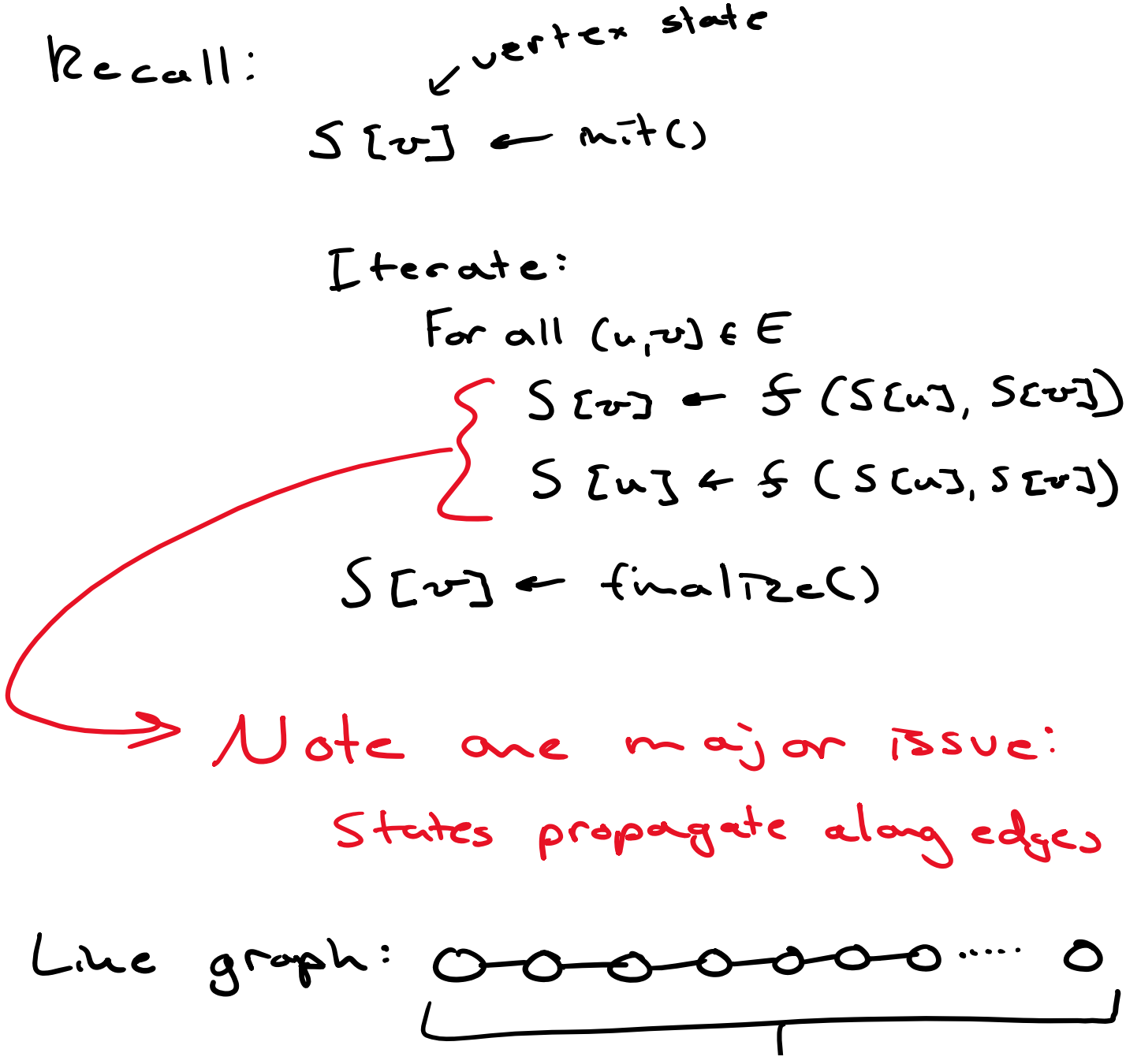
Lecture 2 - Graph Connectivity

Thursday, January 15, 2026 8:50 AM

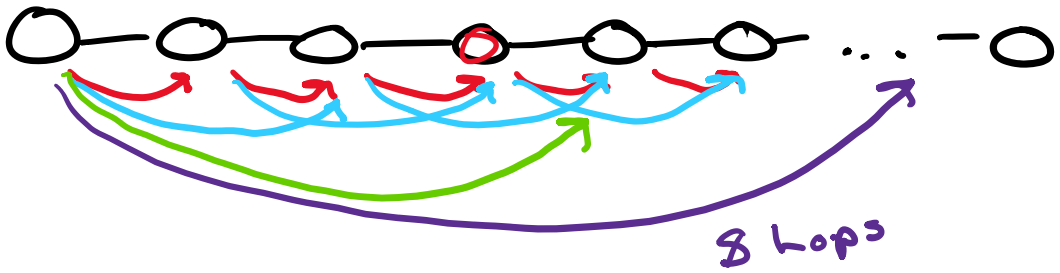
Plan for today:

- Discuss paper and lit reviews update
  - Papers released at 3pm today
  - Enter your requests in Submitty forum post
  - First-posted first-served
  - Also: you can select your own (relevant) papers
- Go over lec01.py code example
- Quick review of graph computations
- Biconnectivity and k-connectivity
- Directed graphs, strong and weak connectivity
- The web graph
- Connectivity in NetworkX
  - Connectivity and weak connectivity functions
  - Strong connectivity for next class



We require O(n) iterations

Solution: pointer-jumping

