# LECTURE #6

CS 170

Spring 2021

## Last time:

Started unit on graph algorithms.

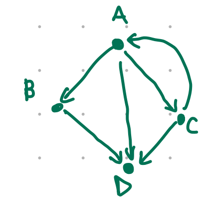
Depth-First Search (with pre and post values)

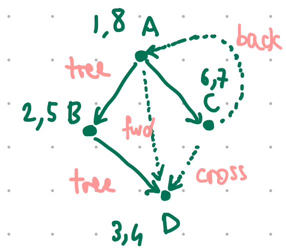
Finding the connected components of an undirected graph

Determining if a graph is acyclic

- tree edge $[_u \;\; [_v \;\; ]_v \;\; ]_u$
  part of DFS forest
- forward edge same as above
  to non-child descendant
- back edge $[_v \;\; [_u \;\; ]_u \;\; ]_v$
  to ancestor
- cross edge $[_v \;\; ]_v \;\; [_u \;\; ]_u$
  to already post-visited

This is impossible: $[_u \;\; ]_u \;\; [_v \;\; ]_v$




## Today:

Topological sort

Finding the strongly connected components of a directed graph.

Breadth First Search for shortest paths with unit distance

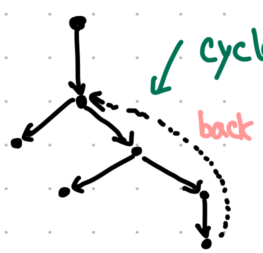
# Directed Acyclic Graphs

Here acyclic means without cycles.

They are useful to model causalities, hierarchies, temporal dependencies, ...

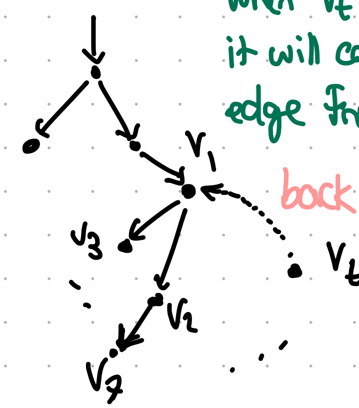
<u>claim:</u> $G$ is acyclic $\Leftrightarrow$ no back edges in DFS$(G)$

<u>proof:</u> ① back edge $\rightarrow$ cycle




② cycle $\rightarrow$ back edge

Say $G$ has cycle $v_1, \dots, v_t$, and WLOG $v_1$ is visited first when running DFS$(G)$. Then:




When $v_t$ is explored, it will cause a back edge from $v_t$ to $v_1$.

■

The claim directly leads to the following algorithm:

**IsDAG(G):**
1. Run DFS$(G)$ to collect pre, post numbers.
2. For each $(u,v) \in E$, if $(u,v)$ is a back edge then output NO.
   post[v] > post[u]
3. Output YES.

The running time is $O(|V|+|E|)$.

# Topological Sort

A topological sort of a DAG $G$ is a total order on vertices so that each edge goes from an earlier vertex to a later one.

Q: how to topologically sort a DAG?

claim: If $G$ is a DAG then $\forall (u,v) \in E$ in DFS($G$) it holds that post[$u$] > post[$v$].

proof: If $\exists\ (u,v) \in E$ s.t. post[$v$] > post[$u$] (i.e. $(u,v)$ is a back edge) then $G$ has a cycle. ∎

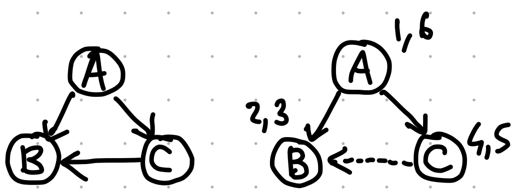
This leads to the following algorithm:

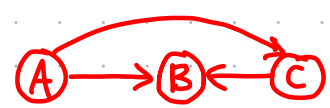
**TopoSort($G$)**: 1. Run DFS($G$) to collect pre, post numbers.
2. Output vertices in descending post order.

} can do in time $O(|V|+|E|)$ because can "push out" a vertex when we are done exploring it

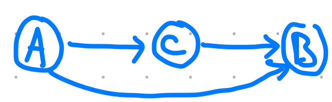
This works because $(u,v) \in E \to$ post[$u$] > post[$v$].

