Day 38 : Graphs.

BFS, DFS → most asked in the interview.

↓

collection of nodes and a collection of edges, where an edge connects two nodes, which could be bidirectional or unidirectional.

(can travel from a→b, b→a)

(only in one direction)

↓

vertices represented by V

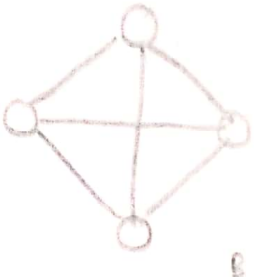
edges represented by E

↓

tree is a type of graph which has only one path b/w two nodes (special kind of graph).

↓

clique (all nodes are connected to each other node)

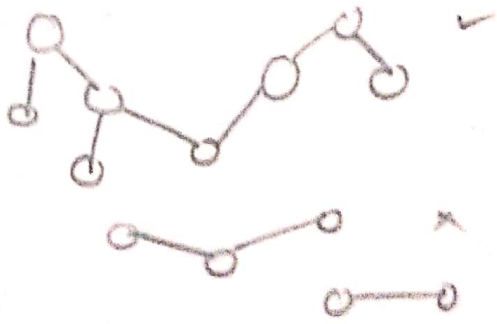


↓

employed in facebook (people can be thought of as nodes, and friendship, relations can be considered edges)

also a graph

Q. Given a bunch of nodes, tell whether they are connected or not?



if it were a tree, then we call on all children, sum them and add 1 to it. If they are equal to the total number of nodes, then connected otherwise of not.

DFS : Depth First Search → keep moving forward till neighbours are explored and then comes back.

The tree approach can't be straight away applied to a graph, because that may lead to an infinite loop.

↓

In case of graphs, where cycles are present, we maintain a visited array. If the node is already visited, then we don't compute it and return else we operate on it & mark it.

↓

visited can be marked before calling or after computation

↑

visited state can be maintained using a map. Or using an array (when nodes are numbered)

(visited <node, bool>)

↓

depending on the implementation

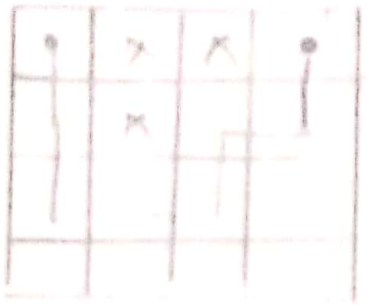
if count returned is equal to total nodes, then connected graph.

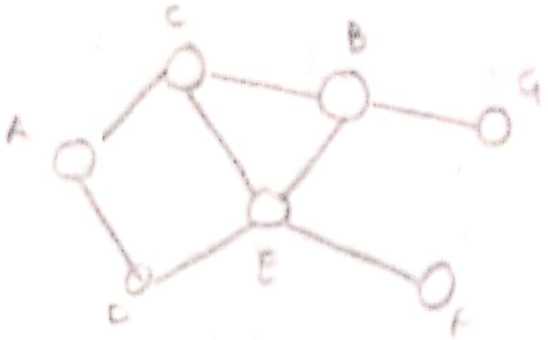
```
int dfs(root){
    if (root == NULL) return;
    int cnt = 1;
    for (int i=0; i<root.neighbors.size(); i++)
    {
        if (visited[root.neighbours[i]])
            continue;
        visited[root.neighbours[i]] = true;
        cnt += dfs(root.neighbours[i]);
    }
    return cnt;
}
```

maintaining the count of nodes.

visited nodes are not repeated

Q.

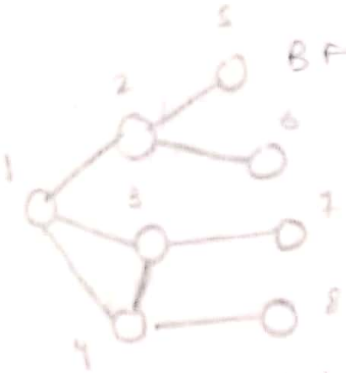




Given a graph with bidirectional edges with same weight find the shortest path between A and B.

**

(next variation : grid)



4 to 3 won't happen since it is already visited and also its shortest distance is 1

BFS Breadth first search

↓

level order traversal in a graph

↓

visited neighbours at distance 1 first, then their neighbours (at distance 2 from B) and so on till A is encountered.

↓

implemented using queue.

and a visited array *

↓

for each node at top of queue, unvisited neighbours are pushed to the queue, with dis + 1.

↓

root or starting vertex is pushed initially with dis = 0.

```
queue < pair <node, int >> q ;
q.push (make-pair (start, 0));
visited [start] = true ;
while ( ! q.empty() ) {
    pair <node, int > top = q.front ();
    q.pop();
    int current-dis = top.second ;
    node current-node = top.first ;
    for (int i = 0 ; i < current-node.neighbours.
                                    size(); i++)
    {
        if (visited [current-node.neighbours[i]])
            continue ;
        else {
            visited [current-node.neighbours[i]];
            q.push (make-pair (current-node.neighbour[i],
                               dist+1));
        }
    }
}
return -1;
```

if (current node == des) return current-dis;

→ if the destination node is not reachable

* connected graph check can be done using a BFS as well.

* BFS won't work on weighted graphs. → Dijkstra's algorithm

when the have various weights → they guarantee shortest path

nodes on same levels are at different distances or have different costs.

→ processing them equally is not correct

→ they need to be processed differently.

