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

Co-instructors: Robbie Hott and Tom Horton

Fall 2023

Readings in CLRS 4th edition:

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

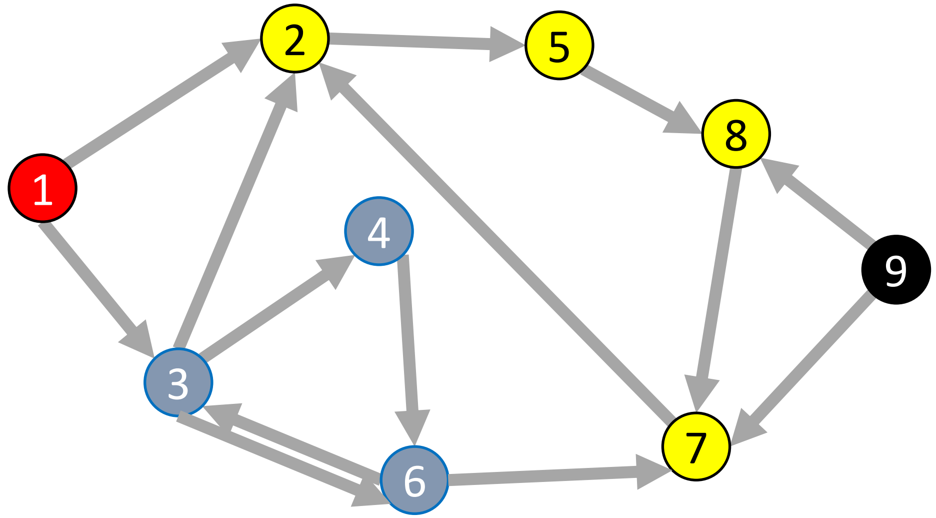
Announcements

- Upcoming dates
  - PS1 due ~~Sept 8 (Friday) at 11:59pm~~ Tuesday, Sept 12 at 11:59pm
  - PA1 due Sept 17 (Sunday) at 11:59pm
- Office Hours
  - Prof Hott: 3-5pm Monday, 4-5pm Thursday
  - Prof Horton: 2-3:30 Mon, 3:30-5 Tue, 2:30-4 Thu, 2-3 Fri
  - TA office hours posted online

DFS: Recursively

```
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
```

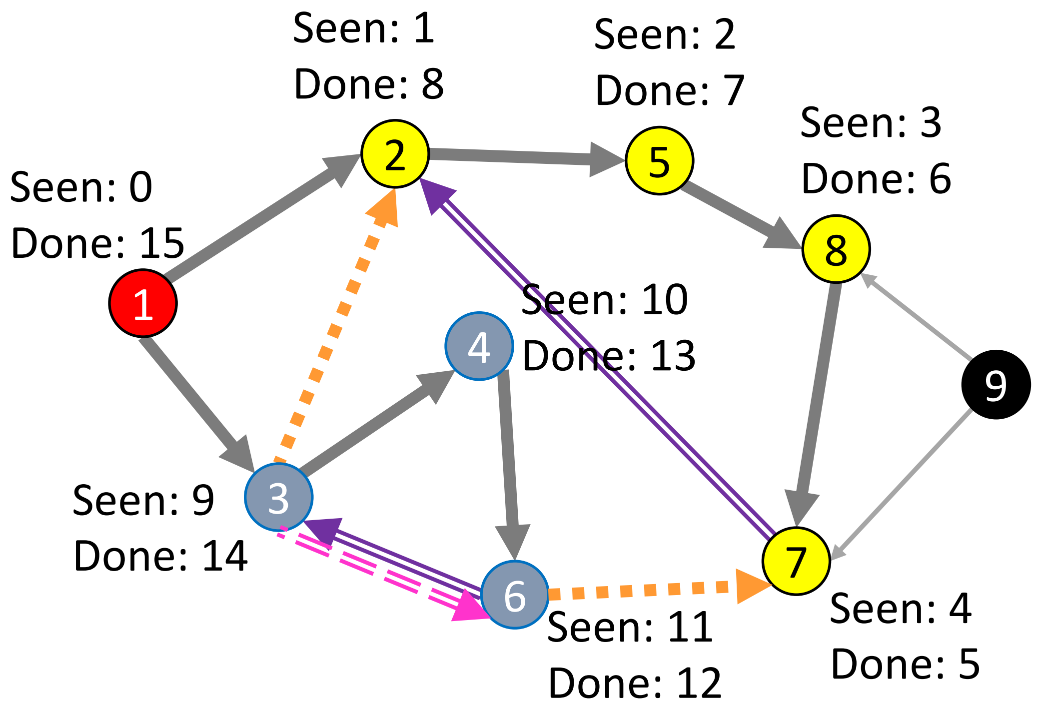


Using DFS

Consider the "seen times" and "done times"

