# TUTORIAL - 5

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

Ans

| BFS | DFS |
| --- | --- |
| It stands for Breadth First Search | It stands for Depth First Search. |
| It uses Queue data structure | It uses stack data structure |
| It is more suitable for searching vertices which are closer to given source. | It is more suitable when there are solutions away from source. |
| ~~Time Complexity of BFS is~~ | |
| BFS considers all neighbours first & therefore not suitable for decision making trees used in games & puzzles. | 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 situation, we stop. |
| Here siblings are visited before children | Here children are visited before siblings. |
| There is no concept of backtracking. | It is a recursive algorithm that uses backtracking. |
| It requires more memory. | 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 puzzle.

Q2) Which data structure are used to implement BFS and DFS and why?

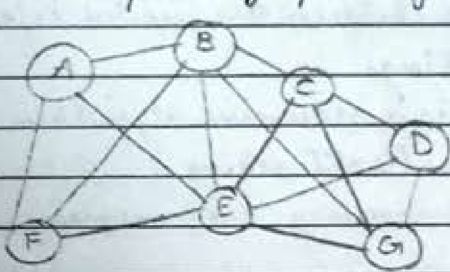
↳ For implementing BFS we need a queue data structure for finding shortest path between any node. We use queue because things don't have to be processed immediately, but have to be processed in FIFO order like BFS. BFS searches for nodes level wise, i.e. it searches nodes w.r.t their distance from root (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 motion and uses stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

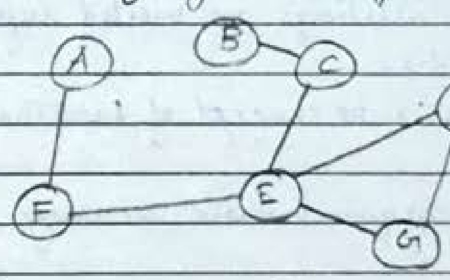
Q3) What do you mean by sparse and dense graphs? Which representation of graph is better for sparse and dense graph?

↳ 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 graphs (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.

Q4) 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 and 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 in-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.

Q5) 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 by recursively traversing the parent array until we hit a node who is parent to itself.

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

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

- **Union** → It takes 2 elements as input. And find representatives of their sets using the find operation and finally puts either one of the trees under root node of other tree, effectively merging the trees and sets.

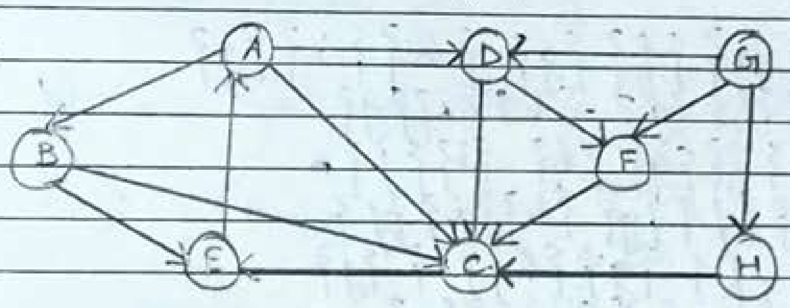
eg:
```
void union (int i, int j) {
    int irep = this.Find (i);
    int jrep = this.Find (j);
    this.parent [irep] = jrep;
}
```

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

eg:-
```
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 and DFS on graph shown below.

↳ 


BFS ~~Child~~

| 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

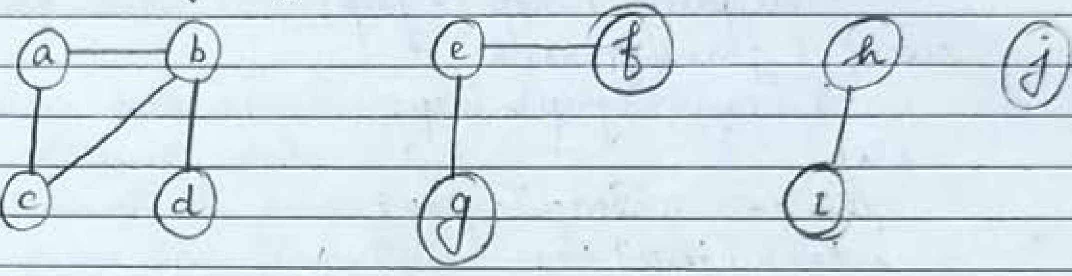
| NODES VISITED | STACK |
|---|---|
| ~~G~~ | G |
| D | F |
| H | C |
| ~~F~~ | E |
| ~~C~~ | A |
| ~~E~~ | B |
| ~~A~~ | |
| ~~B~~ | |

~~PATH~~ STACK

Path → G ~~E~~ → F → C → E → A → B

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

Ques.



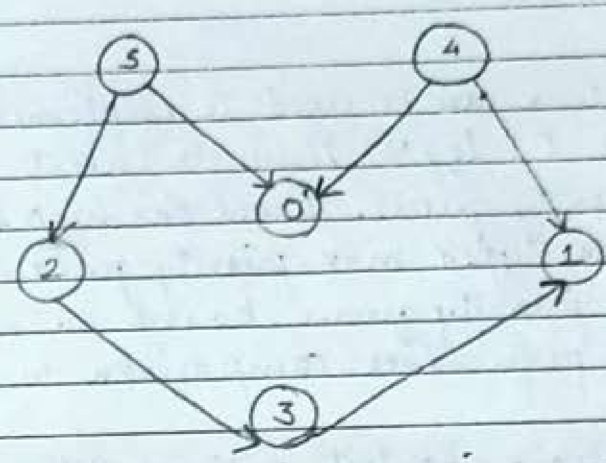

Ans $V = \{a\}\ \{b\}\ \{c\}\ \{d\}\ \{e\}\ \{f\}\ \{g\}\ \{h\}\ \{i\}\ \{j\}$

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

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

No. of connected components $= \underline{3}$ → Ans

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




Ans. We take source node as 5.

Applying Topological Sort

DFS (5)
↓
DFS (0)

DFS (2)
↓
DFS (3)
↓
DFS (1)

DFS (4)
↓
Not possible

DFS

| |
|---|
| 4 |
| 5 |
| 2 |
| 3 |
| 1 |
| 0 |

Stack

$4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0$

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 and min priority queue based on min-heap. Heaps provide better performance comparison to array & list.

The graphs 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 node at every step.

Q10) Differentiate between Min-heap and Max-heap.

↳

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