* time complexity : $O(V+E)$ → visiting each of its neighbours.
BFS (implementation based) — V: visiting each node

* space complexity : $O(V)$ → visited array (V) and queue (V).
BFS

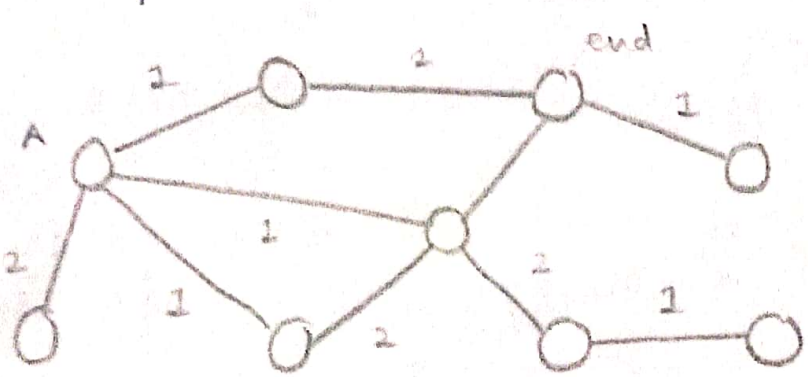
$O(\max(A,B)) \leq O(A+B)$ (in general)

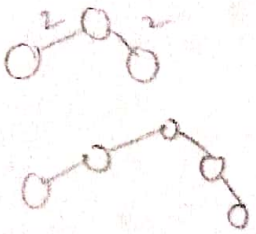
dense → $O(E)$
sparse → $O(V)$
} (equivalent in Big O notation)

* time complexity
DFS : $O(V+E)$

* space complexity DFS : $O(V)$. (max depth it can reach)
stack memory → V nodes at max
visited array → V sized.

Q. shortest path question for a modified graph





(variation → extended to k?)
in place of 2

Approach 1 : (2 entries, 1 exit)
* doubly ended queue -
neighbours at distance 1 inserted at front, and neighbours at distance 2, at the back of the queue

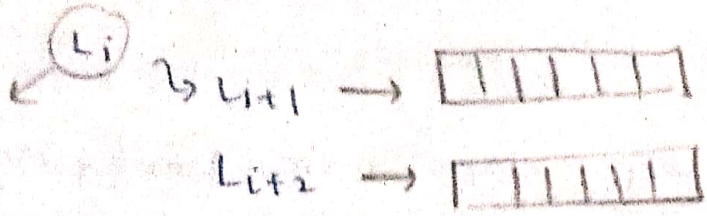
won't work with
↓
automatically sorts the neighbours by the basis of distance.

edges
$O(E)$

Approach 2 : insert a dummy / auxiliary node
whenever 2 is encountered.
(can be represented depending on implementation of graph)

vertices
total space
$O(V+2E)$
$O(V+E+E) \rightarrow O(V+E)$
total space

Approach 3: maintaining 3 queues for 2 weight



the general queue

extending to L_i, L_{i+1} queues can be maintained (given k is very small)

$O((V+E)\log V)$ (Dijkstra's)

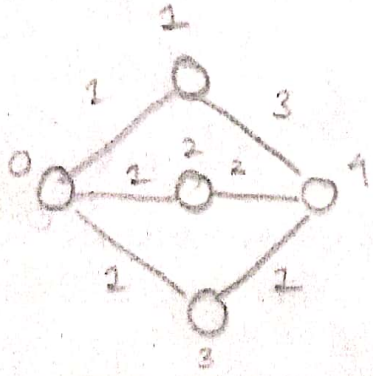
→ think clique

$\leq O(kV + E)$

(k∧V) queues at max, if not operated differently

if k is small, then can be treated as an alternative approach to Dijkstra's.

A metric can be pushed to the system of queues at most k times, & edges are traversed for each node



0	1	2	3	4
0	1	3	2	~~4~~ 3

2

q_0 : ~~0~~

q_1 : 1, 2, 3

q_2 : X

q_3 : X

dist(0,4) = 2

q_0 : ~~1~~

q_1 : 4

q_2 : 4

q_3 : X

q_0 : ~~2~~, ~~2~~, ~~3~~

q_1 : ~~4~~

q_2 : ~~4~~

q_3 : ~~4~~

To utilize only k+1 queues it is important to copy all queues to the queues one level above, except obviously the 0th ~~rate~~ level queue. and to accordingly operate.

↓

this approach is practical if k is small, in other cases Dijkstra's algorithm is used.

↓

one approach

↓

~~Another~~ another approach could be to implicitly use mod and accordingly change the queues.

↳ this is done till either all queues are empty or the final destination node has been reached.

q [(i+d) %. (k+1)]

~~*Approach 4 : Reduction of graph~~

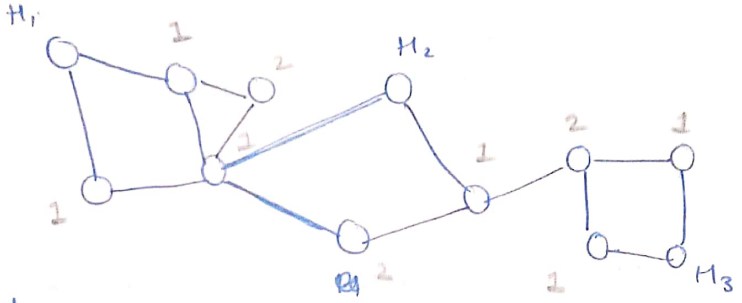
~~(Dorah's)~~

shortest path in a grid.

↓

each cell is a separate node and then applying bfs straight away on that or by constructing a graph, solve for grid.

Q.



Statement:

Farthest node from all houses.

↓

insert all houses at level 0, then all their neighbours if already not visited in level 1 or same queue with distance 1 and so forth; till all vertices have been processed.

↓

The ones at the end of the queue are the farthest and are at the highest distance from all.

(variation) Given one node as your house, figuring out nearest friend.

↓

Do a BFS from your house, the moment the current node at front of queue is a friend, we can say they are the closest and then operate the distance