CS 3100
Data Structures and Algorithms 2
Lecture 5: Topological Sort, Connected Components

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

Readings in CLRS 4th edition:

- Chapter 20: Sections 20-3, 20-4, and 20-5

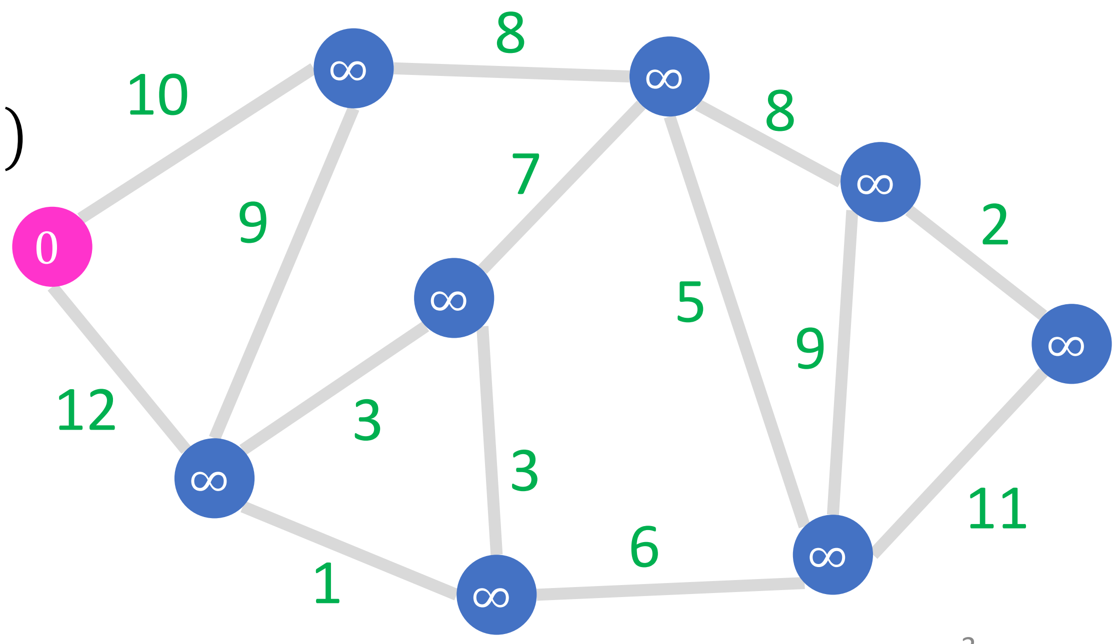
Announcements

- PS2 due tomorrow
- PA1 due Friday
- Office hours
  - Prof Hott Office Hours: Mondays 11a-12p, Fridays 10-11a and 2-3p
  - Prof Pettit Office Hours: Mondays and Wednesdays 2:30-4:00p
  - TA office hours posted on our website

Dijkstra's Algorithm Implementation

Implementation:

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v,u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, d_v + w(v,u))$
            $u.\mathrm{parent} = v$



Dijkstra’s Algorithm Proof Strategy

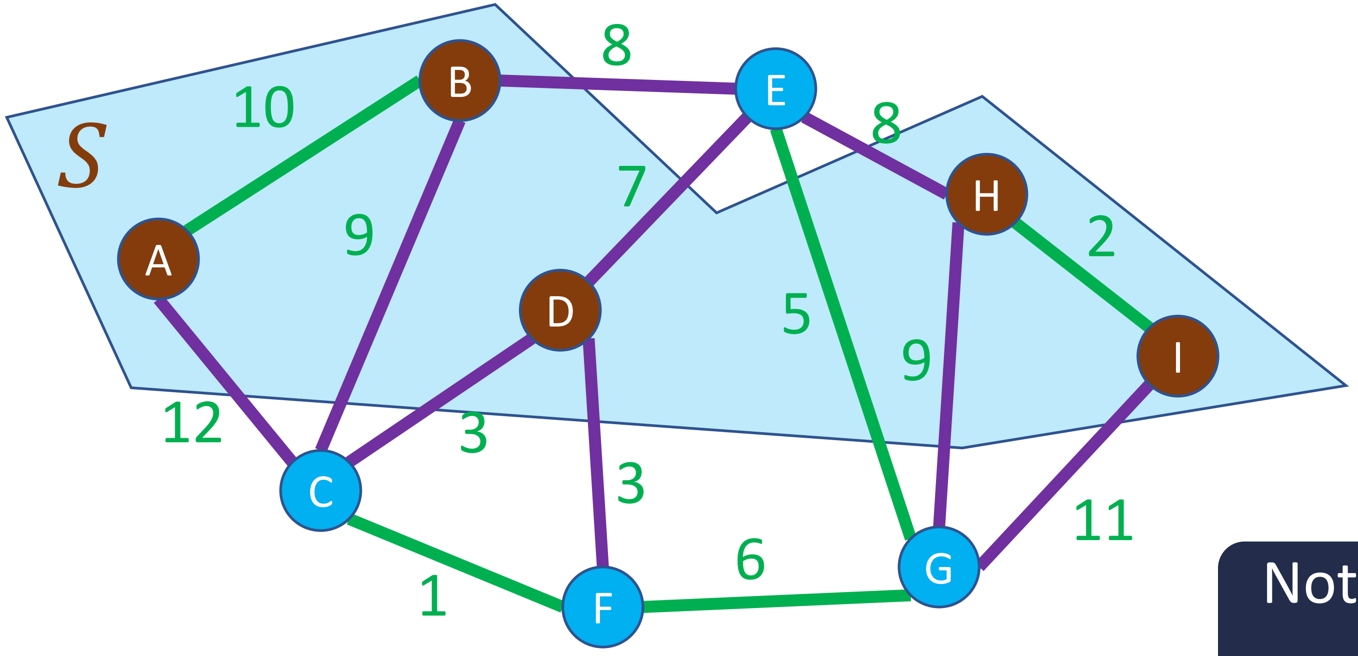
Proof by induction

Proof Idea: we will show that when node u is removed from the priority queue, $d_u = \delta(s, u)$ where $\delta(s, u)$ is the shortest distance

- **Claim 1:** There is a path of length d_u (as long as $d_u < \infty$) from s to u in G
- **Claim 2:** For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$

Graph Cuts

A **cut** of a graph $G = (V, E)$ is a partition of the nodes into two sets, S and $V - S$



Notion extends naturally to a set of edges

An edge $(v_1, v_2) \in E$ crosses a cut if $v_1 \in S$ and $v_2 \in V - S$

An edge $(v_1, v_2) \in E$ respects a cut if $v_1, v_2 \in S$ or if $v_1, v_2 \in V - S$

Correctness of Dijkstra's Algorithm

Inductive hypothesis: Suppose that nodes $v_1 = s, \ldots, v_i$ have been removed from PQ, and for each of them $d_{v_i} = \delta(s, v_i)$, and there is a path from s to v_i with distance d_{v_i} (whenever $d_{v_i} < \infty$)

Base case:

- $i = 0$: $v_1 = s$
- Claim holds trivially

Correctness of Dijkstra’s Algorithm: Claim 1

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 1: There is a path of length d_u (as long as $d_u < \infty$) from s to u in G