Edges can be categorized:

- Tree Edge
  - (a, b) was followed when pushing
  - (a, b) when b was **unseen** when we were at a
- Back Edge
  - (a, b) goes to an "ancestor"
  - a and b **seen** but not **done** when we saw (a, b)
  - $t_{seen}(b) < t_{seen}(a) < t_{done}(a) < t_{done}(b)$
- Forward Edge
  - (a, b) goes to a "descendent"
  - b was **seen** and **done** between when a was **seen** and **done**
  - $t_{seen}(a) < t_{seen}(b) < t_{done}(b) < t_{done}(a)$
- Cross Edge
  - (a, b) connects "branches" of the tree
  - b was **seen** and **done** before a was ever **seen**
  - (a, b) when $t_{done}(b) > t_{seen}(a)$ and

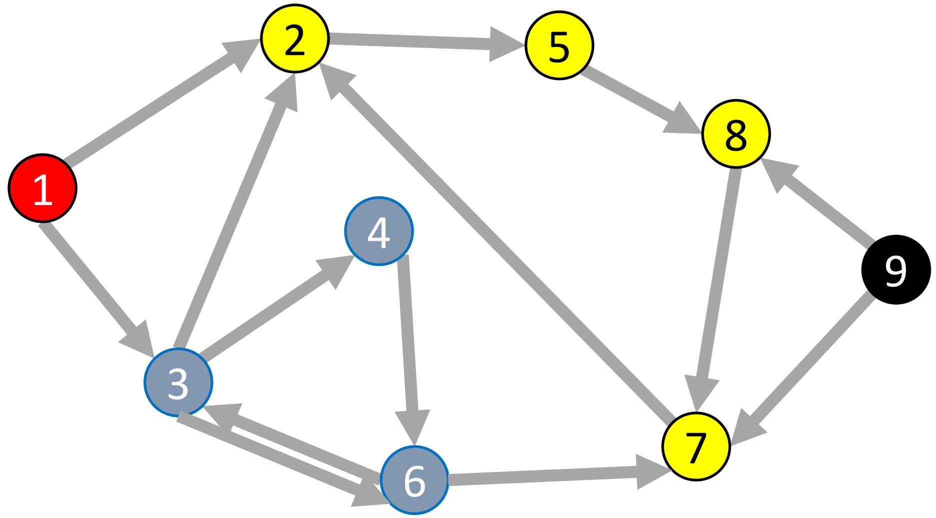


DFS: Cycle Detection

Idea: Look for a back edge!

```
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
```



DFS: Cycle Detection

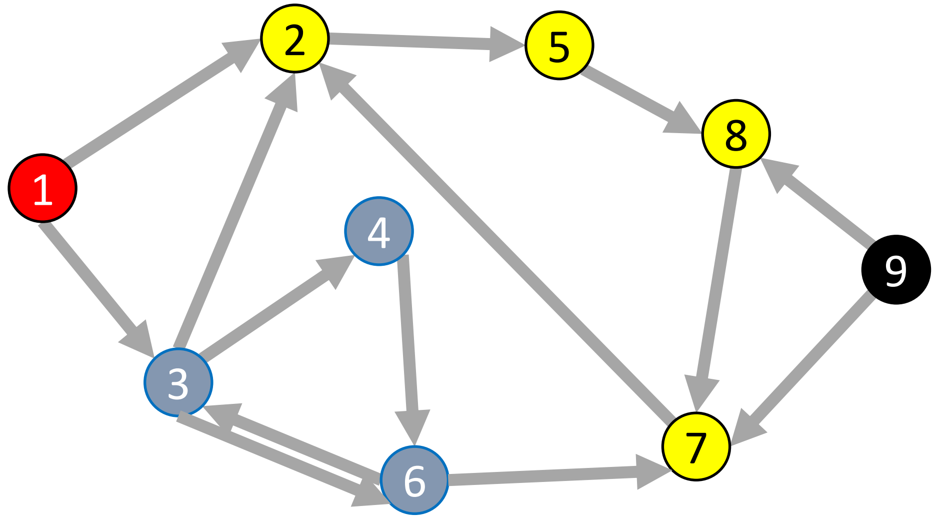
Idea: Look for a back edge!

