

## TUTORIAL - 5.

Name - Samaidhi

Section - P

Roll no. - 24

University Roll no. - 2016985.

Que 1. What is the difference between BFS and DFS.  
Write applications of both the algorithms.

Ans

BFS

DFS.

- It stands for breadth first search.

- It uses queue data structure.

- It is more suitable for searching vertices which are close to give source.

- BFS considers all neighbours first & therefore not suitable for decision making trees used in games & puzzles.

- Here siblings are visited before childrens.

- There is no concept of backtracking.

- It requires more memory.

- It stands for depth first search.

- It uses stack data structure.

- It is more suitable when there are solutions away from source.

- DFS is more suitable for game or puzzle problems.

- We make a decision, then explore all paths through this decision. And if decision leads to win situations, we stop.

- Here children are visited before siblings.

- There is a recursive algorithm in that uses backtracking.

- It requires less memory.

### Applications:

- BFS: Bipartite graph and shortest path, peer to peer networking crawlers in search engine & GPS navigation system.
- DFS: Acyclic graph, topological order, scheduling problems, sudoku puzzles.

Que >

Which data structure is used to implement BFS and DFS and why?

Ans

For implementing BFS we need a queue data structure for finding shortest path between any node. We use queue because things don't have to be processed immediately, but have to be processed in FIFO order like BFS. BFS searches for nodes do not miss, i.e., it searches nodes w.r.t their distance from root (source). For this queue is better to use in BFS.

For implementing DFS we need a stack data structure as it traverses a graph in depthward manner & uses stack to remember to get the next vertex to start a search, when a dead end occurs in any situation.

Que 3 >

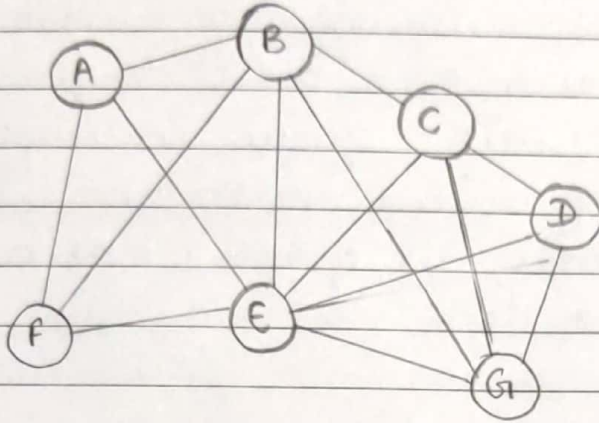
What do you mean by sparse & dense graph? Which representation of graph is better for sparse & dense graph?

Ans

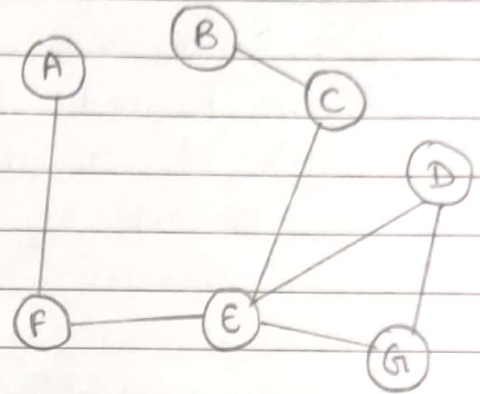
Dense graph is a graph in which no. of edges is close to maximal no. of edges.



Sparse graph is graph in which no. of edges is very less.



Dense graph (many edges b/w nodes)



Sparse graph (few edges b/w nodes)

- For sparse graph it is preferred to use adjacency list.
- For dense graph it is preferred to use adjacency matrix.

Que 4) How can you detect a cycle in a graph using BFS and DFS?

Ans For detecting cycle in a graph using BFS we need to use Kahn's algorithm for topological sorting -

The steps involved are:

- 1) Compute in-degree (no. of incoming edges) for each of vertex present in graph & initialize count of visited nodes as 0.
- 2) Pick all vertices with in-degree as 0 & add them in queue.
- 3) Remove a vertex from queue and then

- increment count of visited nodes by 1.
  - decrease in degree by 1 for all its neighbouring nodes.
  - If degree of neighbouring nodes is reduced to zero then add to queue.
- 4) Repeat ③ until queue is empty.
- 5) If count of visited nodes is not equal to no. of nodes in graph, has cycle, otherwise not.

for detecting cycle in graph using DFS we need to do following.

- DFS for a connected graph produces a tree.
- There is cycle in graph if there is a back edge present 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 trees by checking back edges. To detect a back edge, keep track of vertices currently in recursion stack for DFS traversal. If a vertex is reached that is already in recursion stack, then there is a cycle.