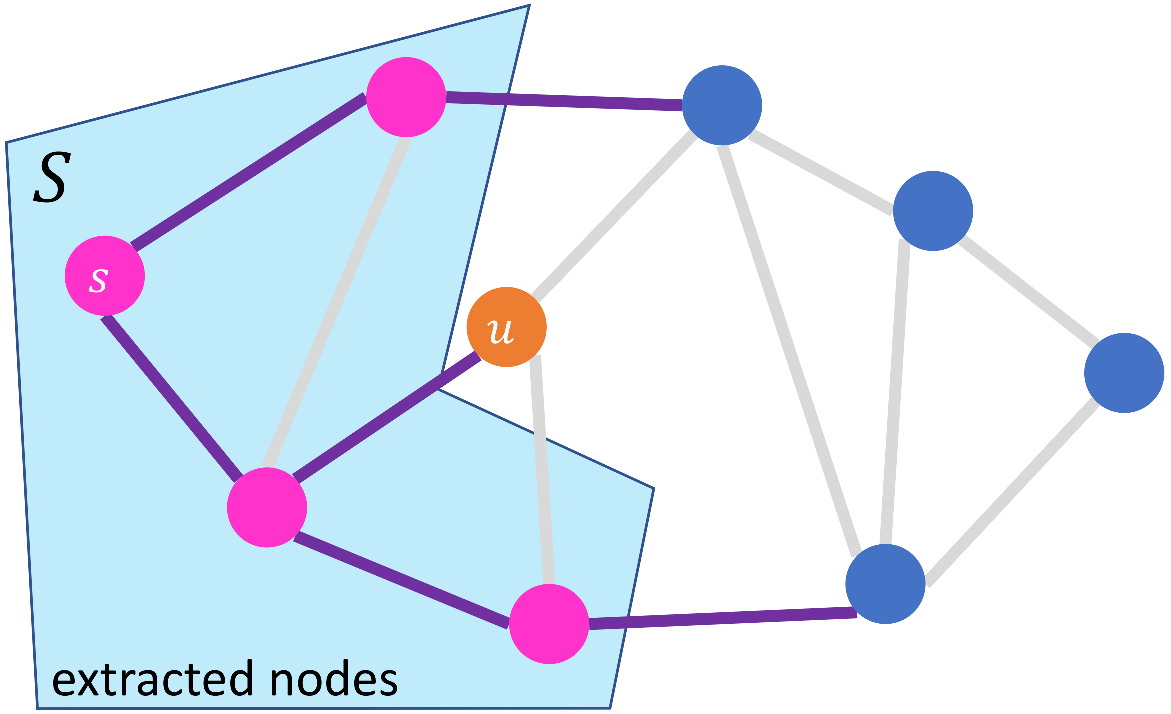
Proof:

- Suppose $d_u < \infty$
- This means that PQ.decreaseKey was invoked on node u on an earlier iteration
- Consider the last time PQ.decreaseKey is invoked on node u
- PQ.decreaseKey is only invoked when there exists an edge $(v,u) \in E$ and node v was extracted from PQ in a previous iteration
- In this case, $d_u = d_v + w(v,u)$
- By the inductive hypothesis, there is a path $s \to v$ of length d_v in G and since there is an edge $(v,u) \in E$, there is a path $s \to u$ of length d_u in G

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \ldots, u)$, $w(s, \ldots, u) \geq d_u$

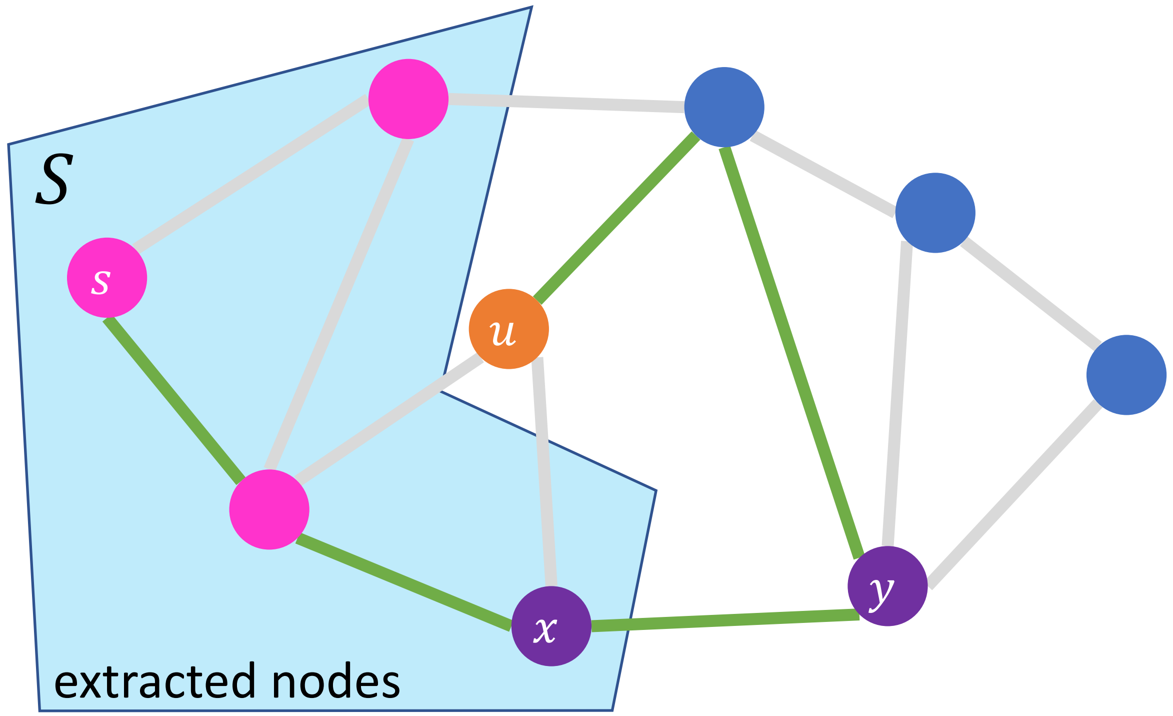


Extracted nodes "cuts" G into two subsets, $(S, V - S)$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{st}$ node extracted

Claim 2: For every path $(s, \ldots, u)$, $w(s, \ldots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \ldots, u)$

Since $u \notin S$, $(s, \ldots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

$$w(s, \ldots, u) \geq \delta(s, x) + w(x, y) + w(y, \ldots, u)$$

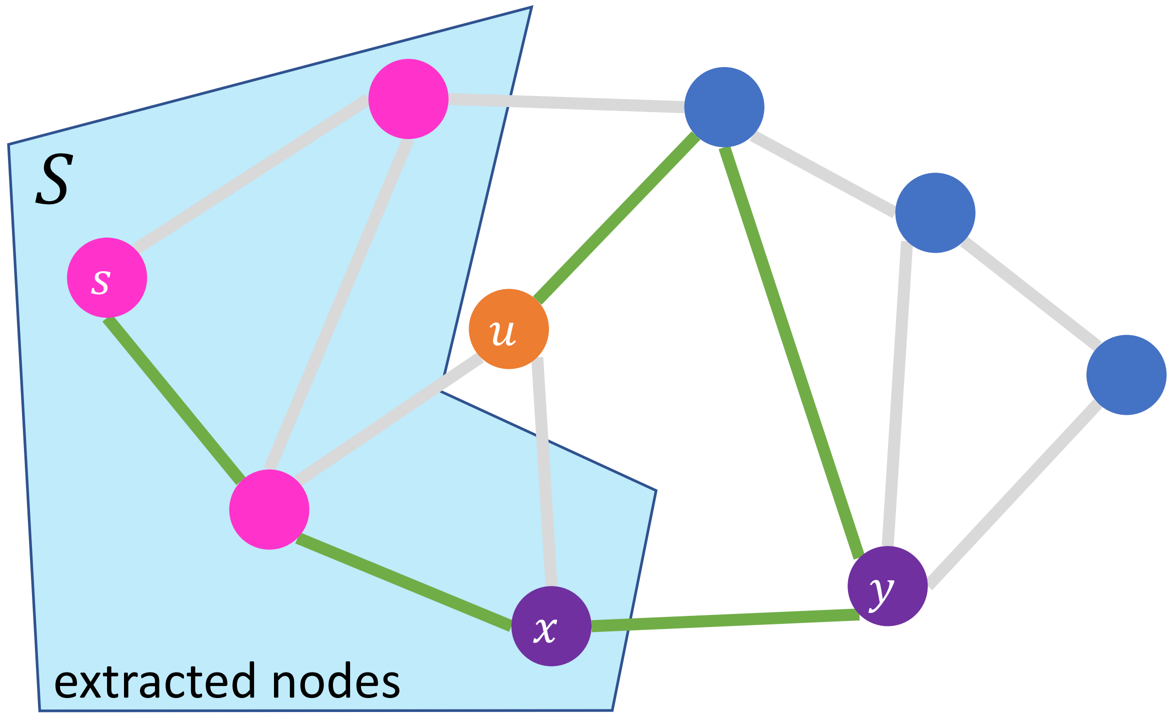
$w(s, \ldots, u) = w(s, \ldots, x) + w(x, y) + w(y, \ldots, u)$

$w(s, \ldots, x) \geq \delta(s, x)$ since $\delta(s, x)$ is weight of shortest path from s to x

