WEEK 1

WHY GRAPHS?

① Check if network is connected: example (Kevin Bacon number movie network)

② driving directions in Google Maps

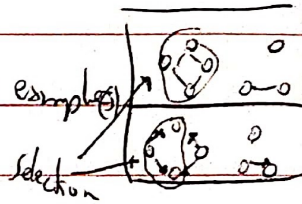
③ formulate a plan when only certain changes can be applied to a system

④ Compute the "pieces" (or "components") of a graph.

- clustering or structure of a certain graph.

GENERIC GRAPH SEARCH

Goals ①: find everything findable from a given start vertex: example



② Don't explore anything twice Goal: $O(m+n)$

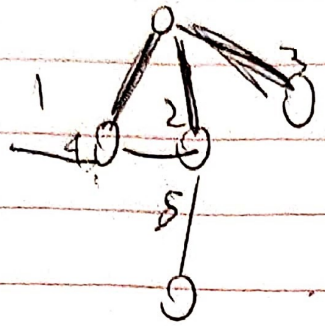
Generic algorithm (given graph G, vertex s)

- initially only s has been explored
- while possible:
  - choose edge (u,v) with u explored and v unexplored, (if none, halt)
  - mark v explored (explored.append(v))

Claim: at the end of the algorithm v explored $\Leftrightarrow$ G has a path from s to v

Note: how to choose between frontier edges?

BFS



etc, explore all nodes some of distance before exploring their ch

DFS plunges as deep as it can. Backtracks when necessary

BFS: O(m+n) time using a Queue (F.F.O)
to compute shortest paths
compute connected components of an undirected graph

DFS:
- Compute topological ordering of directed acyclic graph
- Compute "connected components" in directed graphs

Breadth First Search

THE CODE

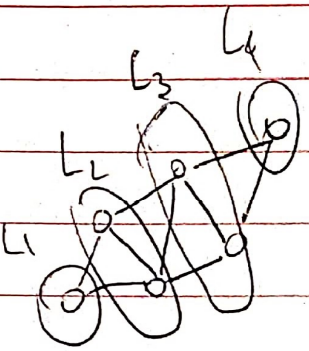
BFS (graph G, start vertex s)

explored = [ ] # all nodes initially unexplored

explored.append(s) #mark s as explored

Let Q = queue data structure (FIFO), initialized with s.

- while Q ≠ ∅:
  - remove the first node of Q, call it v
  - for each edge (v, w)
    - if w unexplored
      - mark w as explored
      - add w to Q



APPLICATION OF BFS: SHORTEST PATHS

Goal: compute dist(v), the fewest # of edges on a path from s to v

Extra code (to add to BFS-2 code):
- initialize $dist(v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \end{cases}$
- when considering edge (v, w):
  - if w unexplored, then set dist(w) = dist(v) + 1

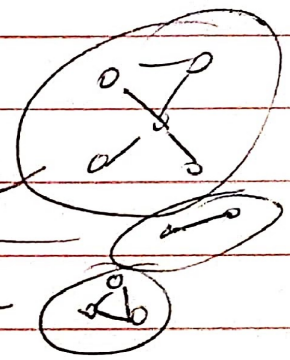
WEEK 1 ✓

APPLICATION OF BFS: UNDIRECTED CONNECTIVITY

Directed/undirected connectivities are very different.

Let $G = (V, E)$ be an undirected graph.

Connected components = the "pieces" of G



Formal definition:
Equivalence classes of the relation $v \sim w \Leftrightarrow \exists$ v-w path in G

