number of edges.

Whereas, 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 can as an adjency matrix.

Ques 4 :- How can you detect a cycle in a graph using BFS and DFS?

Ans :- For BFS :-

1. No. of incomming edges for each of vertex present in graph and initialize the count of visited nodes as 0.
2. Pick all the vertices within degree as 0 and add them into a queue.
3. Remove a vertex from queue.
4. Repeat Step 3 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.

```
Class Graph {
        int V;
        list <int> *adj;
    Public:
        Graph (int V);
        void addEdge (int u, int v);
        bool isCycle ();
};
```

```
Graph :: Graph (int v)
{
  this → v = v;
  adj = new List<int>[v]; }

Void  graph:: add Edge (int u, int v) {
        adj[u]. push_back (v); }
```

For DFS :→ ① Create graph using the given no. of edge & vertices.

② Create a recursive fn that initializes the current index of vertex, visited & recursion stack.

③ Make the current ~~stack~~ node as visited & make index in recursion stack.

④ Find all the vertices which are not visited and are adjacent to the current node.

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

⑥ Create a wrapper class, that calls the recursive fn for all the vertices and if any fn returns true, else if return false.

```
Class Graph {
        int V;
        List<int>* adj;
        bool is cyclic util (int v, bool visited[], bool* rs);
      Public:
        Graph (int V);
        void add edge (int v, int w);
        bool is cyclic(); };

Graph:: Graph (int V) {
        this → V = V;
        adj = new list< int> [v];
    }

void  graph :: add edge (int v, int w) {
        adj [v]. Push_back (w); }

bool Graph:: is cyclic util (int v, bool visited[], bool* restack) {
        if (visited [v] == false) {
            visited [v] = true;
            restack [v] = true;
            List <int> :: iterator i;
        for (i = adj[v]. begin (); i != adj [v]. end (); ++i) {
            if (!visited [*i] && is cyclic util (*i, visited, restack);
            return true;
        else if (restack [*i])  return true; }}
```

```
recstack[v] = false;
return false; }

bool graph :: isCyclic() {
    bool *visited = new bool[v];
    bool *recstack = new bool[v];
    for (int i=0; i<v; i++) {
        visited[i] = false;
        recstack[i] = false; }
    for (int i=0; i<v; i++)
        if (!visited[i] && isCyclicUtil(i, visited, recstack))
        return true;
    return false; }
```

Ques 5 :- What do you mean by disjoint set data structure? Explain 3 operation along with examples, which can be performed on disjoint sets?

Ans :- Disjoint set is defined as the subsets where there is no common element between the two sets.

$S_1 = \{1, 2, 3, 4\}$
$S_2 = \{5, 6, 7, 8\}$

Let, $S_1 \cup S_2 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

$S_3 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

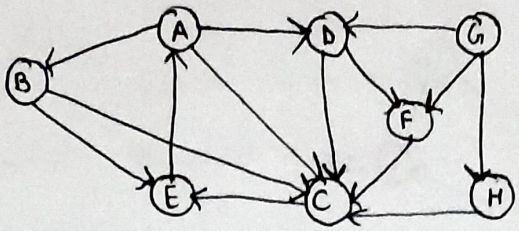
3 Is operation :-

① **Making new sets :-** The (Make Set) operation adds a new element into newest containing only the new element, & new set is added to data structure.

② **Merging Two Sets :-** The operation (Union(x,y)) replace the set containing x & set y with their union.

③ **Finding Set Representation :-** The (find) operation follows the chain of parent pointers from a specified query node x until it reaches a root element. This root element represents the set to which x belongs & may be x itself. (Find) returns the root element it reaches.

Ques 6 :→ Run BFS & DFS on graph shown on right side.



Ans :→ Let source node → A & and goal node → F

① **For BFS →**

A	
{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}

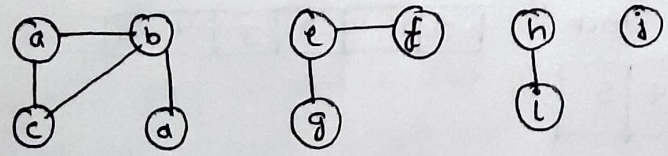
⇒ {A, B, D, C, E, F}

② **For DFS →**

Visited	A	B	D	C	E	F
Stack	~~A~~	E	F	~~E~~	F	
	~~D~~	~~B~~	~~C~~	F		
	~~C~~	~~C~~	~~C~~			

⇒ {A, B, D, C, E, F}

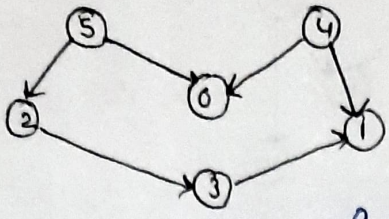
Ques 7 → Find out the number of connected components and vertices in each component using disjoint set data structure.



Ans →

Initial sets	Collection of disjoint sets								
	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b, d)	{a}	{b, d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e, g)	{a}	{b, d}	{c}		{e, g}	{f}		{h}	{i}	{j}
(a, c)	{a, c}	{b, d}			{e, g}	{f}		{h}	{i}	{j}
(h, i)	{a, c}	{b, d}			{e, g}	{f}		{h, i}		{j}
(a, b)	{a, b, c, d}				{e, g}	{f}		{h, i}		{j}
(e, f)	{a, b, c, d}				{e, f, g}			{h, i}		{j}
(b, c)	{a, b, c, d}				{e, f, g}			{h, i}		{j}

Ques Q:→ Apply Topological Sorting & DFS on graph having vertices from 0 to 5?



Ans→

0→
1→
2→ 3
3 → 1
4 → 0, 1
5 → 2, 0

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

Stack (empty)

① Topological Sort (0) visited [0] = true

Stack | 0 |

② Topological Sort (1) visited [1] = true

Stack | 0 | 1 |

③ Topological Sort (2) visited [2] = true
↓
Topological Sort (3) visited [3] = true

Stack | 0 | 1 | 2 | 3 |

④ Topological Sort (4) visited [4] = true and (0 & 1) already visited

Stack | 0 | 1 | 2 | 3 | 4 |

⑤ Topological Sort (5) visited [5] = true and (2, 0) already visited

Stack | 0 | 1 | 2 | 3 | 4 | 5 |

⇒ | 0 | 1 | 2 | 3 | 4 | 5 |

DFS can be '5 4 3 2 1 0' but it is not a topological sort.

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices.

Ques 9:→ Heap data structure can be used to implement priority queue? Name few graph algorithms where you need to use priority queue and why?

Ans:→ Heap data structure can be used to implement priority queue.

as a priority queue is different from normal queue because instead of FIFO, values are come out in order of priority. It is an abstract data type.

Few algorithms are:→

① Dijkstra's Algorithm → 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.

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

Ques 10→ what is the difference b/w Min and Max Heap?

Ans→

Min Heap	Max-Heap
① In this Key present at the root node must be less than or equal to ~~par~~ keys present.	① In this key present at root node must be greater than or equal to keys present.
② In this minimum keys are present at root node.	② In this maximum keys are present at root node.
③ It uses ascending priority.	③ It uses the descending priority.
④ In this Smallest element has priority.	④ In this Largest element has priority.
⑤ In this Smallest element is first to be popped from Heap.	⑤ In this Largest element is first to be popped from Heap.