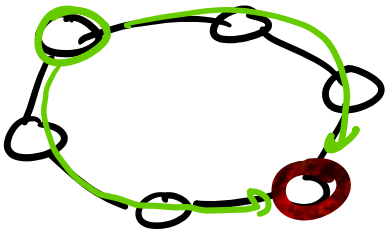


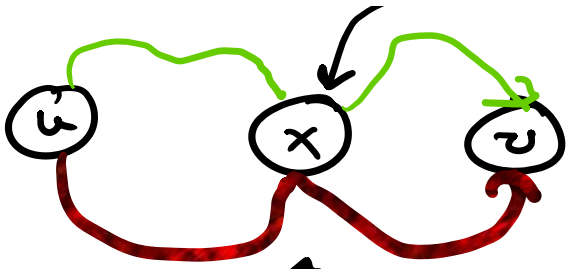
↳ we only require log(n) iterations for an equivalent convergence

---

Biconnectivity and k-connectivity

↳ a graph is biconnected if there exists at least 2 vertex disjoint paths between all pairs of vertice





not vertex disjoint

Cut vertex: removing a cut vertex will disconnect the graph

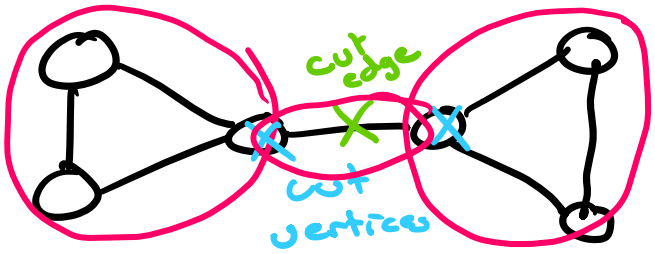
↳ increase the number of comps.

Biconnectivity: no cut vertex in a biconnected graph

What we care about: biconnectivity decompositions

↳ Consider:

1. Maximal biconnected components
2. Cut vertices — articulation vertex
3. cut edges — bridges



G

Note: a single edge is considered biconnected

Why do we care?