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

def hasCycle_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
```

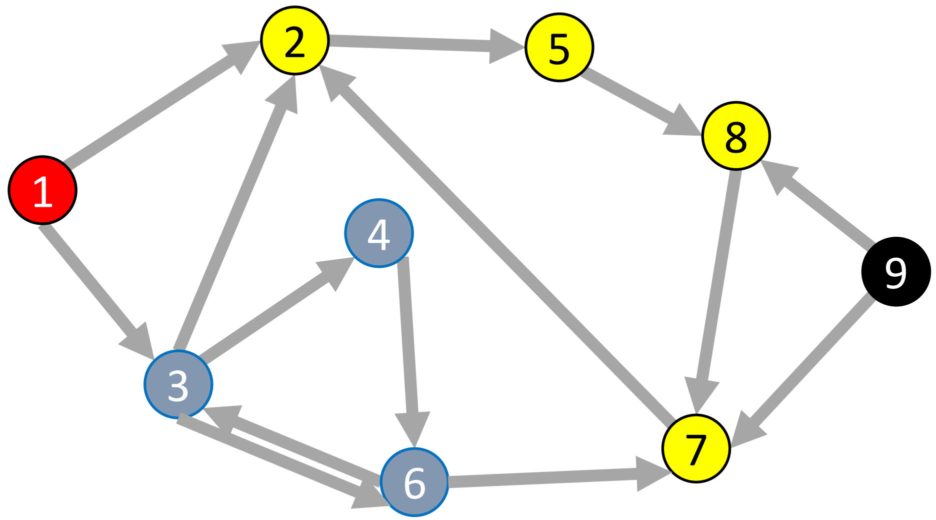


DFS: Cycle Detection

Idea: Look for a back edge!

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

def hasCycle _rec(graph, curr, seen, done):
        cycle = False
        mark curr as seen
        for v in neighbors(current):
                if v seen and v not done:
                        cycle = True
                elif v not seen:
                        cycle = dfs_rec(graph, v, seen, done) or cycle
        mark curr as done
        return cycle
```



Back Edges in Undirected Graphs

Finding back edges for an undirected graph is not **quite** this simple:

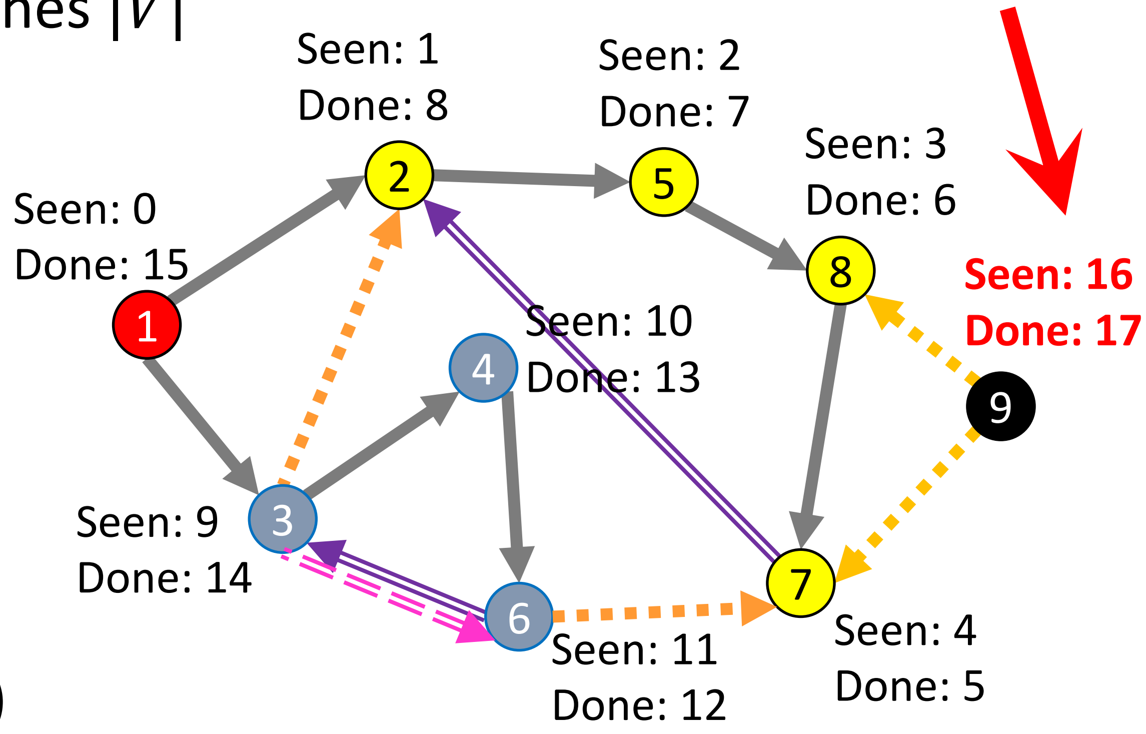
- The parent node of the current node is **seen** but not **done**
- Not a cycle, is it? It's the same edge you just traversed

