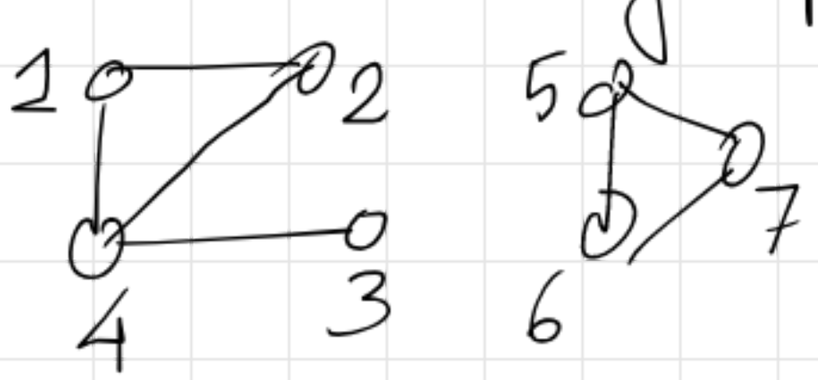
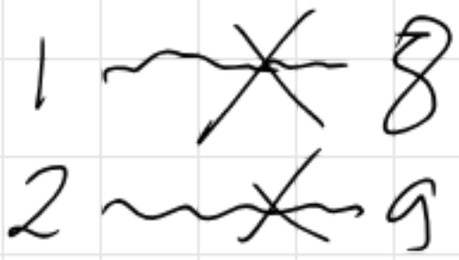
Reachability

Is there a path from s to t in G?