→ cut vertices and edges are "weak" or possible failure points in a network

→ Information diffusion gets concentrated at these points

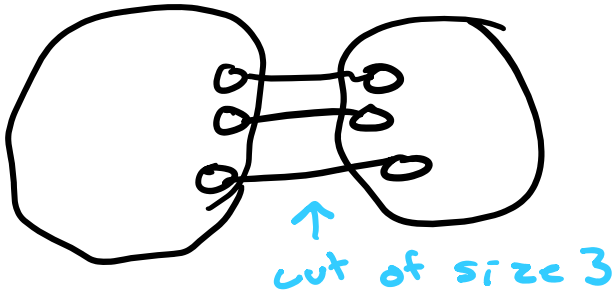
As we've noted:

most real graphs are not biconnected

Reason: trivial components are not uncommon

BUT: we can still care about *how* connected parts

* How * connected parts of a graph are



Generalize: k-connectivity
k-edge-connectivity

1-connected: basic connectivity
algorithms: propagate, BFS | DFS, pointer jumping, etc.

2-connected: biconnectivity, but an edge is not 2-connected
algorithms: Tarjan (DFS)
Slota-Madduri (BFS)

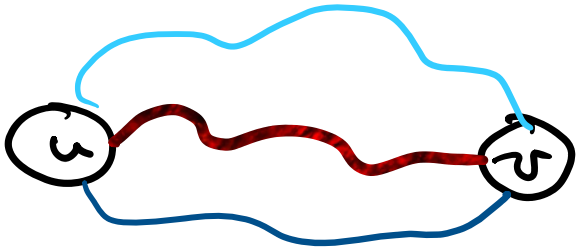
3-connectivity: triconnectivity
cut sets of vertices/edges of size 2
algorithm: Hopcroft-Tarjan (DFS)

k-connectivity:

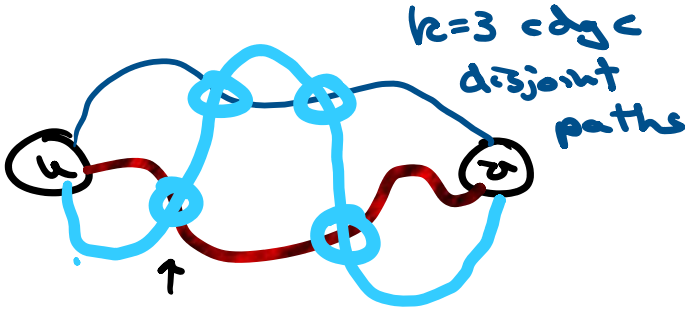
In general:

k-connected implies between all u,v vertex pairs there exists k unique vertex disjoint paths

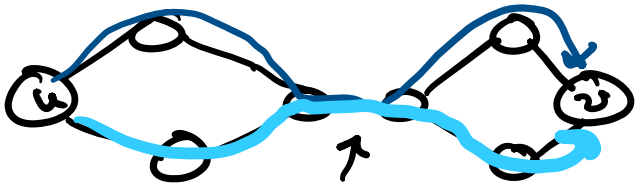
k-edge-connectivity: between all u,v vertex pairs there exists k unique edge disjoint paths



k=3 vertex disjoint paths



not vertex disjoint



not edge disjoint

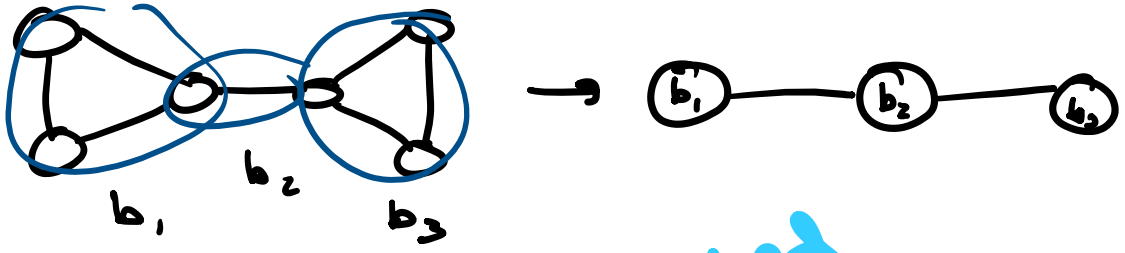
Really: this is all related to network robustness