Goal: compute all connected components

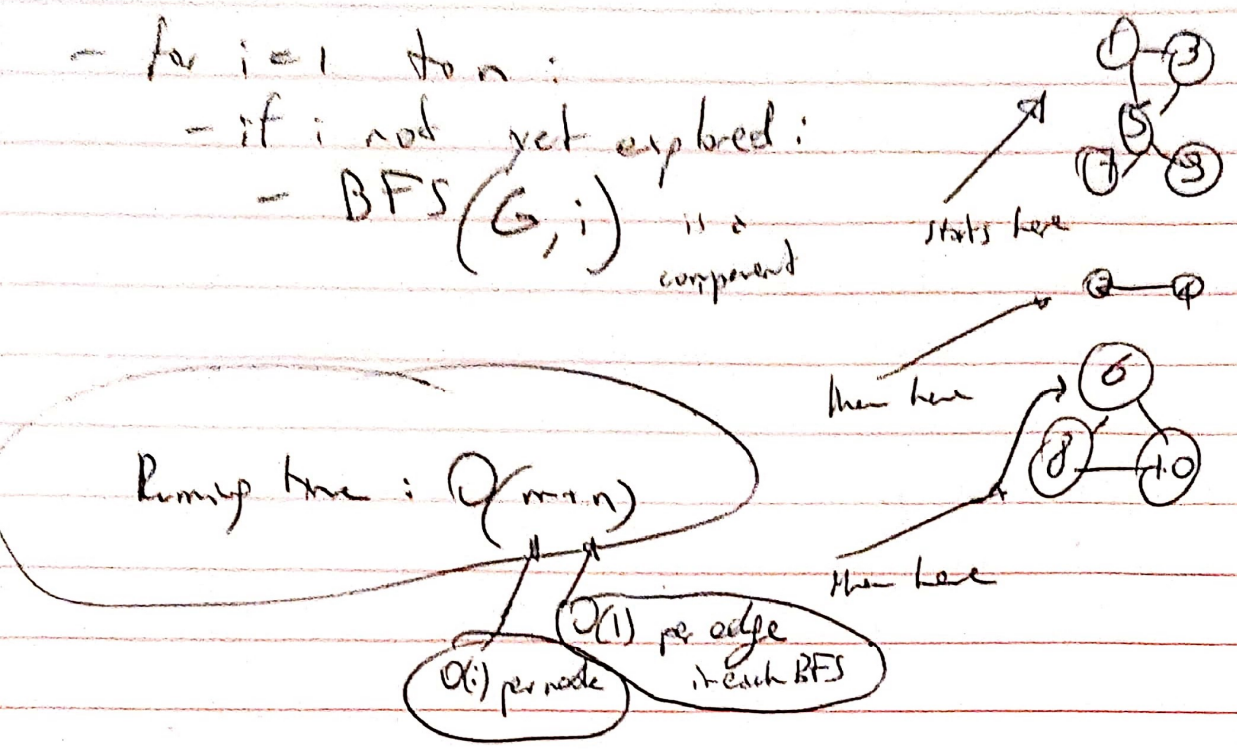
Why?:
- Check if network is disconnected.
- Display graph?
- clustering

CONNECTED COMPONENTS VIA BFS

To compute all components (undirected case):

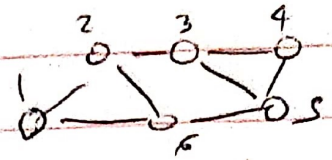
- all nodes unexplored

- for i = 1 to n:
  - if i not yet explored:
    - BFS(G, i) // i a component



APPLICATION OF DFS: UNDIRECTED CONNECTIVITY

DFS



Backtrack when necessary (if we've already been

- computes a topological ordering of a directed acyclic graph
- and strongly connected components of directed graphs.

Run time is $O(m+n)$

TIME CODE

DFS, (LIFO) instead of (FIFO)

DFS(graph G, start vertex s):
  Lifo = []
- mark s as explored
- for every edge from Lifo (s, v):
  - if v unexplored, add to Lifo

Running Time is $O(ns + ms)$

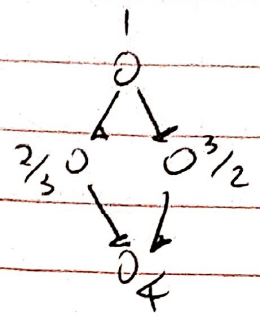
ns: # of nodes reachable from s

ms: # of edges reachable from s

FINALLY, WHAT THE HELL IS A TOPOLOGICAL ORDERING (of a directed acyclic graph)

Definition: A topological ordering of a directed graph G is a labelling F of G's nodes s.t.:

1. The $F(v)$'s are the set $\{1, 2, \ldots, |n|\}$
2. $\forall (v, w) \in G \Rightarrow F(v) < F(w)$



Motivation: Sequence tasks while respecting all precedence constraints

(COND1) DO: You need the graph to be acyclic !!!!!