Note: increasing pre-order does NOT work




increasing pre-order: 

×

decreasing post-order: 

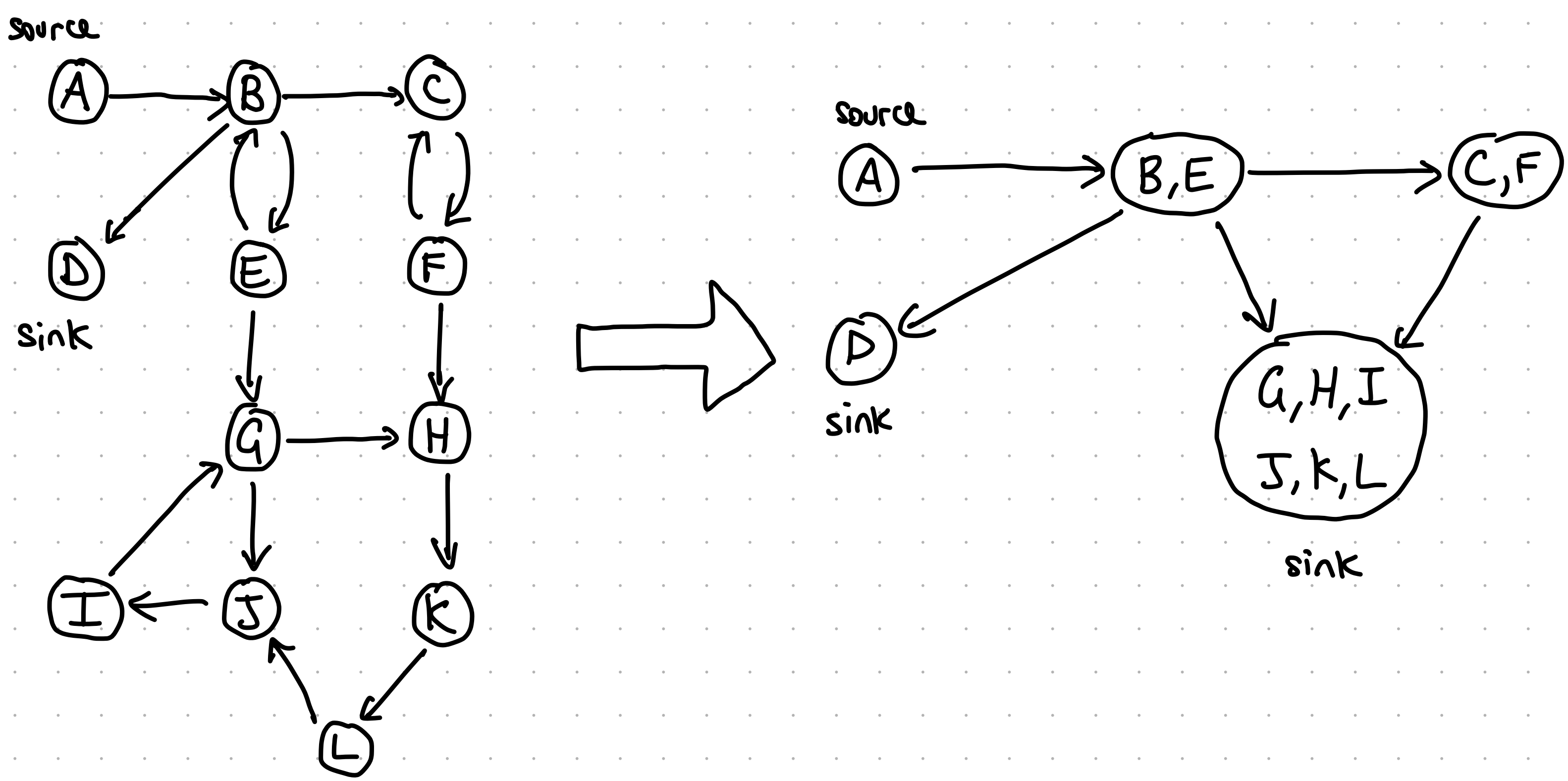
✓

## Connectivity [directed case]

u,v are strongly connected = path from u to v & (possibly diff) path from v to u.

This equivalence relation partitions G into strongly connected components (SCCs).