→ If graph is connected → always

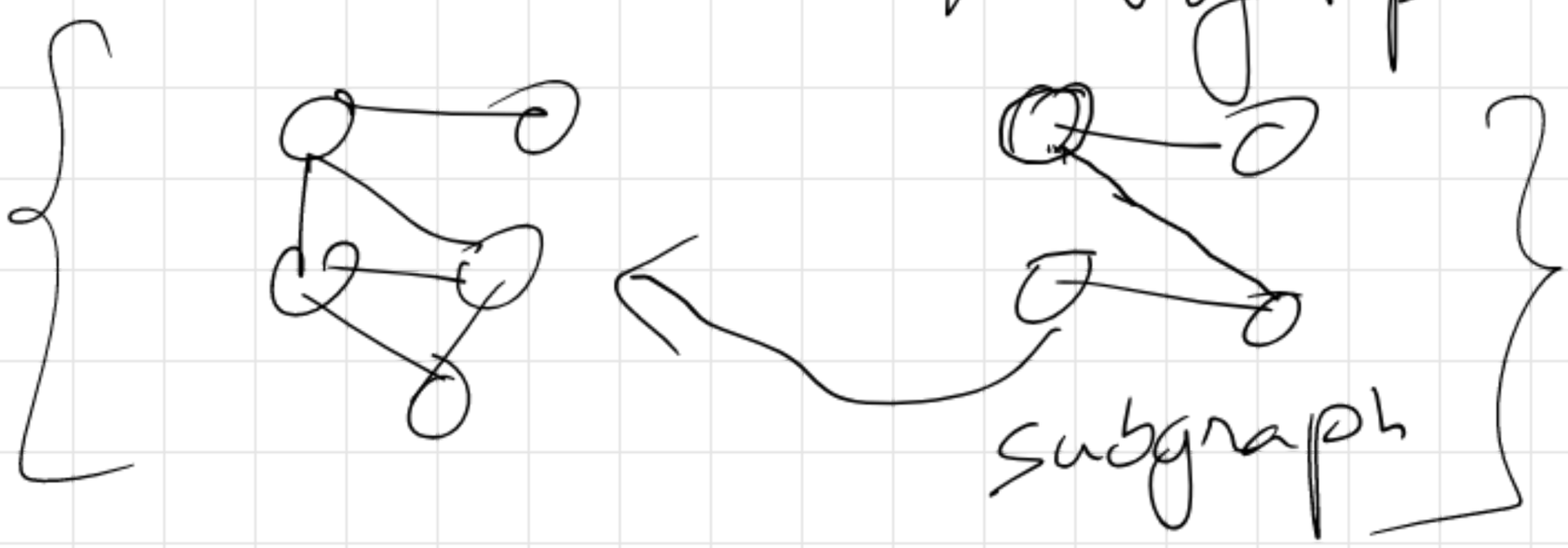
→ disconnected graph





<Connected Component>

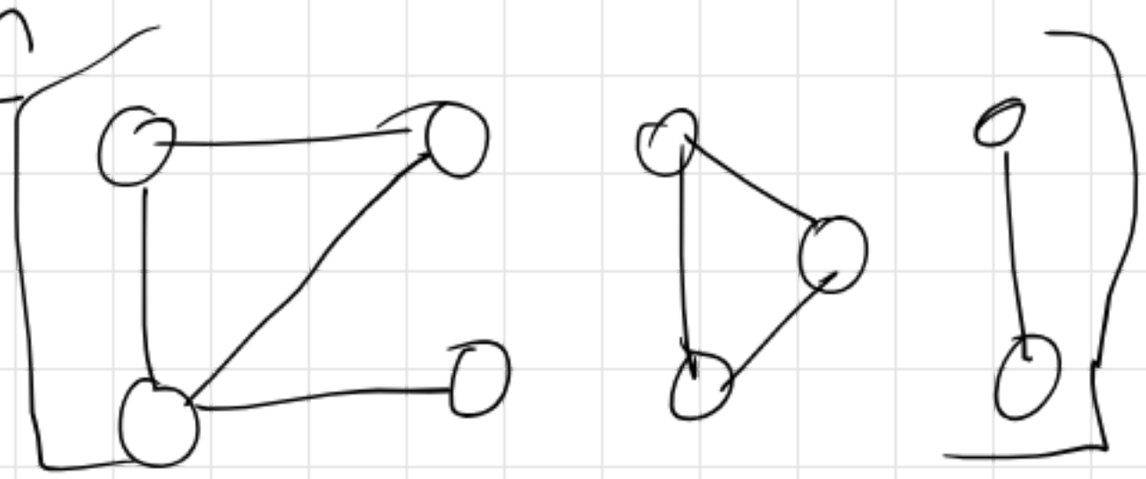
<Subgraph> subset of vertices and edges of a graph forming a graph