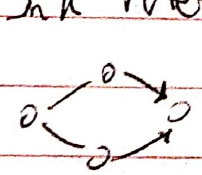
Or you're screwed too!

Theorem: (IF COND1) I can compute TOrd in linear time.

STRAIGHTFORWARD SOLUTION

Note: every directed acyclic graph has at least one sink vertex

(Duh, imagine 0 → 0 then induction)



To compute topological ordering:

The only candidates for last position are sink vertices

- let $f(v) = n$
- recurse on $G - \{v\}$

TOPOLOGICAL SORT via DFS (Slick).

M1

DFS-Loop (graph G)
mark all nodes unexplored
current_label = n (to keep track of ordering)

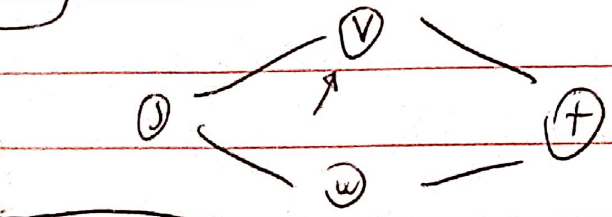
for each vertex $v \in G$
- if v not yet explored (in some previous DFS call)
  - DFS(G, v)

M2

DFS(graph G, start vertex s)
- mark s explored
- for every edge (s, v):
  - if v not yet explored
    - DFS(G, v)

Crucial bit: remember what you've explored already

M1)



v → t, sink so append t to s(t)
Back to v, sink so 3
Back to outer for loop, yay
Skip v because already explored!
Move to w

M2)

"	"
"	"
"	"
Check v	
:	

$O(n+m)$, correct.

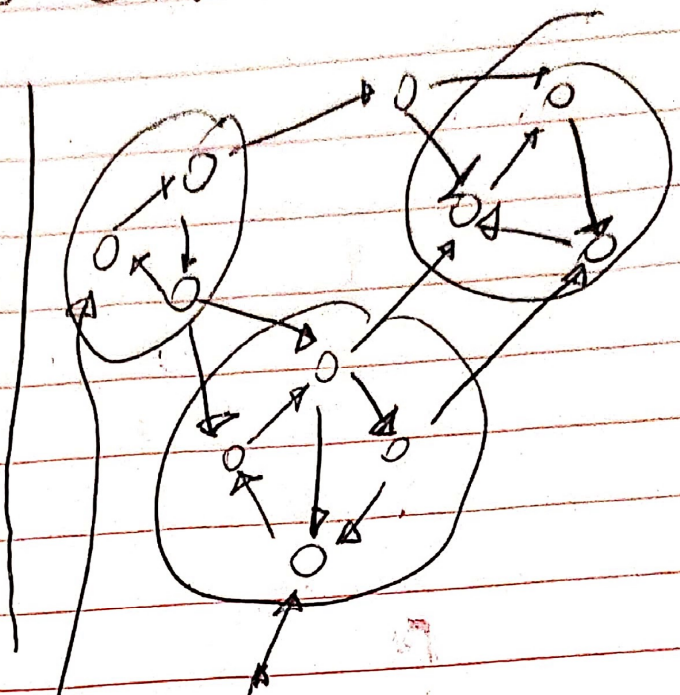
$a=1$

$b=1$

$d=$ calculate, $b^d=2$, case 1,

STRONGLY CONNECTED COMPONENTS in directed graphs

SCCs are cycles and nodes out of cycles



Why DFS?

If we start from ; won't just find SCC. Get a union of possible SCCs

If we start from , all graph!