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

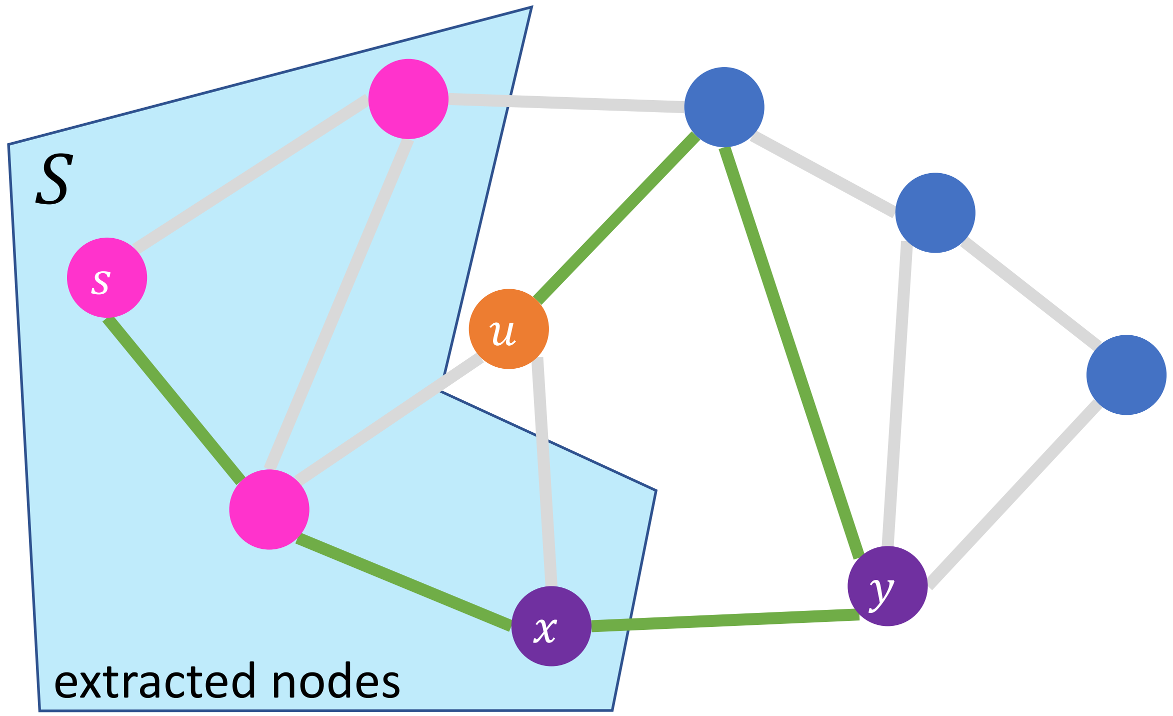
$$\begin{aligned} w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\ &= d_x + w(x, y) + w(y, \dots, u) \end{aligned}$$

Inductive hypothesis: since x was extracted before, $d_x = \delta(s, x)$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

$$\begin{aligned} w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\ &= d_x + w(x, y) + w(y, \dots, u) \\ &\geq d_y + w(y, \dots, u) \end{aligned}$$

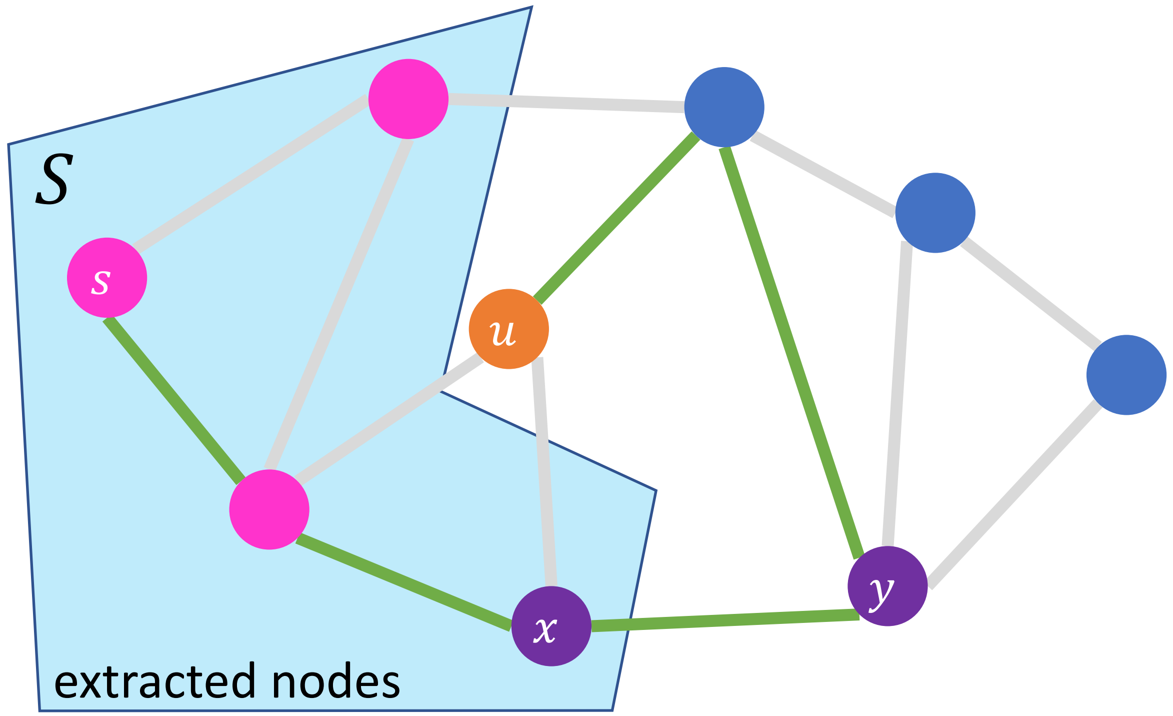
By construction of Dijkstra's algorithm, when x is extracted, d_y is updated to satisfy

$$d_y \leq d_x + w(x, y)$$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

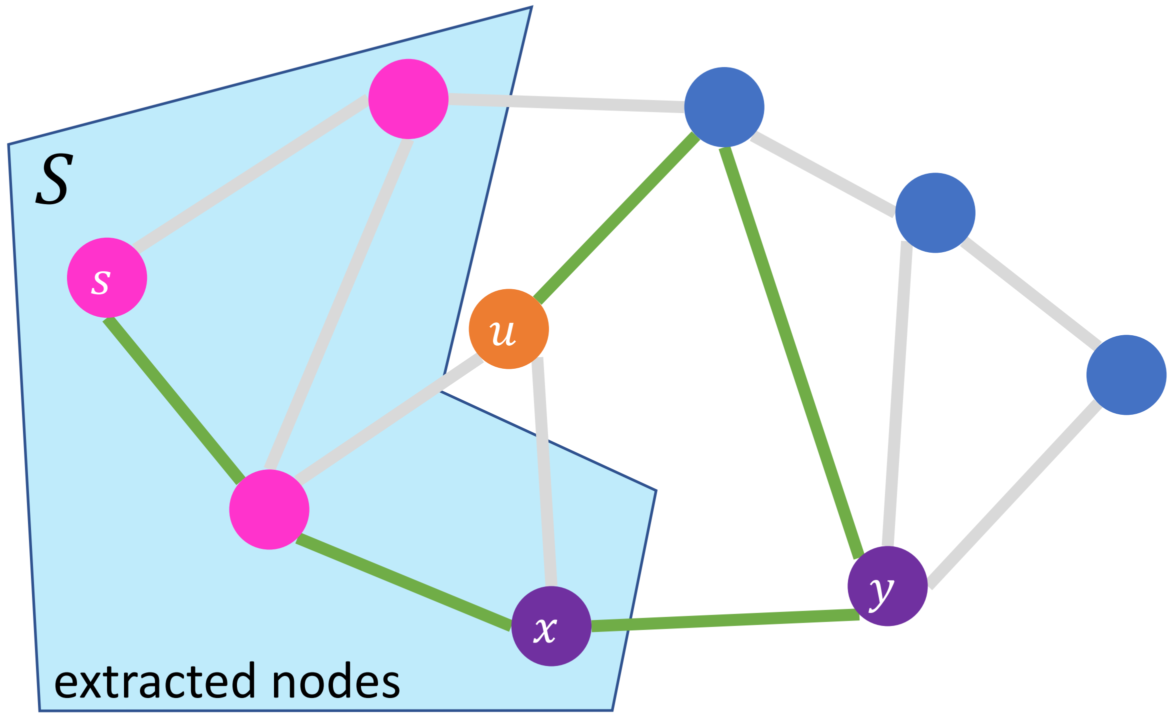
$$
\begin{aligned}
w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\
&= d_x + w(x, y) + w(y, \dots, u) \\
&\geq d_y + w(y, \dots, u) \\
&\geq d_u + w(y, \dots, u)
\end{aligned}
$$