**Every graph is a DAG of SCCs.**



# Finding SCCs

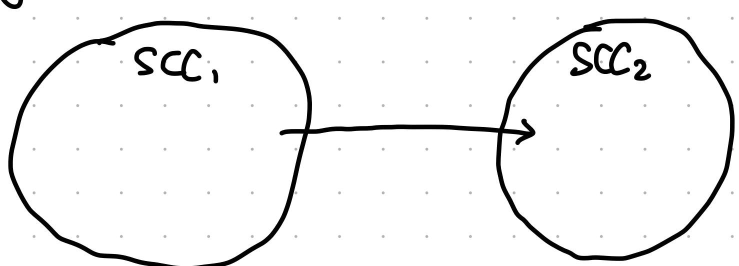
**Idea #1:** explore $(G,v)$ visits all and only vertices ~~in SCC of v~~ reachable from v

**Idea #2:** find a vertex in sink-SCC, explore vertices there and remove them; repeat

Q: how to find sink-SCC?

Finding a vertex in source-SCC is easy: node with highest post number.

Why?




max post in $SCC_1$ > max post in $SCC_2$

Can linearize SCCs by descending max post numbers.

Then note that $v \in$ sink-SCC of $G$

$\Leftrightarrow v \in$ source-SCC of $G^R$.

**FindSCC** $(G)$:
1. Deduce $G^R$ from $G$. (linear time from G's matrix or list representation)
2. Run DFS$(G^R)$ to get post numbers.
3. Initialize $scc := 1$ and $\forall v \in V$ sccnum[v] := null.
4. For each $v$ in $V$ in $\underbrace{\text{reverse post-order}}_{\text{naturally from the stack}}$ of $G^R$:

```
if not visited[v]
    explore(G,v)   [assign sccnum[v] := scc inside explore(G,v)]
    scc++
```

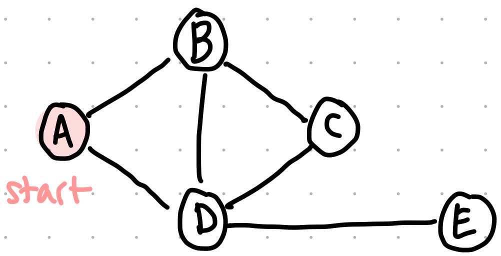
# Paths in Graphs

We have seen that DFS tells us about reachability in a grap (DAG of SCCs), but gives no guarantees about whether getting there is long/short.

NEW GOAL: shortest paths

Given $v \in V$, find distance (& path) from $v$ to all other vertices.

Example:




| | A | B | C | D | E |
|---|---|---|---|---|---|
| dist(A, ·) | 0 | 1 | 2 | 1 | 2 |

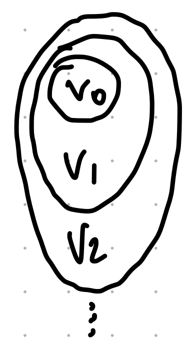
Observe that

$V_0 = \{A\}$ = all vertices at distance 0

$V_1 = \{B, D\}$ = all vertices at distance 1

$V_2 = \{C, E\}$ = all vertices at distance 2

$\vdots$

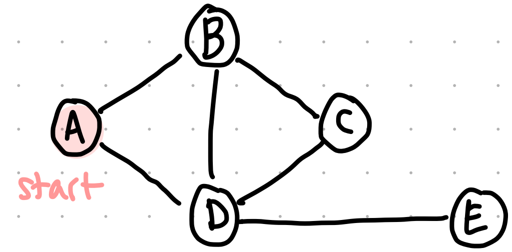



Idea for algorithm: $V_{i+1} :=$ "neighbors of $V_i$ in $V \setminus (V_0 \cup V_1 \cup \dots \cup V_i)$"

so we should design an algorithm that given $V_0, V_1, \dots, V_i$ finds $V_{i+1}$

# Breadth-First Search (BFS)

Initialize a queue (FIFO) with starting vertex.
At each iteration, eject a node and add back all unseen nodes with distances +1.