Need to choose where we want to start.

KOSARAJU'S TWO PASS ALGORITHM

SCCs in $O(m+n)$ time

Algo:

① Reverse all arcs, let G^{rev} = G with all arcs reversed

② Run DFS on G^{rev} ← goal: compute "magical ordering" of nodes

③ Run DFS on G ← goal: discover the SCCs one by one

process nodes in decreasing order of finishing times from ②

DFS-Loop

DFS Loop (graph G)

global variable $t = 0$ (# of nodes finished processing so far)

global variable $s = NULL$ (current source vertex) (for leaders in 2nd pass)

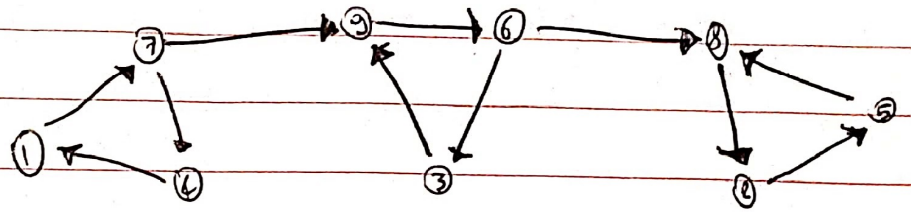
Assume nodes labeled with finishing times 1 to n

for $i = n$ down to 1:
  if i not yet explored:
    DFS (G, i)

DFS (Graph G, node i):

- mark i as explored
- set leader(i) = node s
- For each arc $(i, j) \in G$:
  - if j not yet explored:
    - DFS (G, j)
- t++
- set $f(i) = t$ (finishing time)

G^{rev}



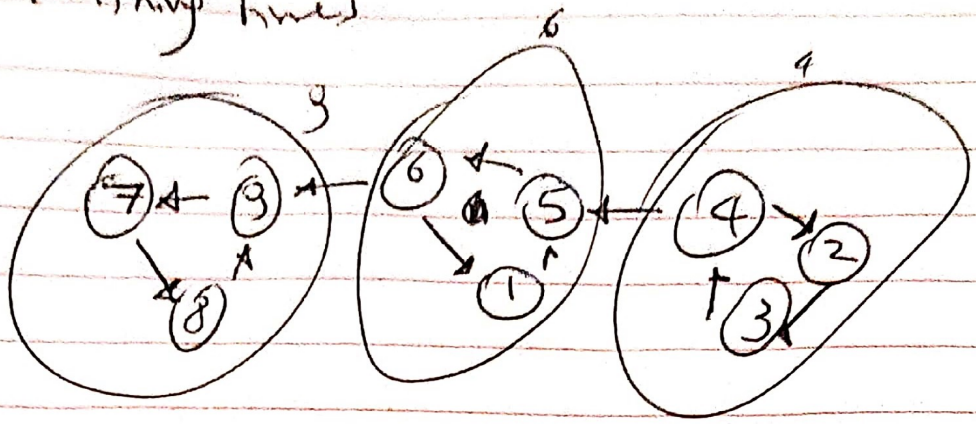
Starting from 9: 9, 6, 3, finish! $f(3) = 1$
Back to 6: 6, 8, 2, 5, finish! $f(5) = 2$
Back to 2: 2, finish! $f(2) = 3$
Back to 8: 8, finish! $f(8) = 4$
Back to 6: 6, finish! $f(6) = 5$
Back to 9: 9, finish! $f(9) = 6$

ALL of these have leader 9!

$f(n) = [7, 3, 1, 8, 2, 5, 9, 4, 6]$

Next [illegible] unexplored node: 7

All the DFS is executed, 9 nodes' names change to finishing times.



Run DFS again for leaders. top → bottom

from 9: 9, 7, 8, done!, leader 9

from ~~8~~, 7, 6: 6, 1, 5, done!, leader 6

from ~~8~~, 4: 4, 2, 3, done!, leader 4.