Greedy choice property: we always extract the node of minimal distance so $d_u \leq d_y$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes "cuts" G into $(S, V - S)$

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

$$
\begin{aligned}
w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\
&= d_x + w(x, y) + w(y, \dots, u) \\
&\geq d_y + w(y, \dots, u) \\
&\geq d_u + w(y, \dots, u) \\
&\geq d_u
\end{aligned}
$$

All edge weights assumed to be positive

Correctness of Dijkstra's Algorithm

Conclusion: We used proof by induction to show:

When node u is removed from the priority queue, $d_u = \delta(s, u)$

- **Claim 1:** There is a path of length d_u (as long as $d_u < \infty$) from s to u in G
- **Claim 2:** For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$

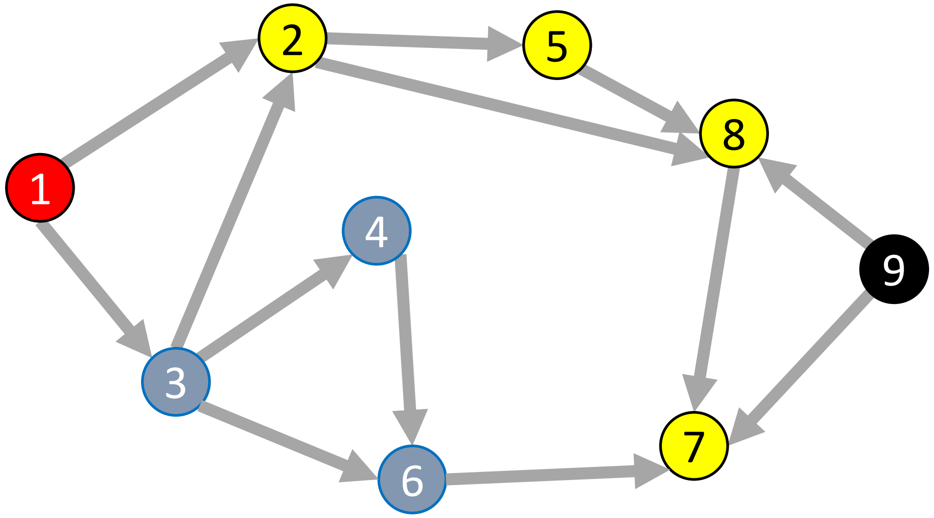
In other words, all paths $(s, \dots, u)$ are no shorter than d_u

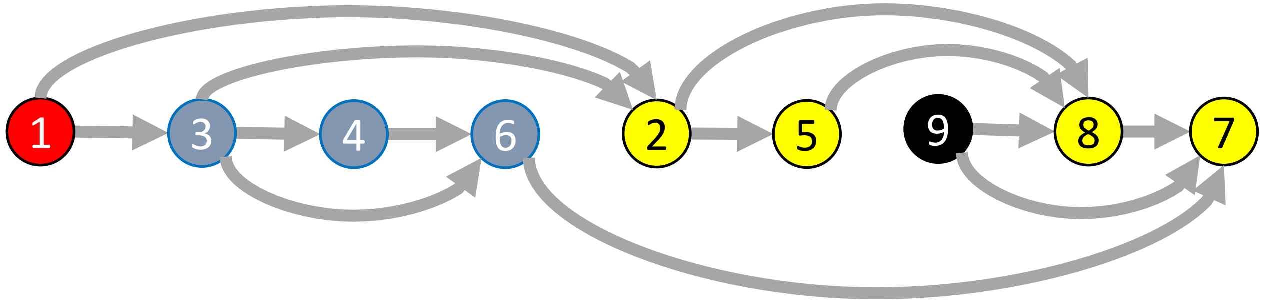
which makes it the shortest path (or one of equally shortest paths).

Topological Sort

Topological Sort

A Topological Sort of a **directed acyclic graph** $\boldsymbol{G = (V, E)}$ is a permutation of V such that if $(u, v) \in E$ then u is before v in the permutation