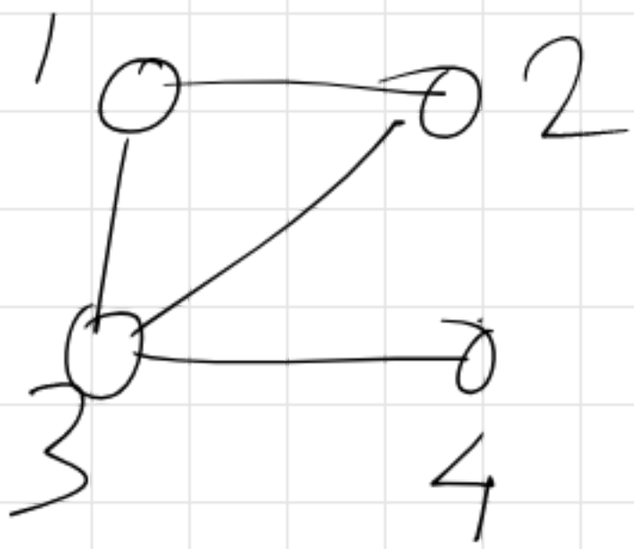


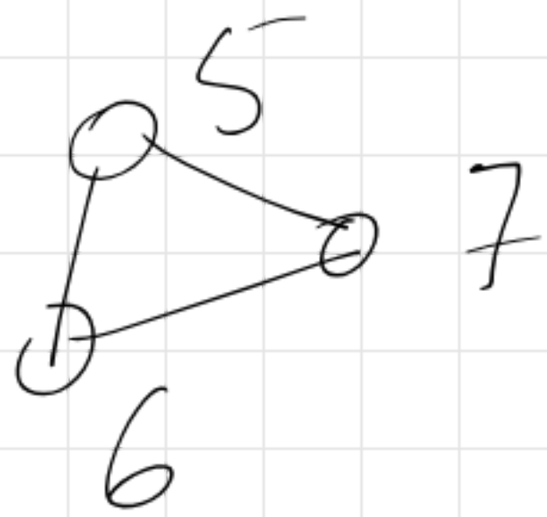
Connected Component

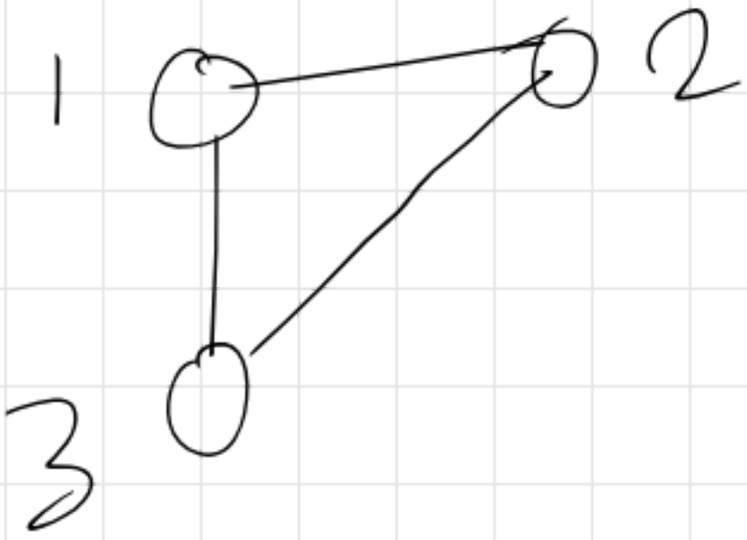
maximally connected subgraph of a graph

G

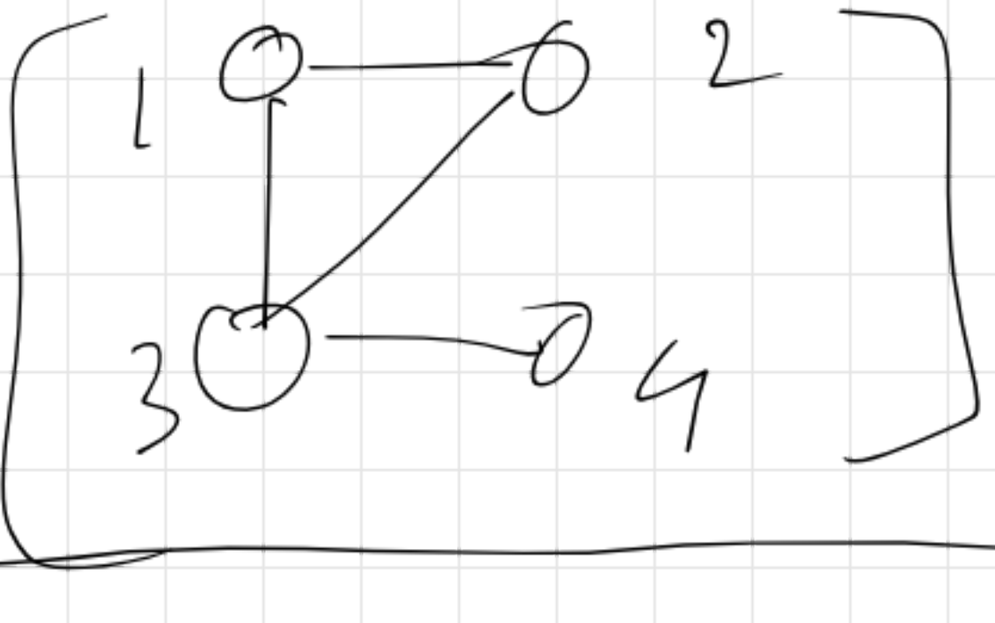








→ Connected + subgraph but not maximal



Connected component

Is there a path from s to t?

Observation s has a path to all the vertices that belong to the connected component containing s and doesn't have a path to any other vertex.

s-t connectivity → find connected component containing s

R : connected component containing S.

We build it (R) by exploring G

Algo

```
R = {s}
while ∃ e = (u, v) in E : u ∈ R & v ∉ R
    R = R ∪ {v}
end while
```

Claim: R is precisely the connected component / containing s.

1. ∃ an s-t path to every node t ∈ R
2. for no node t outside R, there is no s-t path.

1) Proof: Consider an edge (u, v) for which v is added to R,

→ u ∈ R, ∃ s ⇝ u path

⇓

s ⇝ u — v ($s \overset{\text{path}}{\rightsquigarrow} v$)

2. $s \in R$. Consider $w \notin R$

Assume that $\exists$ s – w path P.

There should be atleast one node $v \notin R$, but it falls in the path P.