Running time: 2 × DFS = O(m+n)

WEEK 1 L8

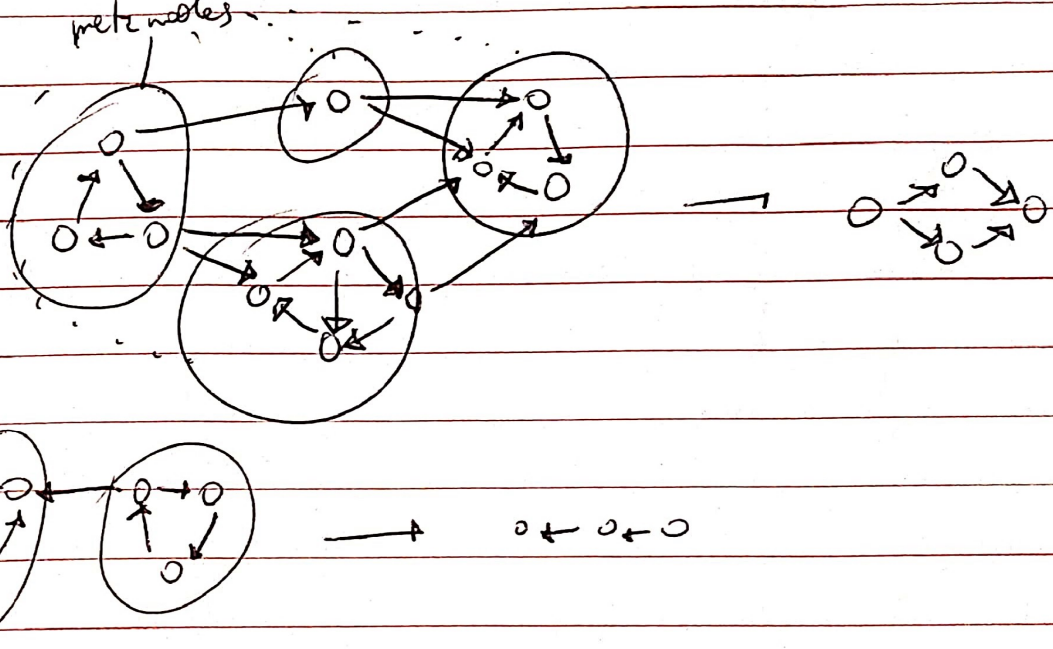
GENERAL ARGUMENT FOR LINEAR TIME

OBSERVATON

Claim: The SCCs of a directed graph induce an acyclic "meta-graph"

- meta ~~graph~~ nodes: the SCCs
- $\exists$ arc $C \to \hat{C} \Leftrightarrow \exists$ arc $(i,j) \in G$, with $i \in C$, $j \in \hat{C}$

Examples:



The meta-graph is always acyclic, any cycle of SCCs is just one SCC.

THE SCCs ~~ARE~~ the same IN G and G^{rev}

(and in the second pass of DFS in Kosaraju's algo too!)

Key lemma: | We'll prove this later

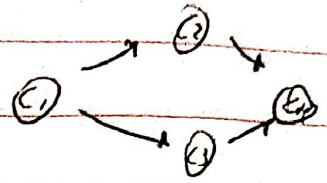
Consider the "sink" SCC in G

(cont)

...d pass of DFS (cop being somewhere in a sink SCC C^*

First call discovers C^* and nothing more.

After that, goes back up" to C_2 or C_3, etc etc.



Got it? Now let's prove the Lemma.