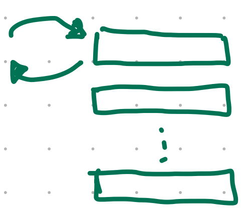



**BFS**$(G,s)$:

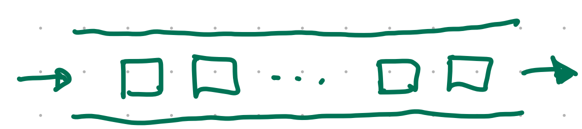
```
1. dist[s] := 0                // starting vertex
   dist[V \ {s}] := ∞          // all other verts are unseen
   Q := InitQueue({s})
2. while Q ≠ {}
      u := eject(Q)
      for (u,v) ∈ E :
         if dist[v] = ∞        // not seen yet
            inject(Q,v)
            dist[v] := dist[u]+1
```

| Q | A | B | C | D | E |
|---|---|---|---|---|---|
| $[\overset{0}{A}]$ | 0 | ∞ | ∞ | ∞ | ∞ |
| $[\overset{1}{D}\ \overset{1}{B}]$ | 0 | 1 | ∞ | 1 | ∞ |
| $[\overset{2}{C}\ \overset{1}{D}]$ | 0 | 1 | 2 | 1 | ∞ |
| $[\overset{2}{E}\ \overset{2}{C}]$ | 0 | 1 | 2 | 1 | 2 |
| $[\overset{2}{E}]$ | | | // | | |
| $[\ ]$ | | | // | | |

- In DFS we explore via stack (FILO):



- In BFS we explore via queue (FIFO):



(can realize via a linked list)

# Analysis of BFS

- **Running time:**

  Initialization preamble is $O(|V|)$.

  Each vertex is injected and ejected exactly once. This adds up to $|V|$ injects + $|V|$ ejects.

  Each directed edge is examined once. This adds up to $O(|E|)$ work.

  Total time is $O(|V|+|E|)$ (like DFS).

- **Correctness:**


Initially: $Q$ contains exactly $V_0 = \{s\}$

Later: for $d = 1, 2, 3, \ldots$ there is a point at which $Q$ contains exactly $V_d$.

At that time: (1) all nodes of distance $\leq d$ have correct dist[·]

(2) all other nodes have dist[ ] $= \infty$

(3) queue contains only nodes at dist[ ] $= d$.

## Lengths on Edges

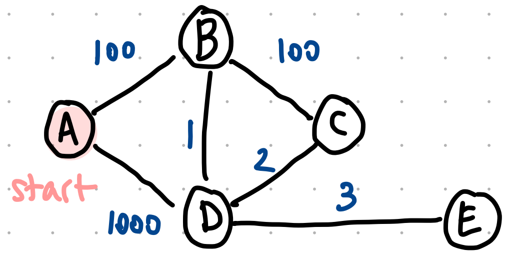
So far: all edges have the same length.

We now introduce a label for each edge that denotes its length $\ell : E \to \mathbb{N}$.

These lengths do not have to be physical (could be money, time, strength, ...).

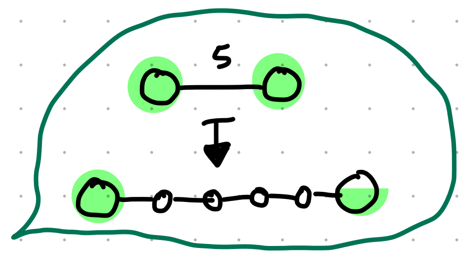
Q: how to solve the shortest path problem with edge lengths?

**Idea #1: use BFS**




This does not make much sense.

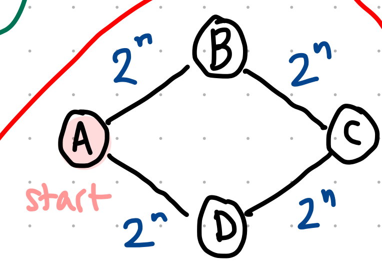
**Idea #2: recycle BFS**




1. Transform $G$ into $G'$ by adding dummy nodes.
2. Run BFS on $G'$ rather than $G$.

The approach is correct but running time is $O(|V'|+|E'|)$.

This is problematic because $|V'|, |E'|$ may be exponential in input size!




input size is $O(n)$
but running time is $\exp(n)$

**Idea #3: recycle BFS in a better way** — **Dijkstra's Algorithm**

It uses a priority queue to consider nodes in an order that follows "best distance so far".