$s \neq v$ [($v = w$ possible)]

$u \in R$

[$u - v \in E$ then $\Rightarrow v \in R$]

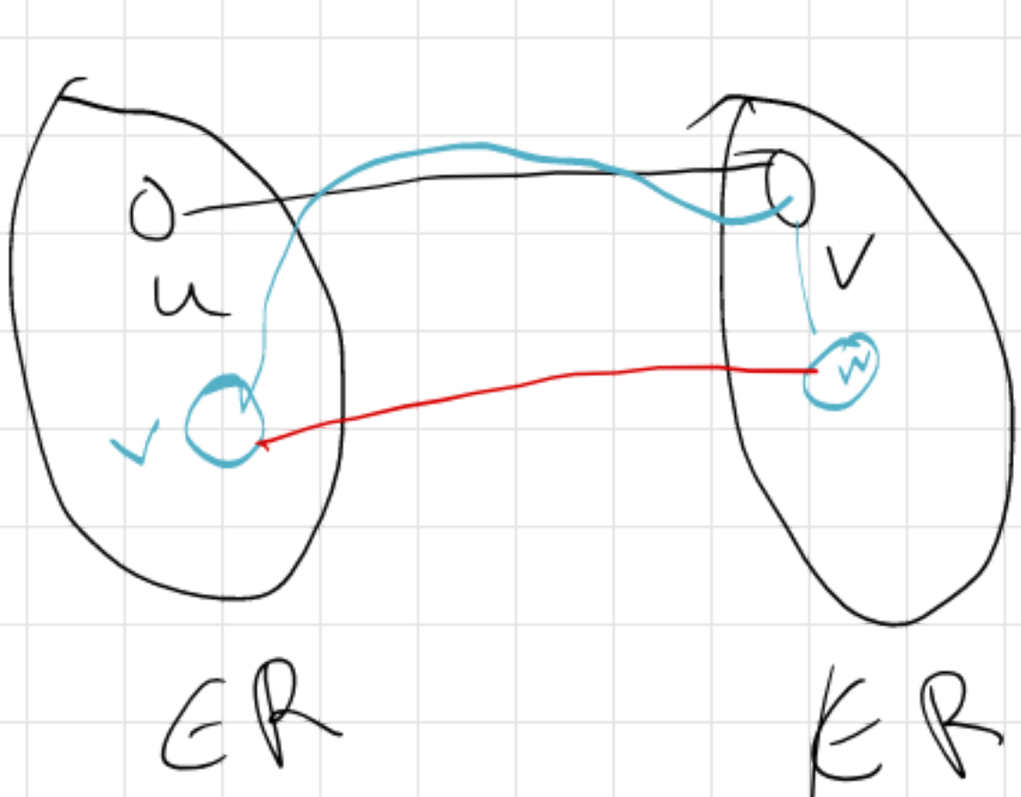
$\Rightarrow \Leftarrow$

Algorithm is underspecified,

< which edge to pick >

we didn't specify which edge to pick

Specify the idea → different algorithm.