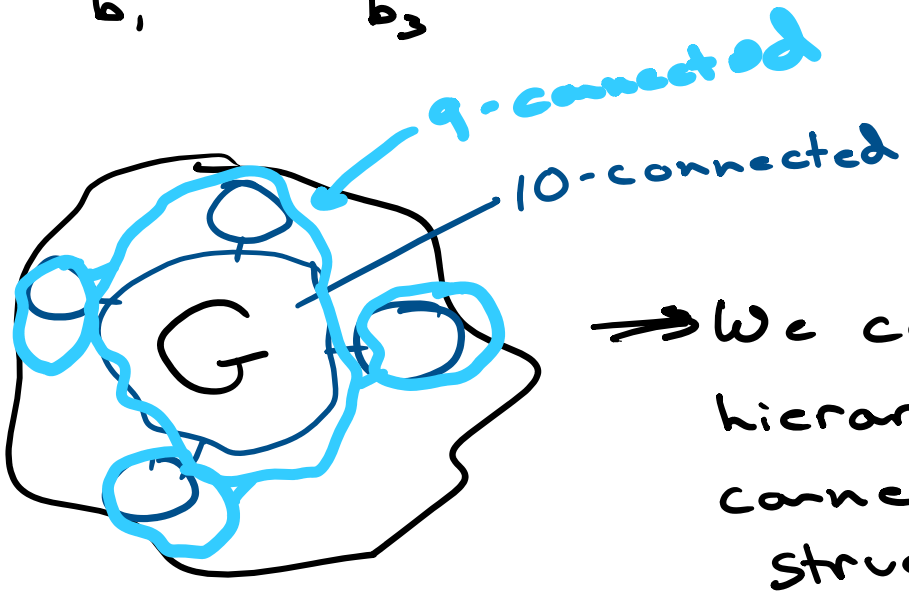
OR: where is information

OR: where is information diffusion being concentrated

One final thing:

we can iterate our connectivity decompositions to build a supergraph of connectivity information





→ We can build a hierarchical connectivity structure

Directed Graphs

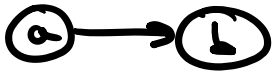
Directed graph $D = (U, E)$

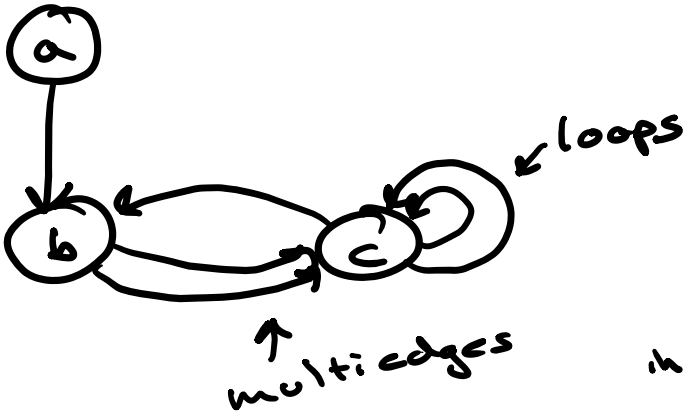
Directed graph $D = (V, E)$

$V = \{a, b, c, \ldots\}$

$E = \{ f = (a, b), g = (b, c) \ldots \}$

↑ directed edges





We also consider both in-degree and out-degree

$d^-(a) = 0$ in degree

$d^+(a) = 1$ out degree

$d^-(b) = 2$

$d^+(b) = 2$

$d^-(c) = 4$

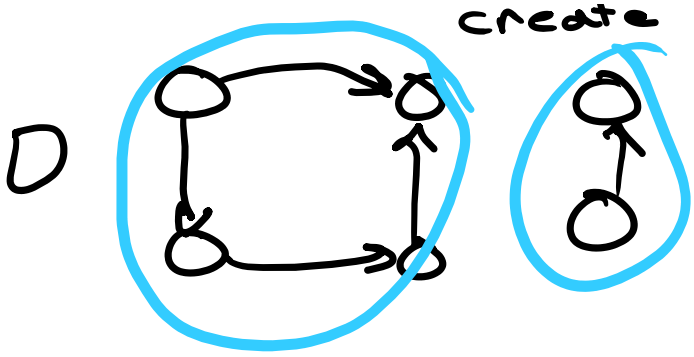
$d^+(c) = 3$

Weak connectivity: a graph is weakly connected if the underlying graph is connected