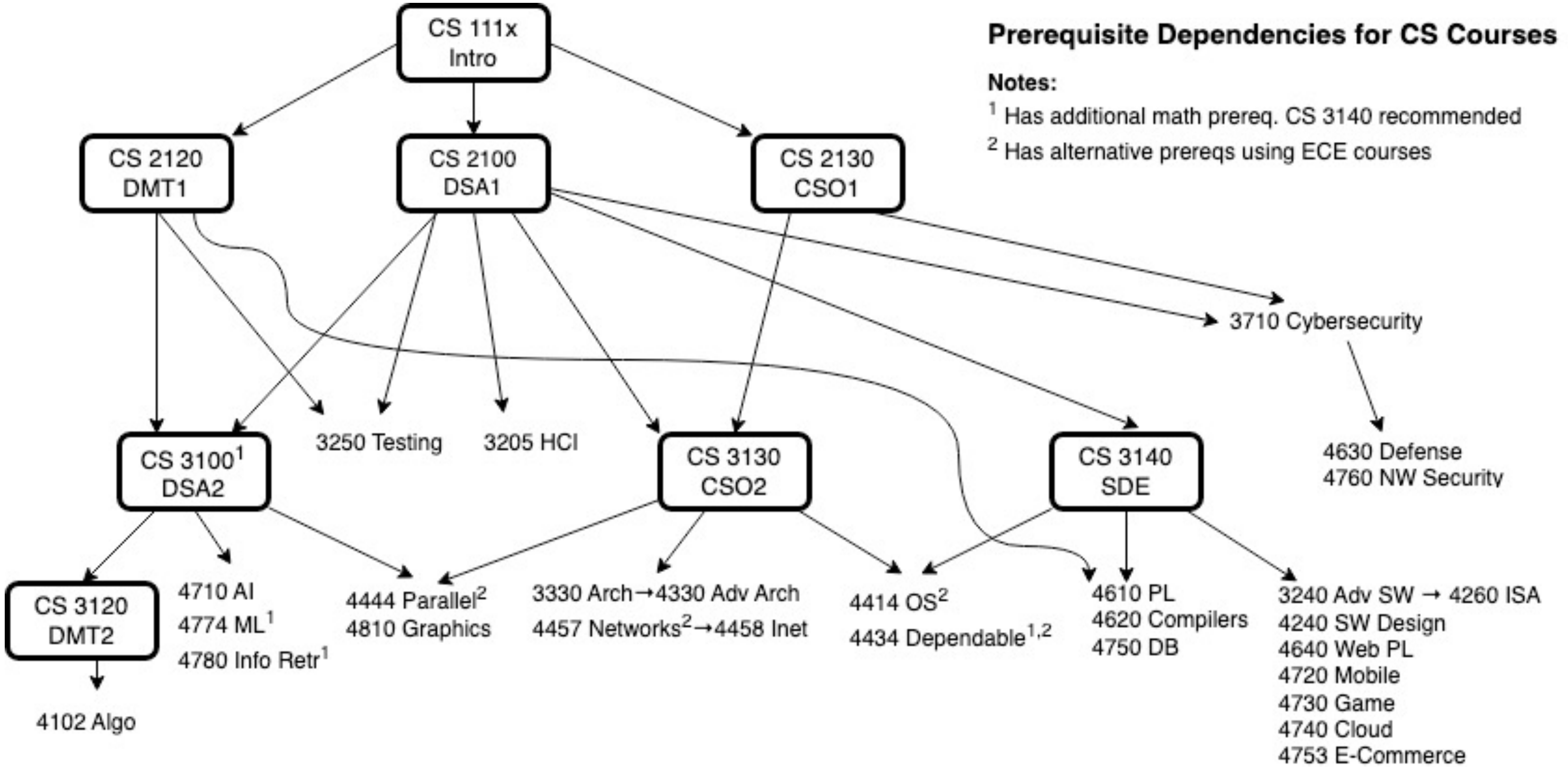




Topological Sort

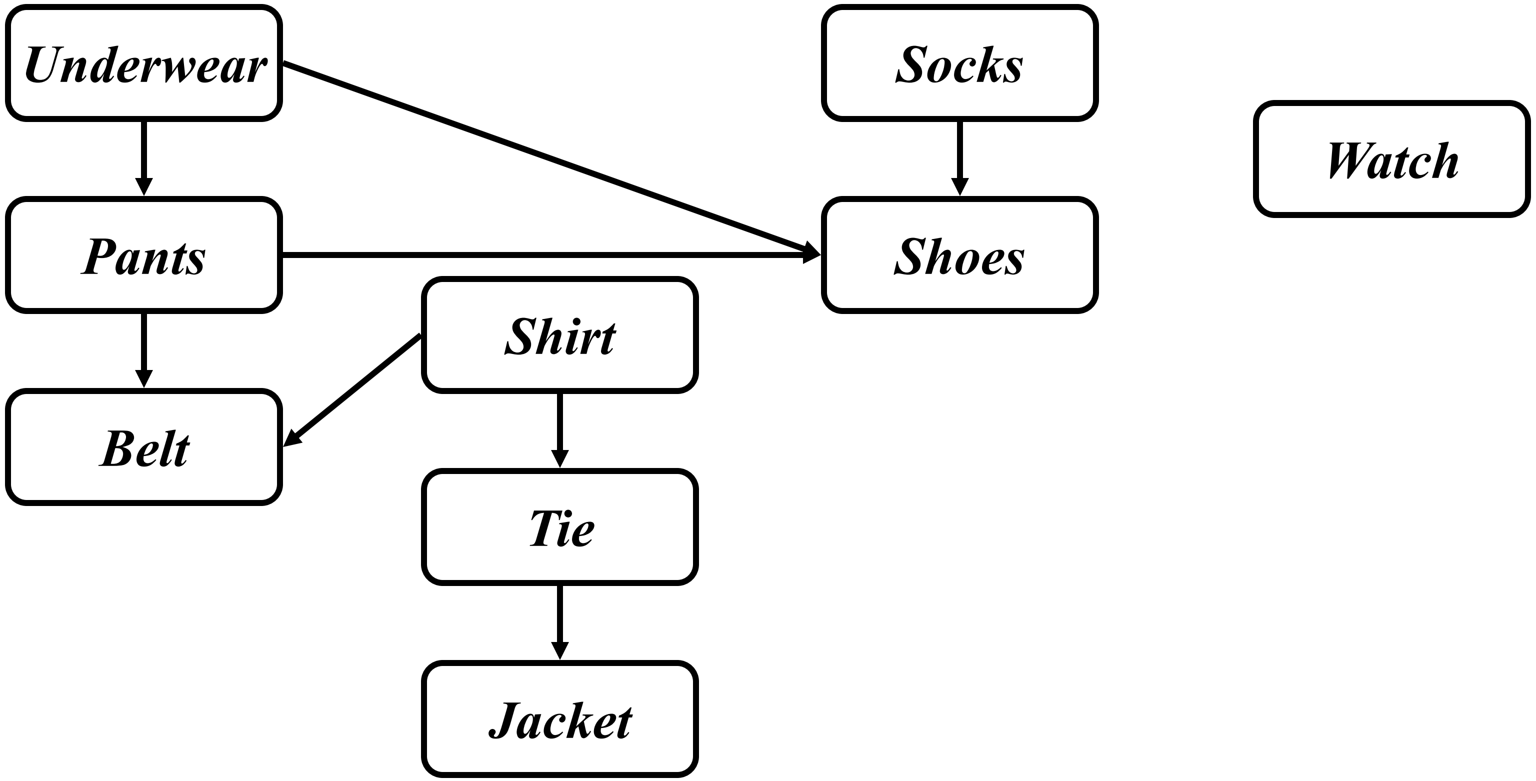
What are allowable orderings I can take all these CS classes?

- Note there are many possible orderings
- Unlike sorting a list

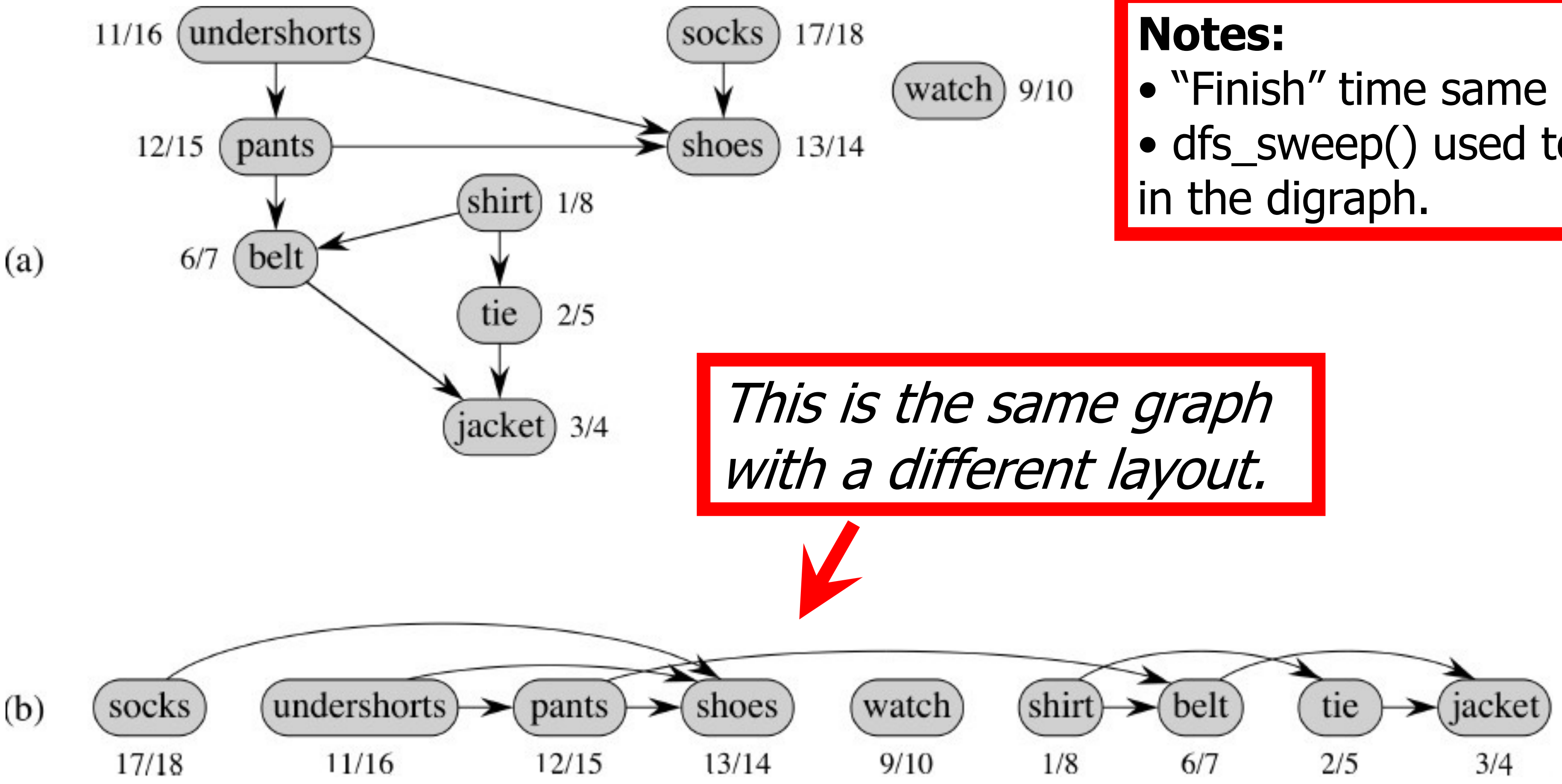


Topological Sort

Getting dressed



We Can Use DFS and Finish Times



Notes:
- "Finish" time same as "done" time.
- dfs_sweep() used to visit all nodes in the digraph.

This is the same graph with a different layout.

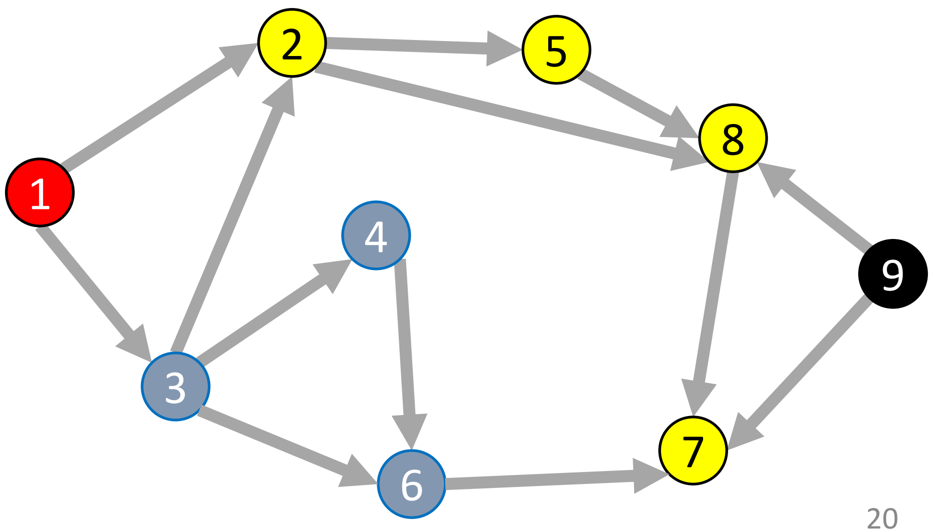
Topologically sorted vertices appear in reverse order of their finish times!