Two diff

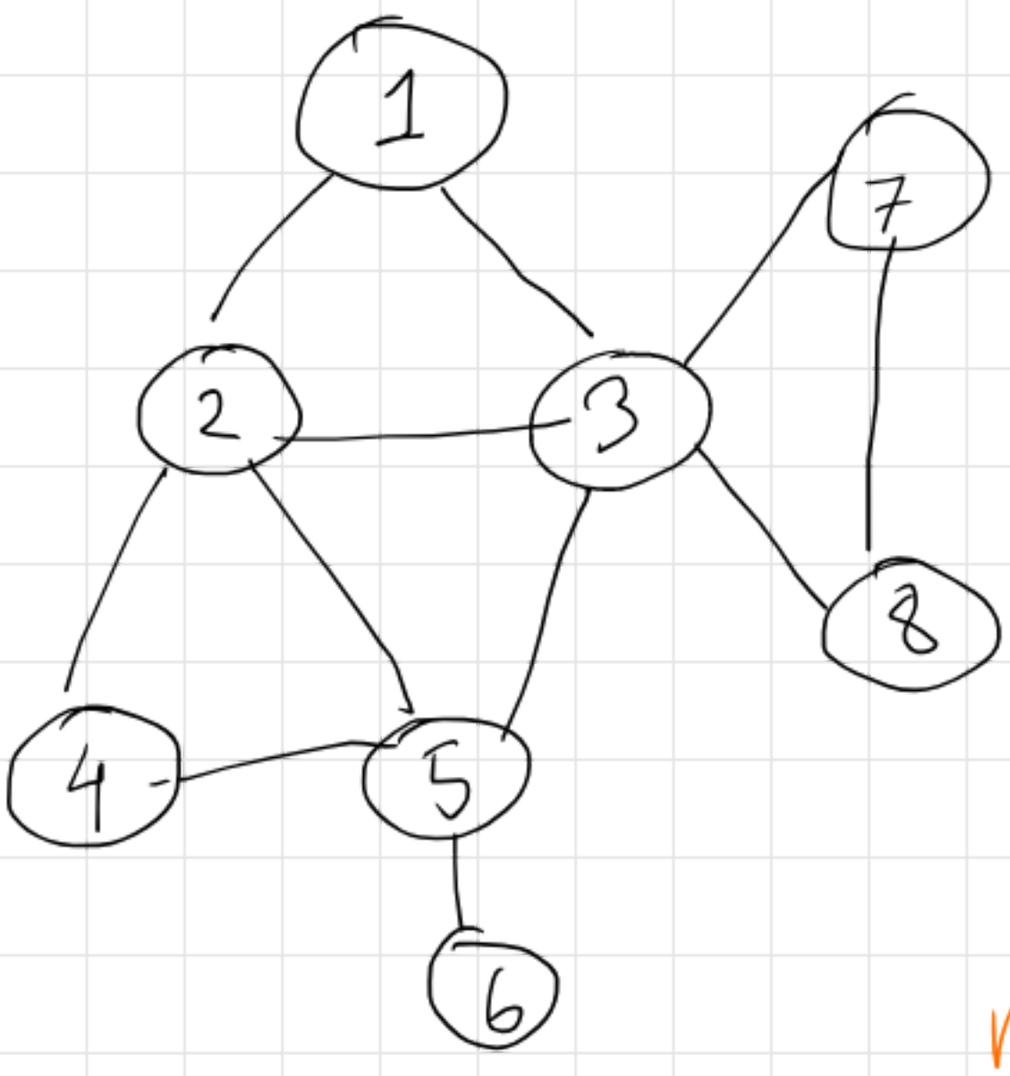
BFS

DFS

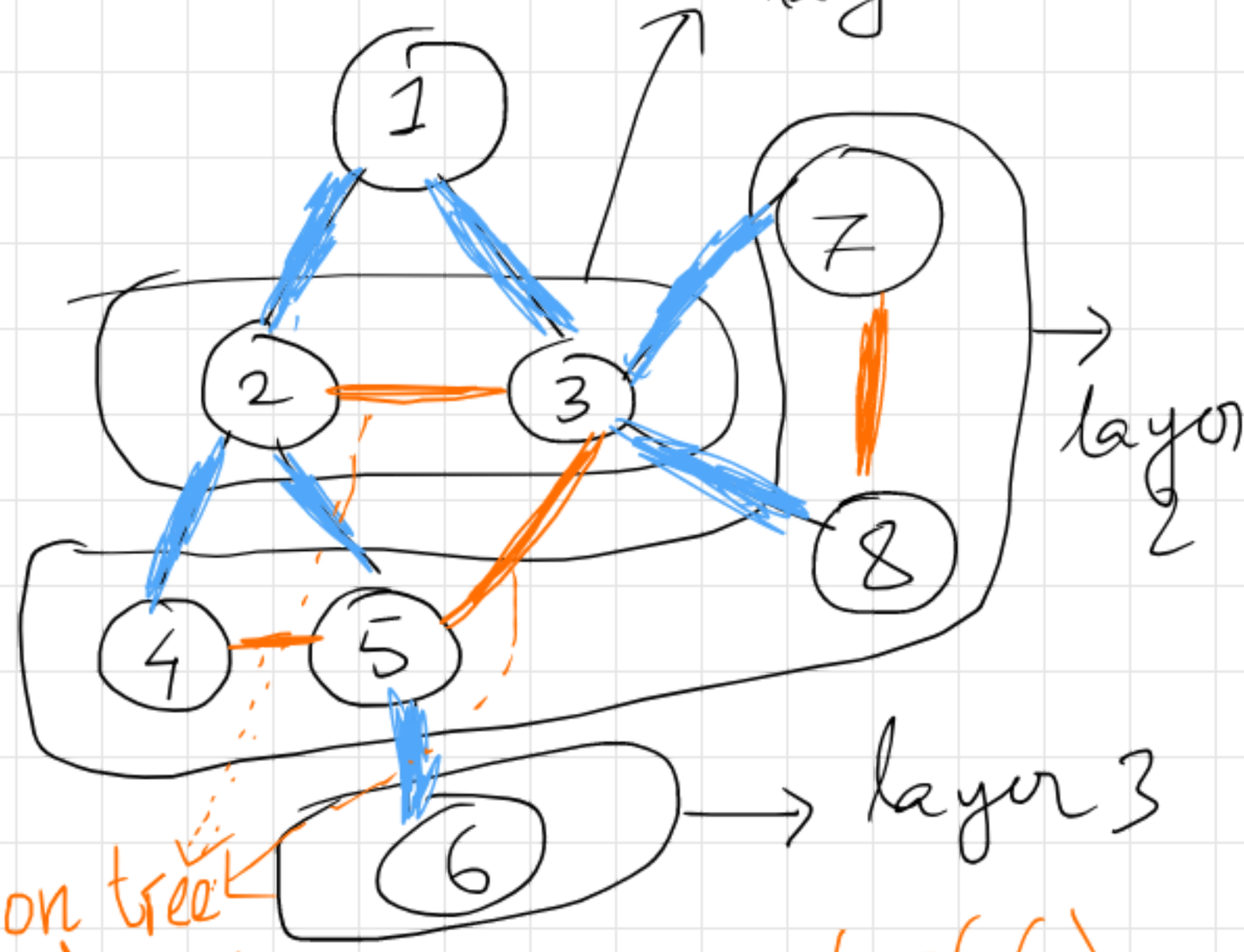
"Graph Traversals"