Ques 5) What do you mean by disjoint set data structure? Explain 3 operations along with examples which can be performed on disjoint sets?



Ans

A disjoint set is a data structure that keeps track of set of elements partitioned 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:

\* Find: can be implemented recursively traversing the parent array until we hit a node who is parent to itself.

Ex:

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

\* Union: It takes 2 elements as input. And find representation of their sets using the find operation & finally puts either one of the trees under root node of other tree, effectively merging the trees & sets.

Ex:

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

\* Union By Rank: We need a new array `rank[]`. Size of array same as parent array. If  $i$  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 & right.

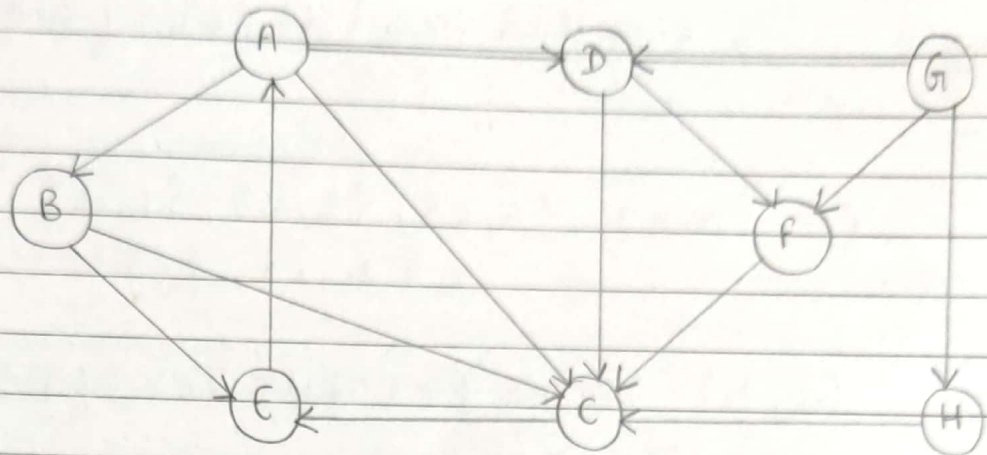
- If rank of left is less than right then it's best to move left under right & vice versa.
- If rank are equal, rank of result will always be one greater than rank of trees.

Ex:

```
void union (int i, int j) {
    int iup = this.find(i);
    int jup = this.find(j);
    if (iup == jup) return;
    irank = rank[iup];
    jrank = rank[jup];
    if (irank < jrank)
        this.parent[iup] = jup;
    else if (jrank < irank)
        this.parent[jup] = iup;
    else {
        this.parent[iup] = jup;
        rank[jup]++;
    }
}
```

Que 6) Run BFS & DFS on graph shown below.





BFS

child	G	H	D	F	C	E	A	B
Parent.	G	G	G	G	H	C	E	A

Path  $G \rightarrow H \rightarrow C \rightarrow E \rightarrow A \rightarrow B$ .

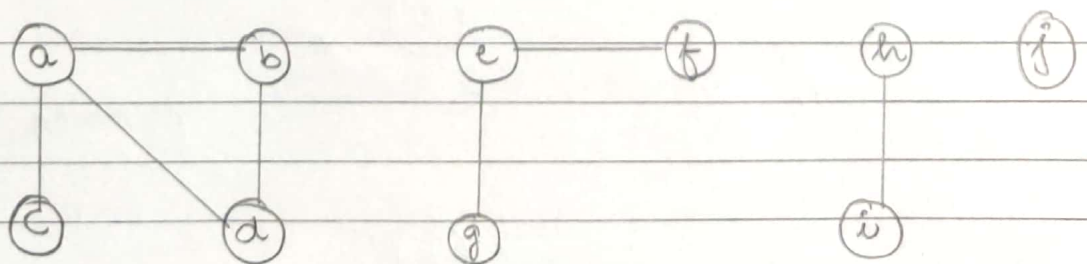
DFS

<del>G</del> <del>D</del> <del>H</del> <del>F</del> <del>E</del> <del>A</del> <del>B</del>	Nodes visited	<del>G</del> <del>F</del> <del>C</del> <del>E</del> <del>A</del> <del>B</del>	stack
--	------------------	--	-------

Path  $\rightarrow G \rightarrow F \rightarrow C \rightarrow E \rightarrow A \rightarrow B$ .

Que 7)

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



Ans

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

$E = \{a,b\}, \{a,c\}, \{b,c\}, \{b,d\}, \{c,f\}, \{c,g\}$   
 $\{h,i\}, \{j\}$ .