DFS: Topological sort

```
def dfs(graph, s):
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        dfs_rec(graph, s, seen, done)

def dfs_rec(graph, curr, seen, done):
        mark curr as seen
        for v in neighbors(current):
                if v not seen:
                        dfs_rec(graph, v, seen, done)
        mark curr as done
```

Idea: List in reverse order by finish time

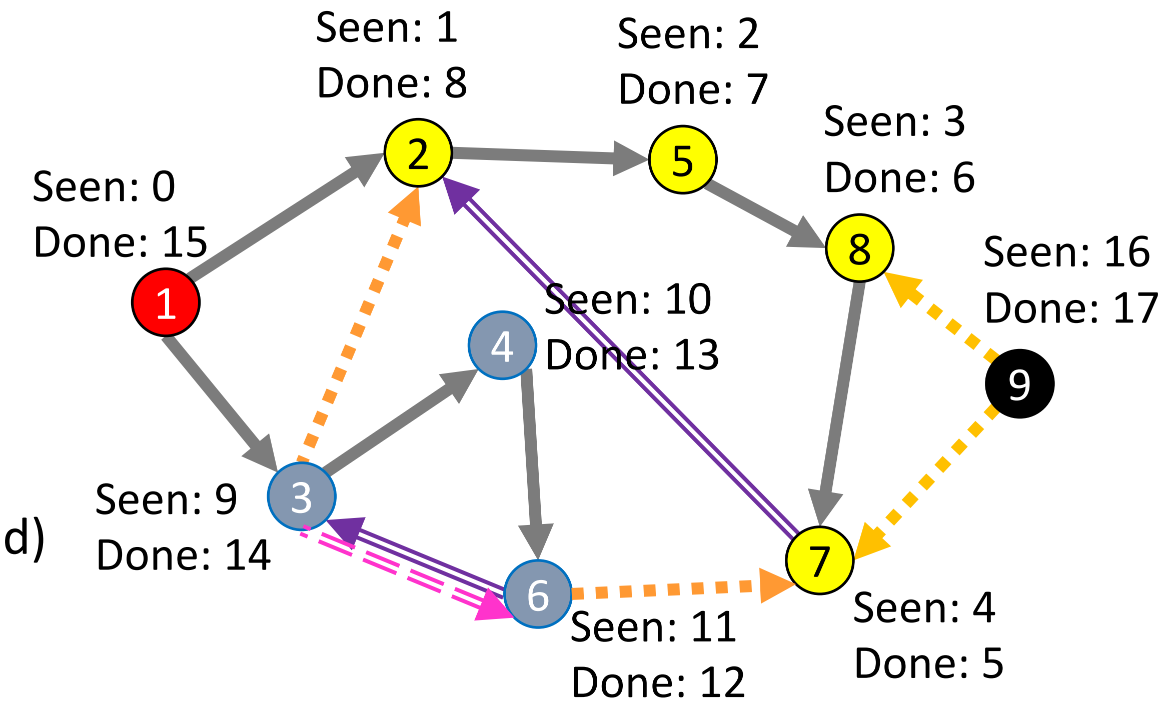


DFS: Topological sort

```
def top_sort(graph):  # has loop like dfs_sweep
        seen = [False, False, False, …] # length matches |V|
        finished = []
        for s in graph:
            if s not seen:
                finish_time(graph, s, seen, finished)
        return reverse(finished)

def finish_time(graph, curr, seen, finished):
        seen[curr] = True
        for v in neighbors(current):
                if v not seen:
                        finish_time(graph, v, seen, finished)
        finished.append(curr)
```

Idea: List in reverse order by done/finish time



Strongly Connected Components

Readings: CLRS 20.5, but you can ignore the proof-y parts

Strongly Connected Components (SCCs)

In a digraph, Strongly Connected Components (SCCs) are subgraphs where all vertices in each SCC are reachable from one another

- Thus vertices in an SCC are on a directed cycle
- Any vertex not on a directed cycle is an SCC all by itself

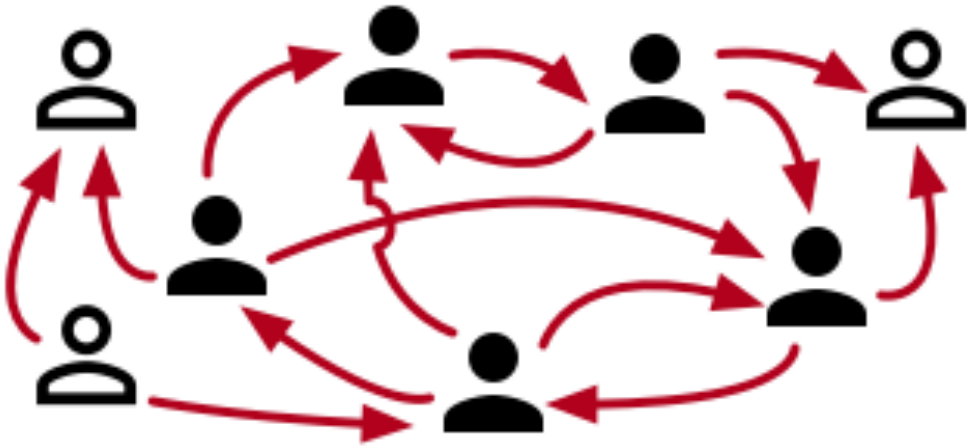
Common need: decompose a digraph into its SCCs

- Perhaps then operate on each, combine results based on connections between SCCs

Real-world Example: Social Networks

Model a social network of users

- Directed edge *u->v* means *u* follows *v*



We want to identify a group of users who follow each other

- Maybe not directly
- OK if it's indirect, i.e. if there's a path connecting any pair in the group

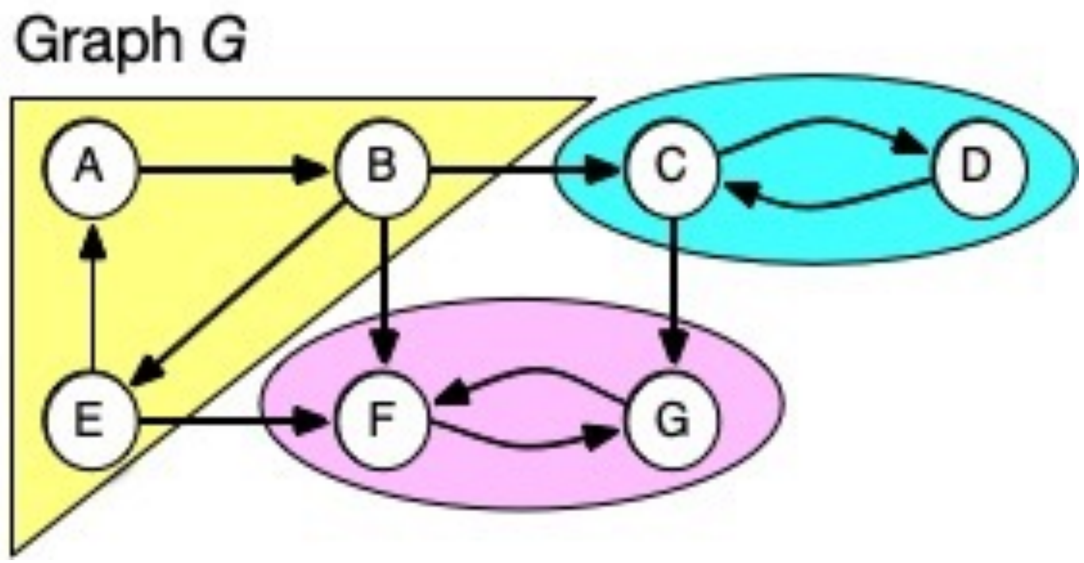
In this example, the group of solid-colored users is an SCC