Topics BFS, DFS
- Application of BFS
- Application of DFS

Breadth First Search



[Blue → BFS Tree edge]



Layer 0 → starting node s (= 1)

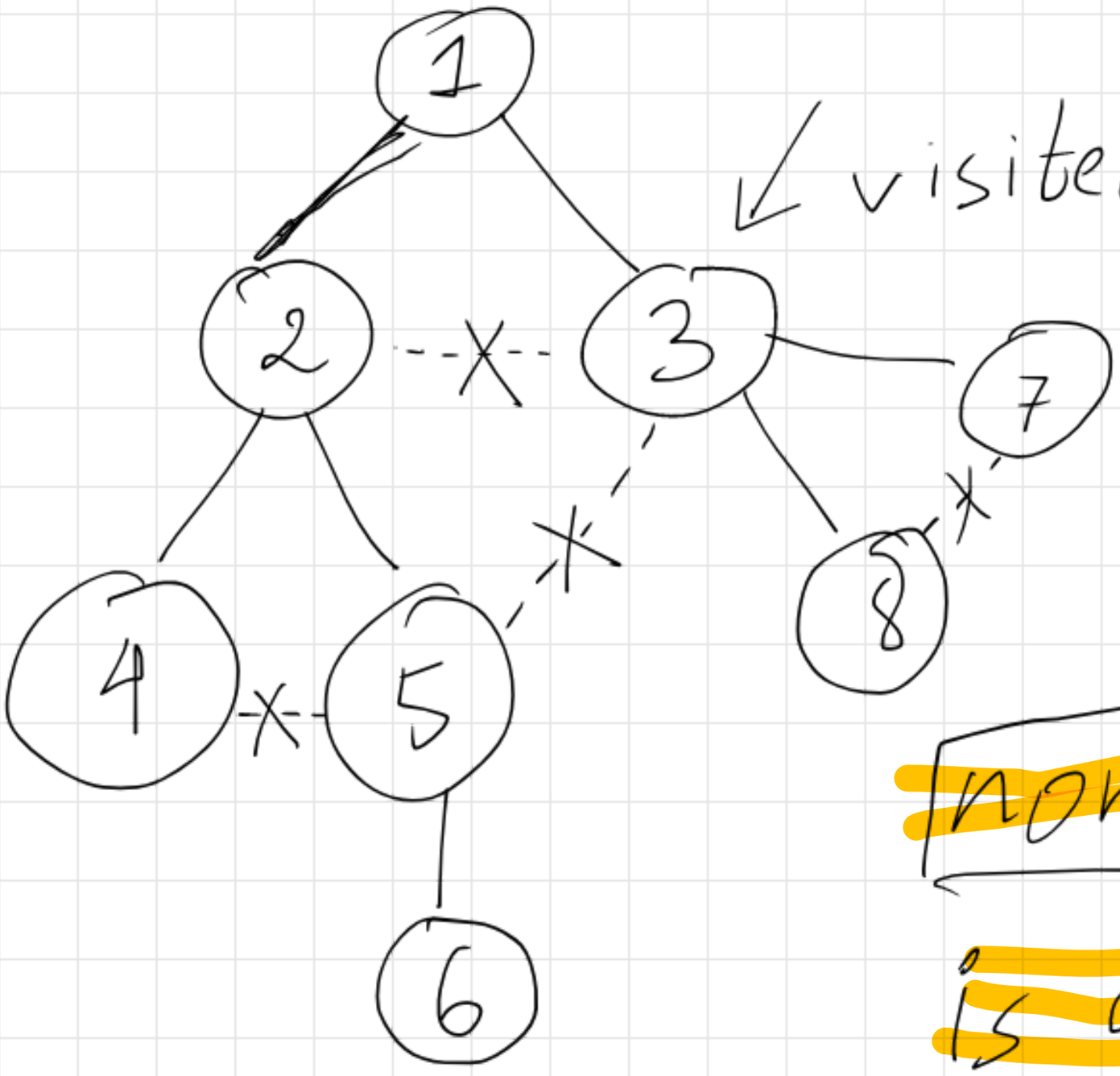
Layer 1 → neighbors of node s (= 1) (2, 3)