$(a,b) \quad \{a,b\} \{c\} \{d\} \{e\} \{f\} \{g\} \{h\} \{i\}$   
 $\{j\}$

$(a,c) \quad \{a,b,c\} \{d\} \{e\} \{f\} \{g\} \{h\} \{i\} \{j\}$

$(b,c) \quad \{a,b,c\} \{d\} \{e\} \{f\} \{g\} \{h\} \{i\} \{j\}$

$(b,d) \quad \{a,b,c,d\} \{e\} \{f\} \{g\} \{h\} \{i\} \{j\}$

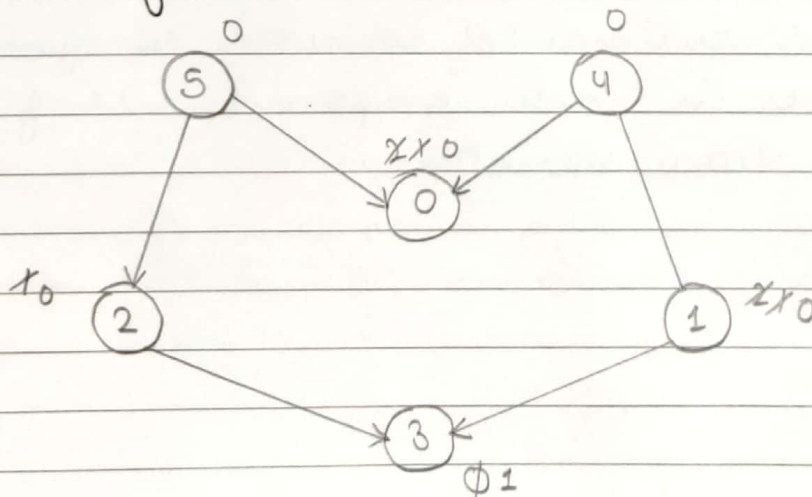
$(c,f) \quad \{a,b,c,d\} \{e,f\} \{g\} \{h\} \{i\} \{j\}$

$(c,g) \quad \{a,b,c,d\} \{e,f,g\} \{h\} \{i\} \{j\}$

$(h,i) \quad \{a,b,c,d\} \{e,f,g\} \{h,i\} \{j\}$

No. of Connected components = 3  $\rightarrow$  Ans

Que 8 Apply topological sort & DFS on graph having vertices from 0 to 5.



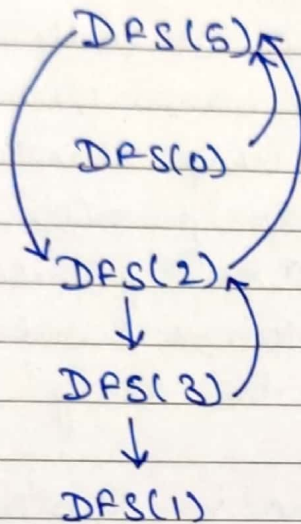


Ans

We take source node as 5

Applying Topological Sort

q: 5/4 : Pop 5 & decrement  
-ent indegree by 1.



q: 4/2 ; Pop 4 & decrement  
-ent & push 0.

DFS(4)

↓

Not possible

q: 2/0; Pop 2 & decrement  
-ent indegree & push 3

q: 0/3; Pop 0, Pop 3,  
Push 1.

q: 1; Pop 1.

Ans: 5 4 2 0 3 1

Topological sort.

DFS

4

5

2

3

1

0

stack

4 → 5 → 2 → 3 → 1 → 0 Ans

Que 9)

Heap data structure can be used to implement priority queue. Name two graph algorithms where you need to use priority queue & why?

Ans

Yes, heap data structure can be used to implement priority queue. It will take  $O(\log V)$  time to insert & 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 & LL.

The graph like Dijkstra's shortest path algorithm, Prim's minimum spanning tree use Priority Queue.

- > Dijkstra's algorithm: When graph is stored in form of adjacency 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 and extract minimum key nodes at every step.

Que 10) Differentiate b/w min-heap & max-heap

Min heap

→ In min-heap, key present at root node must be less than or equal to among keys present at all of its children.

Max-heap.

→ In max-heap, key present at root node must be greater than or equal to among keys present at all of its children.



→ The minimum key element is present at the root.

→ It uses ascending priority

→ The smallest element has priority while construction of Min-heap.

→ The smallest element is the first to be popped from the heap.

→ The maximum key element is present at the root.

→ It uses descending priority

→ The largest element has priority, while construction of Max-heap.

→ The largest element is the first to be popped from the heap.