Note: if all pairs had to follow each other, we call this a *clique*

SCC Example

Example: digraph below has 3 SCCs

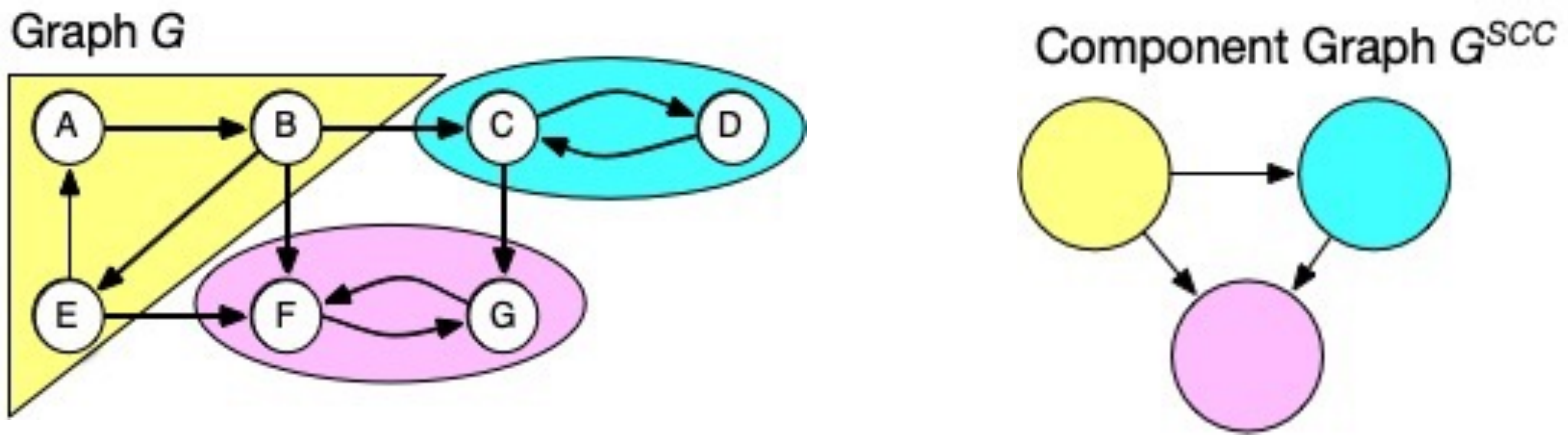
- Note here each SCC has a cycle. (Possible to have a single-node SCC.)
- Note connections to other SCCs, but no path leaves a SCC and comes back
- Note there's a unique set of SCCs for a given digraph



Component Graph

Sometimes for a problem it's useful to consider digraph G's **component graph,** G^{SCC}

- It's like we "collapse" each SCC into one node
- Might need a topological ordering between SCCs



How to Decompose Digraph into SCCs

Several algorithms do this using DFS

We'll use CLRS's choice (by Kosaraju and Sharir)

Algorithm works as follows:

1. Call *dfs_sweep(G)* to find finishing times *u.f* for each vertex *u* in *G*.
2. Compute G^T, the transpose of digraph *G*.

   (Reminder: transpose means same nodes, edges reversed.)
3. Call *dfs_sweep*(G^T) but do the recursive calls on nodes in the order of decreasing *u.f* from Step 1. (Start with the vertex with largest finish time in G's DFS tree,…)
4. The DFS forest produced in Step 3 is the set of SCCs

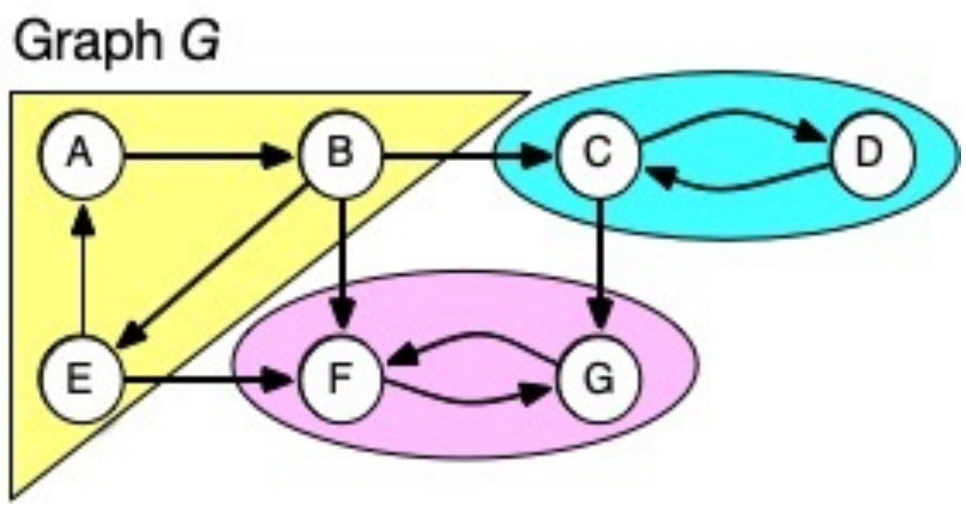
Why Do We Care about the Transpose?

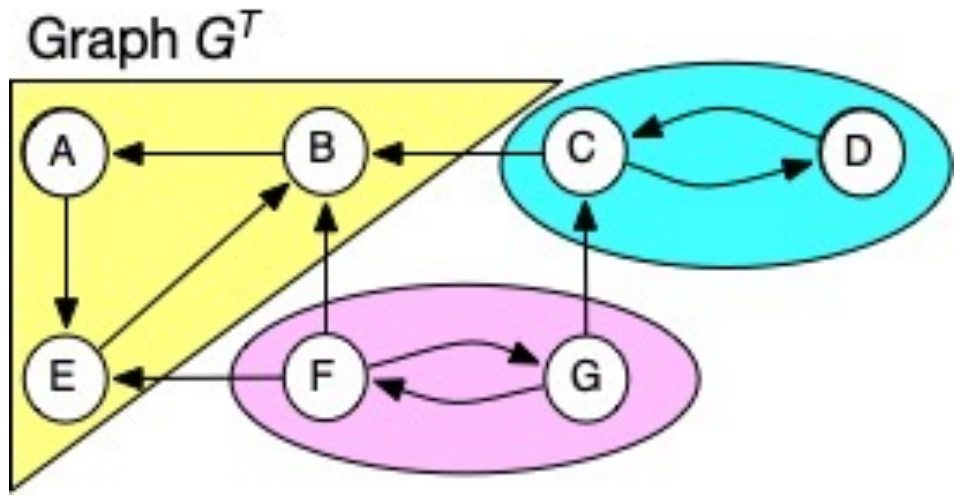
If we call DFS on a node in an SCC, it will visit all nodes in that SCC

- But it could leave the SCC and find other nodes ☹
- Could we prevent that somehow?

Note that a digraph and its transpose have the same SCCs

- Maybe we can use the fact that edge-directions are reversed in G^T to stop DFS from leaving an SCC?
- But this depends on the order you choose vertices to do *dfs_sweep()* in G^T