Layer 2 → (4, 5, 7, 8) (neighbor of 2 and 3)

Layer 3 → (6) (neighbor of 5)

BFS algorithm produces a tree (rooted at s) with all nodes reachable from s.

BFS Tree



they are part of G.

non tree edge is always b/t two layers L_i and L_j $i < j$

$|i - j| \leq 1$

Claim: Let T be a BFS tree, x and y be nodes in L_i and L_j respp and let $(x, y) \in E$ of G. Then i and j differ by at most 1.

Proof W.L.O.G suppose

$i \leq j$ (Fact)

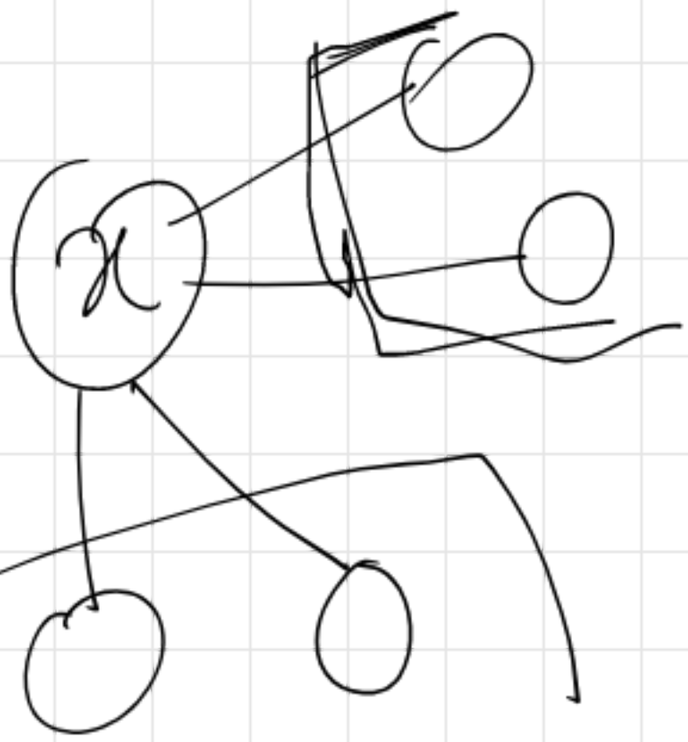
Suppose by contradiction

$i < j - 1$

gap is atleast 2

$x \in L_i$ $y \in L_j$

Consider the point when x was examined during algo.



$y \in$ undiscovered

$j = i + 1$

$y \in$ discovered

layer(j) ≤ Layer(i)

$j \leq i$

$j \leq i + 1$ (contradicts fact)

Thus $i \leq j$ and $i < j - 1 \Rightarrow$ desired.

Imp: Suppose there are layers $L_1, L_2, \ldots, L_j$ then layer L_{j+1} consists of all nodes that do not belong to an earlier layer and that have an edge to a node in layer L_j (but not in any layer L_k; $k \neq 1, 2 \ldots j-1$).

Thm: For each $j \geq 1$, layer L_j produced by BFS consists of all nodes at a distance exactly j from starting node (s).

BFS (pseudo-code)

1. Mark starting node s as explored, all other vertices as unexplored
2. Q = a queue data structure, initialized with s
3. while Q is not empty do
4. Remove the vertex front of Q, Call it v
5. for each edge (v, w) do
6. if w is unexplored then mark w as explored and add w to the end of Q

Line 5: Algo iterates through all the edges incident to vertex v (if G is undirected) or through all outgoing edges from v (if G is directed).

