Name → Muskan Agarwal DAA Tutorial - 5

Section → F

Roll no. → 15

Q1. What is the difference between DFS and BFS. Write the applications of both algorithms.

Ans-

BFS	DFS
(i) BFS stands for Breadth First Search.	(i) DFS stands for Depth First search.
(ii) BFS uses Queue data structure for finding the shortest path.	(ii) DFS uses Stack data structure for finding one of the possible path.
(iii) BFS can be used to find single source shortest path in an unweighted graph, we reach a vertex with minimum no of edges from a source vertex.	(iii) In DFS, we might traverse through more edges to reach a destination vertex from a source.
(iv) BFS is more suitable for searching vertices which are closer to the given source.	(iv) DFS is more suitable when there are solutions away from source.

BFS Applications :-

(i) BFS is based on path finding algorithm.

(ii) Used in Ford-fulkeson algorithm to find maximum flow in a network.

(iii) Using GPS navigation system BFS is used to find neighbouring places.

DFS Applications :-

(i) Using DFS we can find path between two given vertices u and v.

(ii) Topological sorting is used to scheduling jobs from given dependencies among jobs.

Q2. Which Data structure are used to implement BFS and DFS and why?

Ans- BFS uses queue data structure to traverse a graph in a breadth forward motion and uses a queue to remember to set the next vertex to start and search, when a end occur in any The queue follow the queue concept that grant one discounted pass will be explained.

DFS uses the stack data structure to traverse the graph to depth the motion and uses stack to remember to go to the next level to search, when a dead end occur in any iteration.

Q3 What doyou mean by sparse and dense graphs? Which representation of graph is better for sparse and dense graphs?

Ans- Sparse → Sparse graph is a graph in which the number of edges is close to the minimal no. of edges. Sparse graph can be disconnected graph.

Dense → Dense graph is a graph in which the number of edges is close to the maximum no. of edges

For Sparse graph : Adajacency list representation of graph is better and it is generally prefered.

For Dense graph: For $O(t) = O(V^2)$ and so adjacency matrices are a good representation strategy because in big-O terms they don't take up more space than storing all the edges in a linked list and operations are much faster.

Q4- How can you detect a cycle in a graph using BFS and DFS?

Ans- Steps involved to detect a cycle in graph using BFS

Step-1: Compute in degree (no. of edges) for each vertex present in graph and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)

Step-3: Remove a vertex from queue & then

(i) Increment count by 1.

(ii) Decrease in-degree by 1 for all its neighbouring nodes.

(iii) If in-degree of a neighbouring nodes is reduced to zero, then add it to the queue.

Step-4: Repeat step 3 until the queue is empty.

Step-5: If count of visited nodes is not equal to the no. of nodes in the graph has cycle, otherwise not.

Detecting cycle in graph using DFS we need to do following:

DFS for a connected graph produces a tree. There is cycle in graph for a connected graph produces a tree if there is a back edge in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by DFS. For a disconnected graph, get DFS forest as output. To detect cycle, check for a cycle in individual tree by checking back edges. To detect a back edge, keep track of vertices currently in recursion track for DFS traversal. If a vertex is reached that is already in recursion stack, then there is a cycle.

Qs. What do you mean by disjoint set data structure? Explain 3 operations along with examples which can be performed an disjoint sets.

Ans- A disjoint set is a data structure that keeps track of set of elements partioned into several disjoint subsets. In other words, a disjoint set is a group of sets where no item can be in more than one set.

3 operations:

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

For ex →

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

(ii) Union by Rank → We need a new array rank[] size of array sameas parent array. It is representative of set rank[i] is height of tree. We need to minimize height of tree. If we are uniting 2 trees, we call them left and right, then it all depends on rank of left and right.

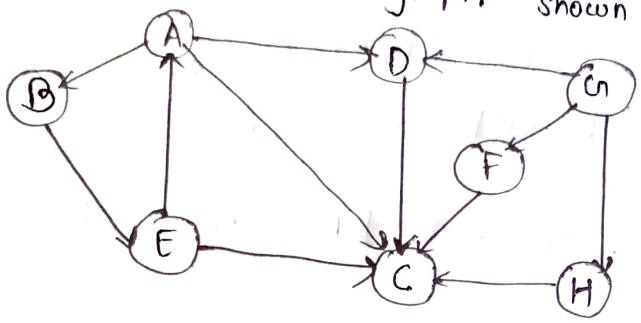
- If rank of left is less than right then its best to move left under right & vice versa.
- If ranks are equal, rank of result will always be one greater than rank of trees.