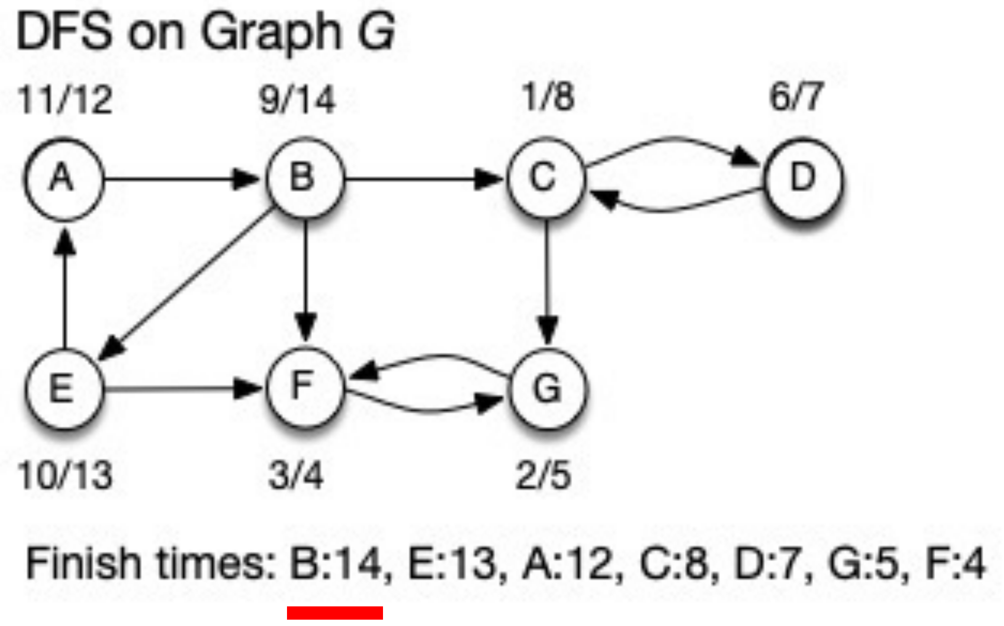


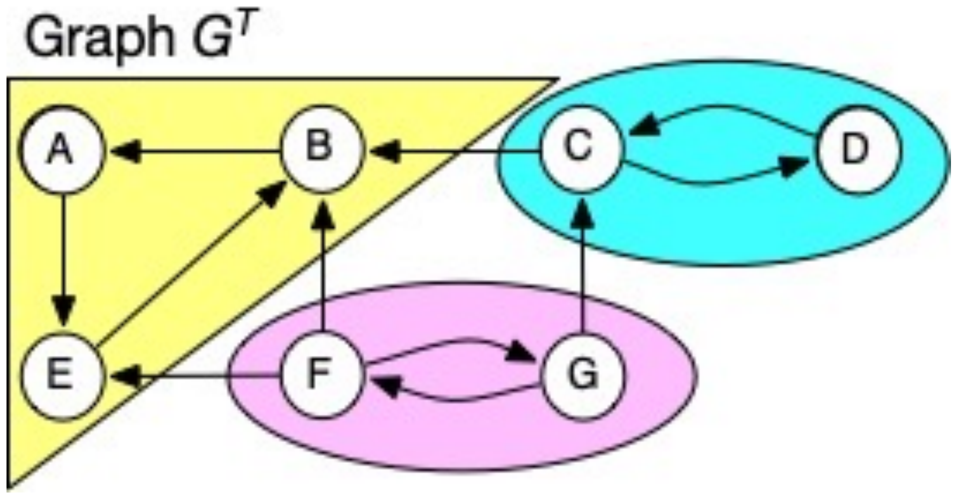
Why Do We Care About Finish Times?

Our algorithm first finds DFS finish times in G

Then calls recursive DFS on transpose G^T from vertex with largest finish time (here, B)

- Reversed edges in G^T stop it visiting nodes in other SCCs



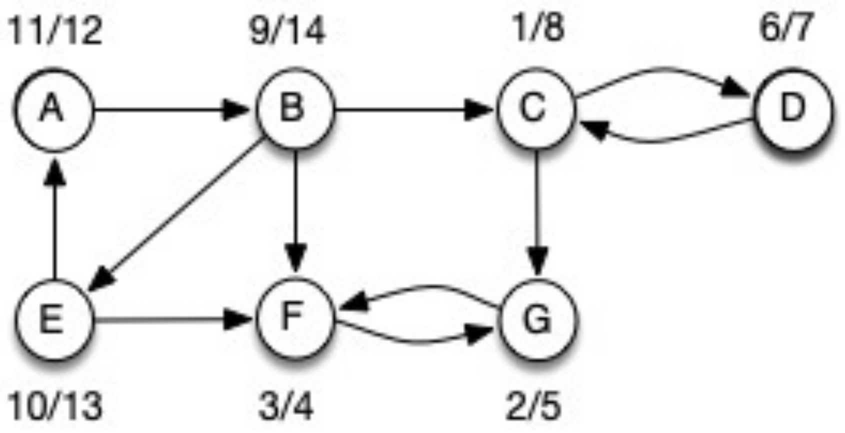


Why Do We Care About Finish Times?

After recursive DFS <u>on transpose G^T</u> finds SCC containing B, next DFS will start from C

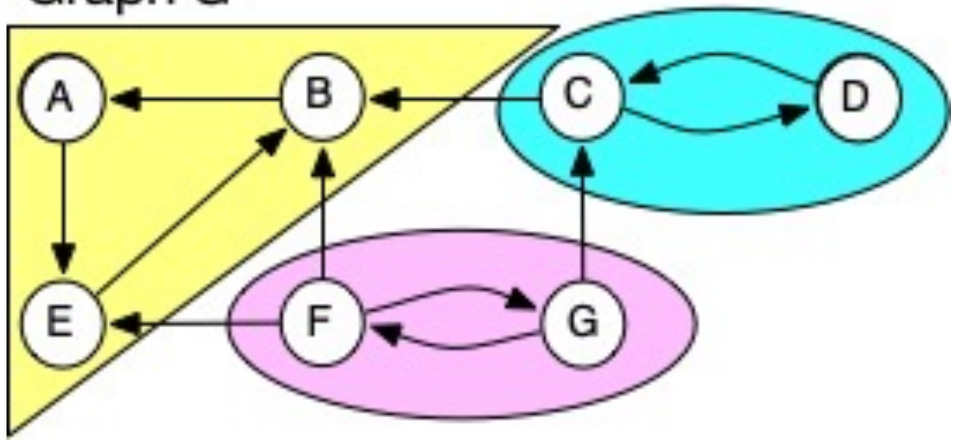
- Nodes in previously found SCC(s) have been visited
- Reversed edges in G^T stop it visiting nodes in SCCs yet to be found

DFS on Graph G



Finish times: B:14, E:13, A:12, C:8, D:7, G:5, F:4

Graph G^T

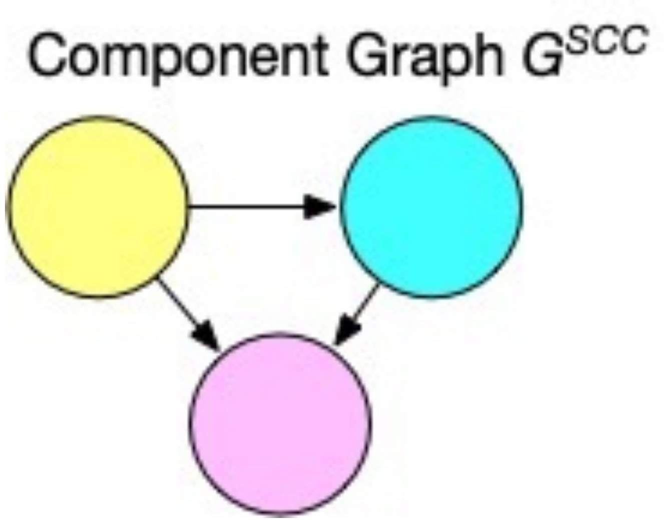


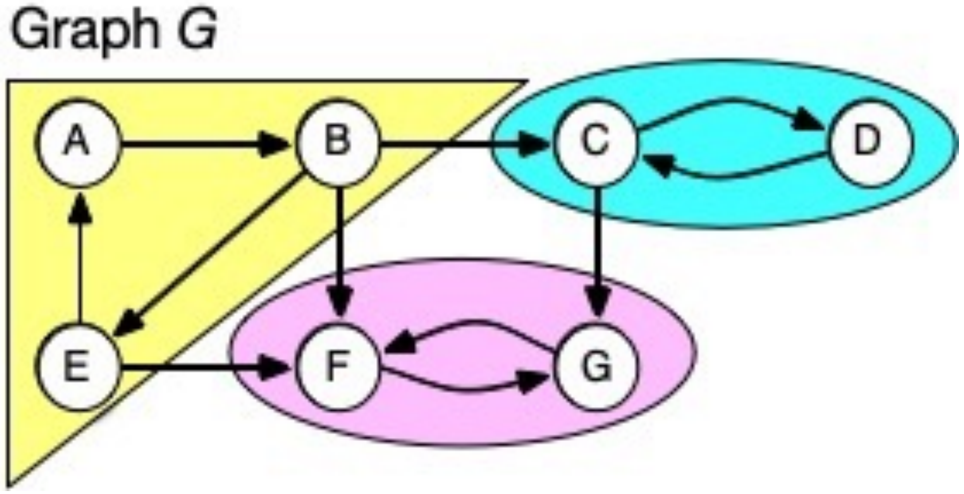
Ties to Topological Sorting

Formal proof of correctness in CLRS, but hopefully from previous slides you're convinced it works!

Note how the use of finish times makes this seem like topological sort. And it is, if you think of topological ordering for G^{SCC}

- Cycles in G, but no cycles in G^{SCC} so we could sort that
- Topological sort controls the order we do things, and DFS finds all the reachable nodes in an SCC





Final Thoughts

There are many interesting problems involving digraphs and DAGs

They can model real-world situations

- Dependencies, network flows, …

DFS is often a valuable strategy to tackle such problems

- For DAGs, not interested in back-edges, since DAGs are acyclic
- Ordering, reachability from DFS can be useful