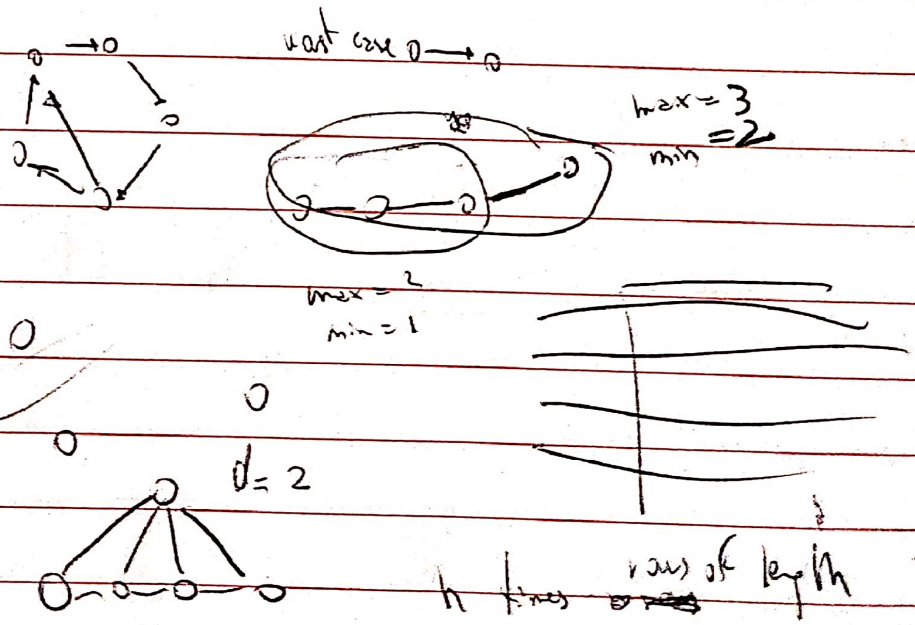
PROOF OF KEY LEMMA

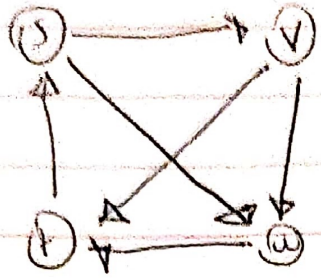
Case ($v \in C_1$): we'll see all of C_1 before C_2

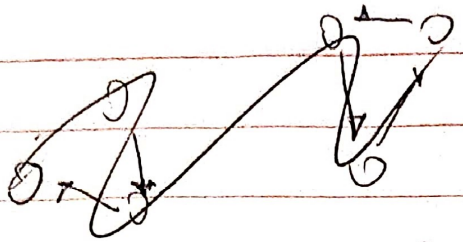
Case ($v \in C_2$): DFS (G^{rev}, v) won't finish until $C_1 \cup C_2$ completely explored

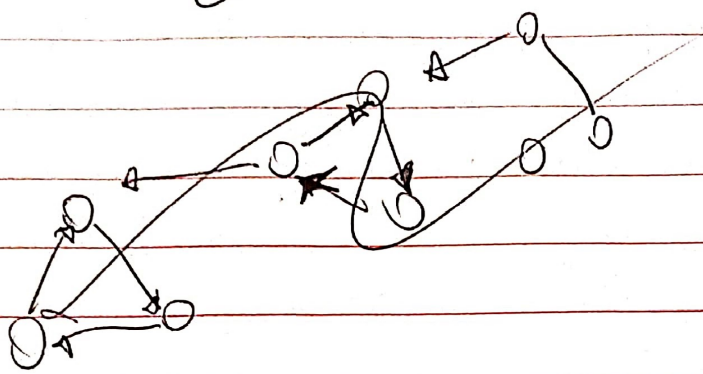
---

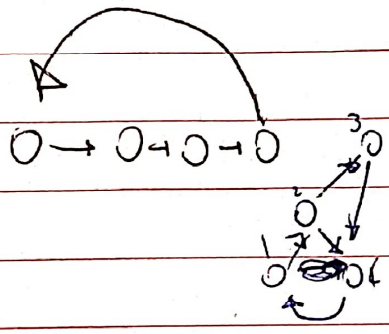
PROBABILITY



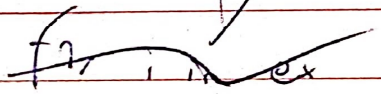








$$\left(1:[2],\ 2:[3,4],\ 3:[4],\ 4:[1]\right)$$



2

$$\left(2:\ \right)$$