Time Complexity

Adjacency List:

Loose bound: When we consider a node v (in line no. 5), we look through all edges (v, w) incident to v. There can be at most n such edges, and we spend O(1) time considering each edge. So, total time spent on one iteration is O(n).

there are at most n iterations

$\therefore O(n^2)$

Improve bound: for a node v, we look only neighbors of v (which is not necessarily all the nodes)

$O(\deg(v))$ is req^{d}.

this we do for all the nodes

$$\sum_{v \in G} \deg(v) = 2|E|$$

$$= O(m)$$

Additional time to set up (explored & unexplored)

$$= O(n)$$

$$O(m+n)$$

⟨Application of BFS⟩

1) Shortest path from v to w in a graph G

Input: An undirected (directed) Graph $G=(V,E)$
and a starting vertex $s \in V$.

Output: $dist(s, v)$ for every $v \in V$.

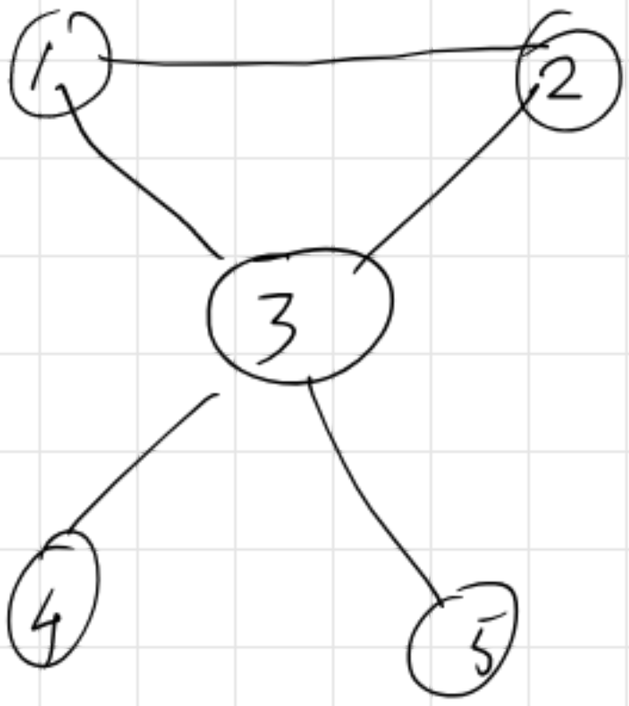
Same algorithm as before.

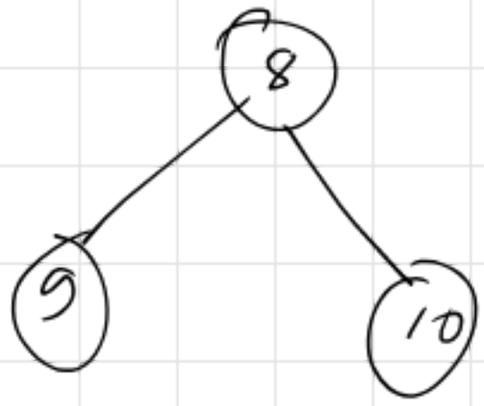
1. mark s as unexplored, all other vertices as unexplored
2. $l(s)=0$, $l(v)=+\infty$ for every $v \neq s$
3. $Q :=$ a queue, initialized with s.
4. while Q is not empty do
5. remove the vertex from the font of Q call it v.
6. for each edge (v, w) :
7. if w is unexplored then
8. mark w as explored
9. $l(w) = l(v) + 1$
10. add w to the end of Q.

$O(m+n)$ [if $l(w) = +\infty \rightarrow$ no path from s to w exists]

2> Computing Connected Component

→ only for undirected graph.





Maximal Connected Subgraph

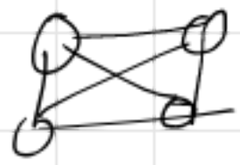
A connected component is a maximal subset $S \subseteq V$ s.t. there is a path from any vertex in S to any other vertex in S

Foreg: $\{1, 2, 3, 4, 5\}$ $\{6, 7\}$ $\{8, 9, 10\}$

Quiz: Consider an undirected graph with n vertices and m edges. What are minimum and maximum no. of conected components that the graph could have?

$min = 1$

$max = n$



o o o o

Why we need to find connected components.

→ find applications

Discovered[i] = F
for i = 1 to n

for i=1 to n
  if discovered(i) = F
    BFS(i)

during BFS(i), make discovered[] = T for that node

Pseudo code

1. Mark all vertices as unexplored
2. num = 0
3. for i = 1 to n // loop all vertices
4. if i is unexplored
5. num = num + 1
6. Q = a queue, initialized with i
7. while Q is not empty
8. remove the vertex from the front of Q
   Call it v
9. CC(v) = num
10. for each (v,w) ∈ E
11. if w is unexplored
12) mark w as explored
13) add w to end of Q.