For ex →

```
void union (int i, int j){
  int irep = this.Find (i);
  int jrep = this.Find (j);
  if (irep == jrep)
  return;
  irank = Rank[irep];
  jrank = Rank[jrep];
  if (irank < jrank)
  this.Parent[irep] = jrep
```

```
else if (jrank < irank)
this.parent[jrep] = irep;
else {
    this.parent[irep] = jrep;
    Rank[jrep]++;
}
}
```

Q6- Run BFS & DFS on graph shown below:



BFS

Child	G	H	D	F	C	E	A	B
Parent		G	G	G	H	C	E	A

Path → G → H → C → E → A → B

DFS

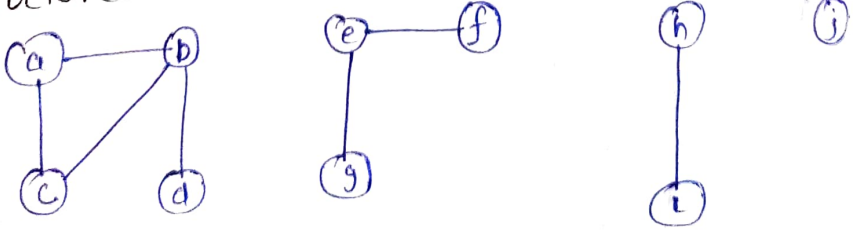
~~G~~ D H ~~F~~ C ~~E~~ ~~A~~ ~~B~~ — Nodes visited

G F C E A B — Stack

Path → G → F → C → E → A → B

Q7. Find out no. of connected components and vertices in each component using disjoint set data structure.

Sol-



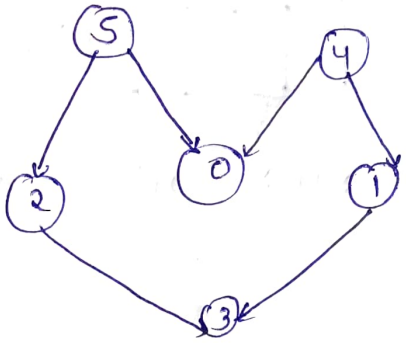
V = {a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

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} {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}

No. of connected components = 3 → true

Q8. Apply topological sort & DFS on graph having vertices from 0 to 5.



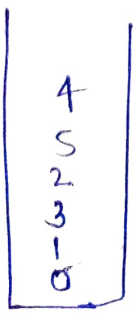
We take source code as 5.

Apply T.S.

DFS(5)
↓
DFS(0)
DFS(2)
DFS(3)
DFS(1)

DFS(4)
Not possible.

DFS



stack

4 → 5 → 2 → 3 → 1 → 0

Ans.

Q9. Heap data structure can be used to implement priority queue. Name few graph algorithm where you need to use priority queue and why?

Ans – Yes, heap data structure can be used to implement priority queue. It will take O(log N) time to insert and delete each element in priority queue. Based on heap structure, priority queue has two types max-priority queue based on max heap & min priority queue based on min heap Heaps provide better performance comparison to array & other structures.

The graphs like Dijkatra's shortest path algorithm, Prim's Minimum spanning tree use Priority Queue.

- Dijkatra's Algorithm → When graph is stored in form of adajancency list or matrix, priority queue is used to extract minimum efficiently when implementing the algorithm.
- Prim's Algorithm → It is used to store keys of nodes & extract minimum key node at every step.

Q10. Differentiate between Min-heap & Max-heap.

Ans

Min Heap	Max Heap
(i) In Min Heap, key present at root node must be less than or equal to among keys present at all of its children.	(i) In Max-heap the key present at root node must be greater than or equal to among keys present at all of its children.
(ii) It uses ascending priority.	(ii) It uses descending priority
(iii) The smallest element has priority while construction of min heap.	(iii) The largest element has priority while construction of max heap.
(iv) The smallest element is the first to be popped from the heap.	(iv) The largest element is the first to be popped from the heap.