Question: how would you modify our code to recognize this?

DFS “Sweep” to Process All Nodes

```
def dfs_sweep(graph):  # no start node given
        seen = [False, False, False, …] # length matches |V|
        done = [False, False, False, …] # length matches |V|
        for s in graph :  # do DFS at every vertex
            if s not seen:
                dfs_rec(graph, s, seen, done)

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



Time Complexity of DFS

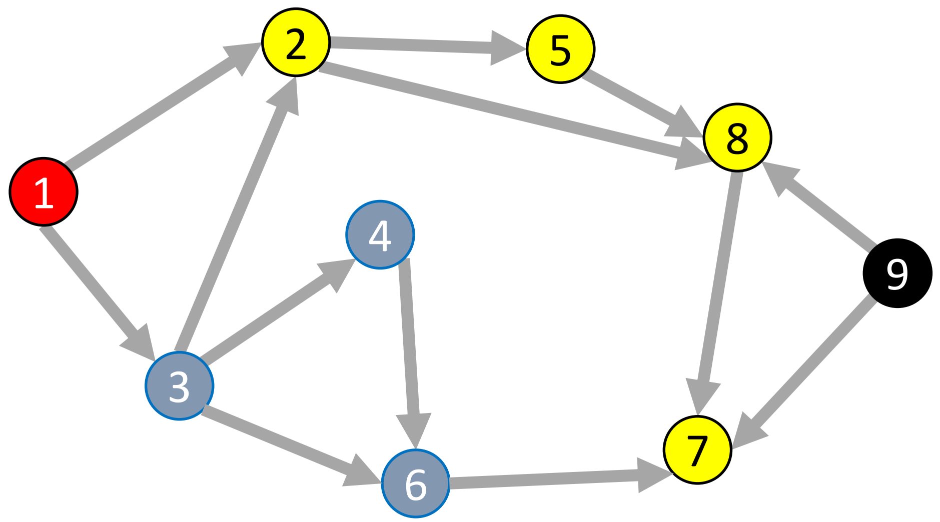
For a digraph having V vertices and E edges

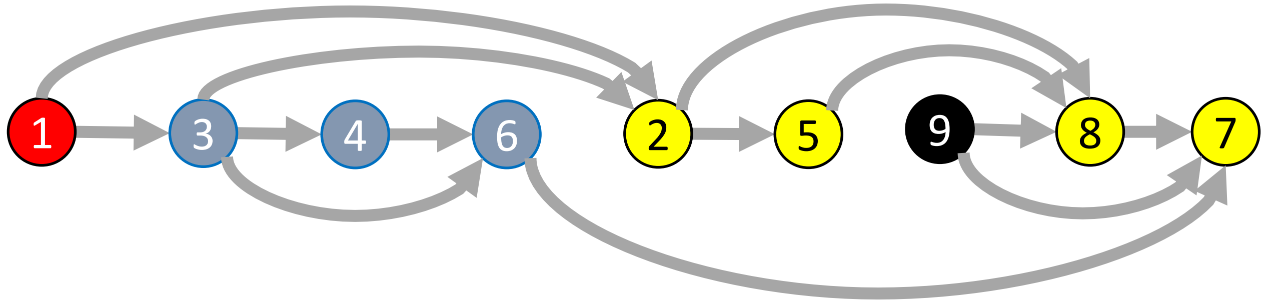
- Each edge is processed once in the while loop of dfs_rec() for a cost of $\Theta(E)$