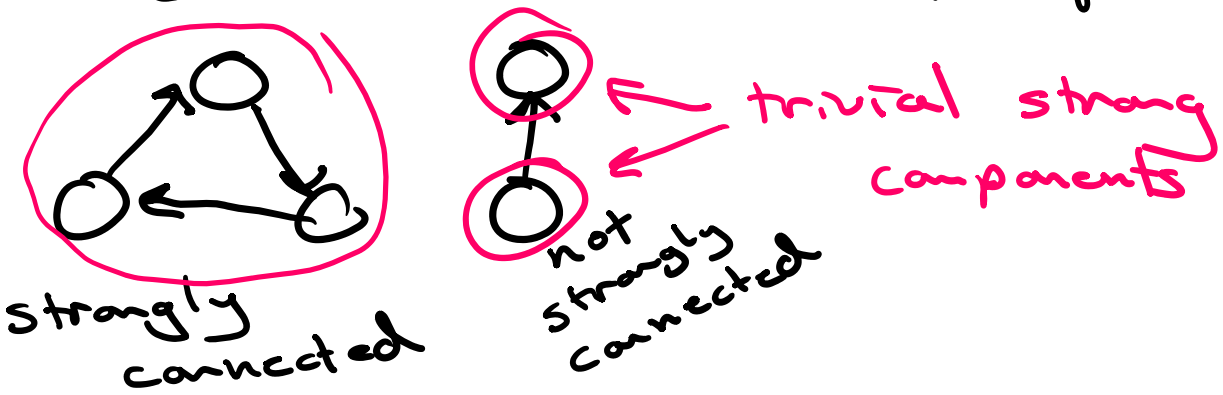
---

↳ ignore edge direction to create an undirected graph

create an undirected graph



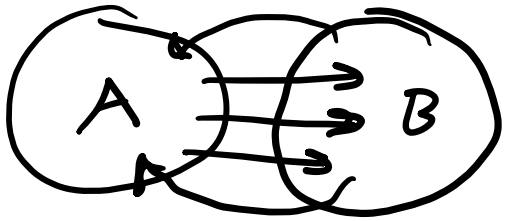
Strong connectivity: a graph is strongly connected if for all $u, v \in V(D)$ there exists a directed u,v-path



Algorithms: Tarjan (DFS)
FW-BW (parallel BFS-based)

Note: k-connectivity (edge) can also be generalized to directed graphs

cut $|[A, B]| = 3$



cut $|[A, B]| = 3$

cut $|[B, A]| = 2$

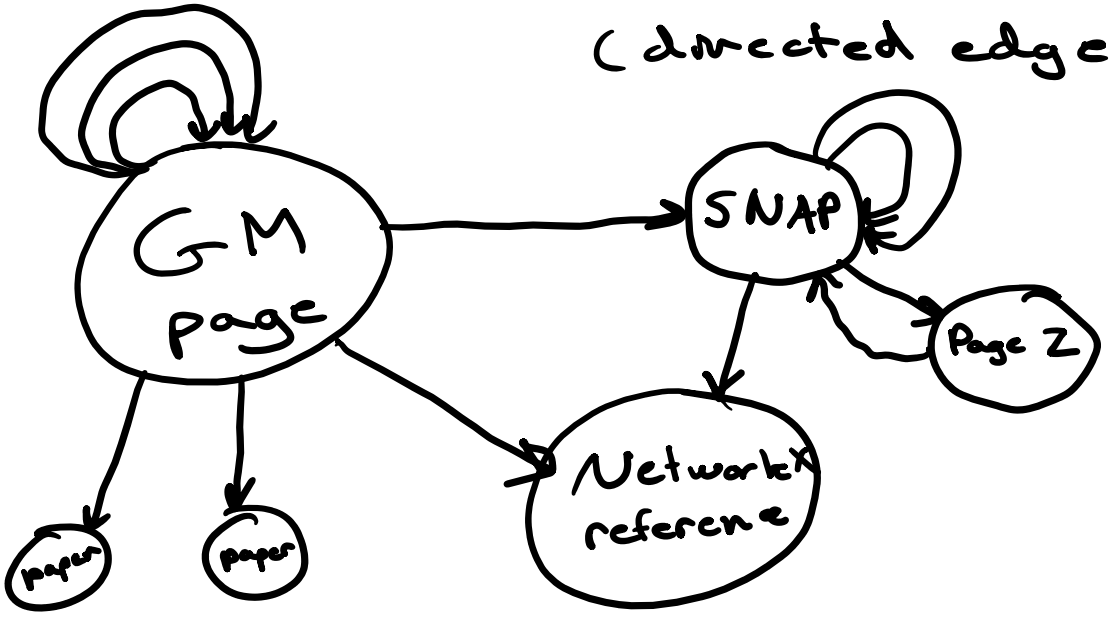
Let's put the above to practice on the web graph

web graph:

$V = \{$ web pages $\}$

$E = \{$ hyperlinks between pages $\}$

(directed edges)



The web graph considers all pages and all links

pages: vertices

links: directed edges

vertices

directed edges

The structure of the web:

How to determine?

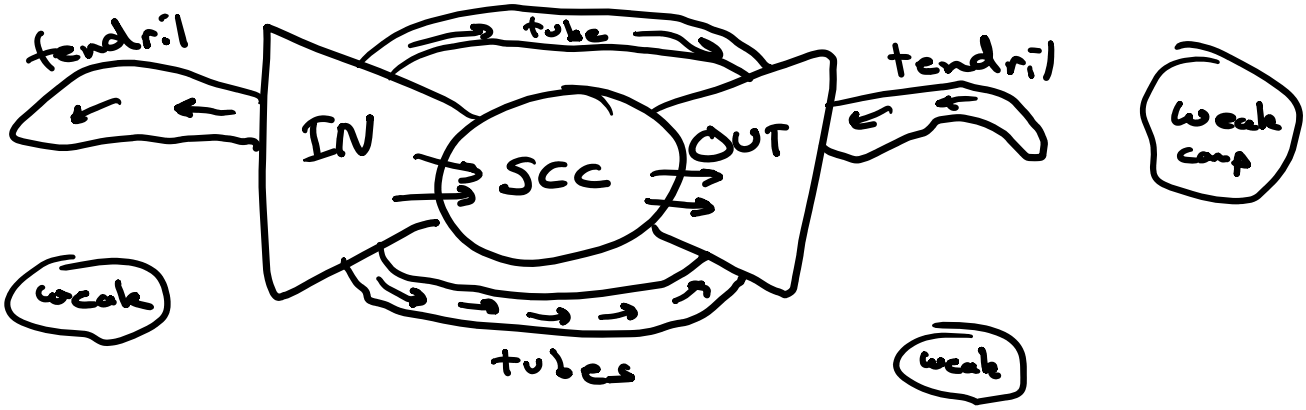
weak/strong connectivity

Results:

- Not weakly connected
  - ↪ many small components
- One "massive" strong and weak components
- Defined sets relative to these components: (all part of weak comp.)

SCC, IN, OUT, tendril, tubes

- SCC: massive strong comp.
- IN: vertices can reach SCC
- OUT: reached by SCC
- tendril: everything else
- tubes: can reach OUT, can be reached by IN



How to perform a decomposition such as this?

1. Find weak components trivially
2. SCC → do a strong components decomposition, find largest
3. OUT → search from SCC vertices (BFS with all SCC as roots)
4. IN → search back from SCC (BFS following reverse edges with SCC as roots)
5. Tendrils & tubes

↳ exercise for the reader