  - Think about *adjacency list* data structure.
  - Traverse each list exactly once. (Never back up)
  - There are a total of **E** nodes in all the lists
- The non-recursive dfs() algorithm will do $\Theta(V)$ work even if there are no edges in the graph
- Thus over all time-complexity is $\Theta(V + E)$
  - Remember: this means the larger of the two values
  - Reminder: This is considered “linear” for graphs since there are two size parameters for graphs.
- Extra space is used for seen/done (or color) array.
  - Space complexity is $\Theta(V)$

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

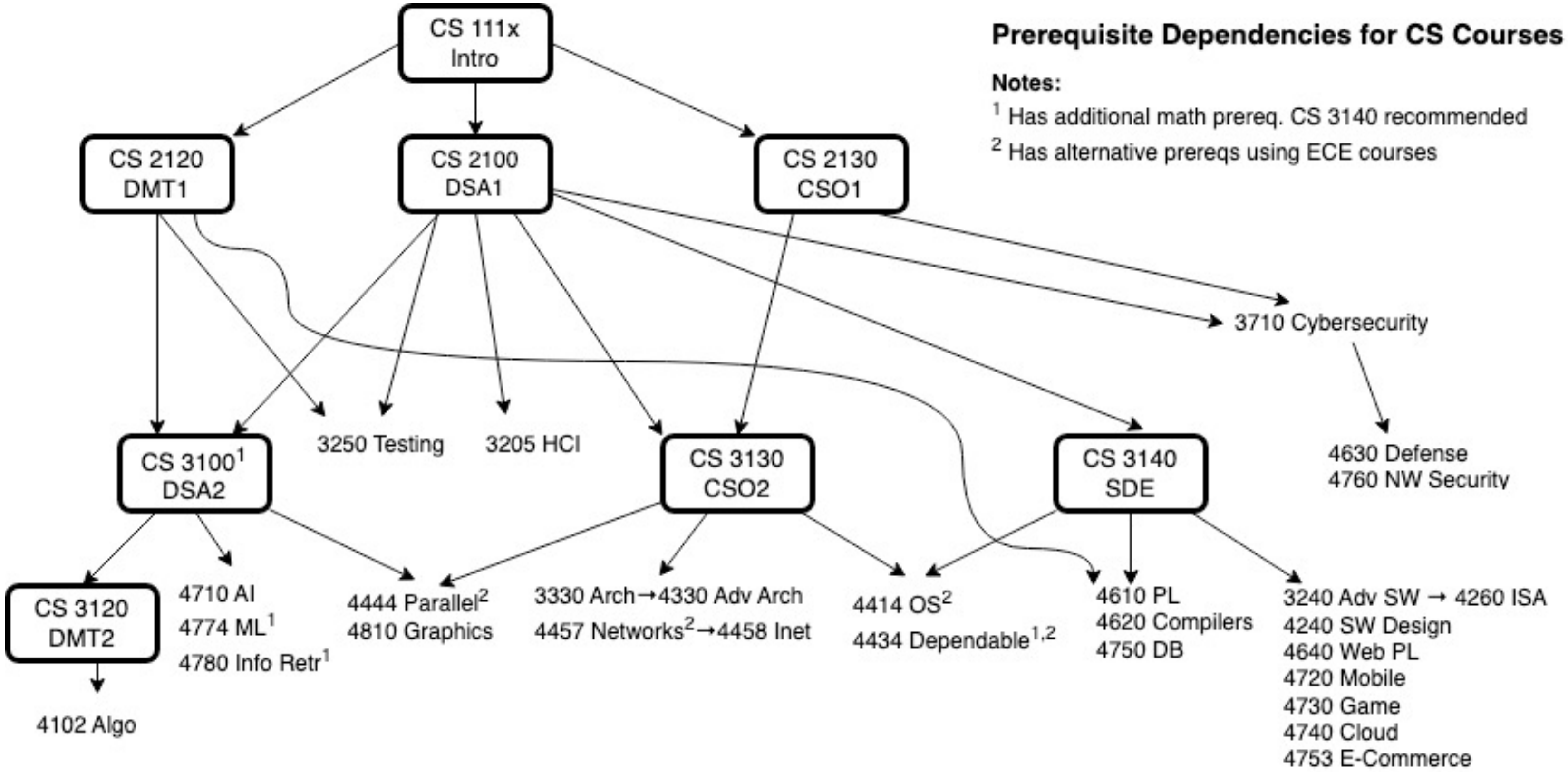




Topological Sort

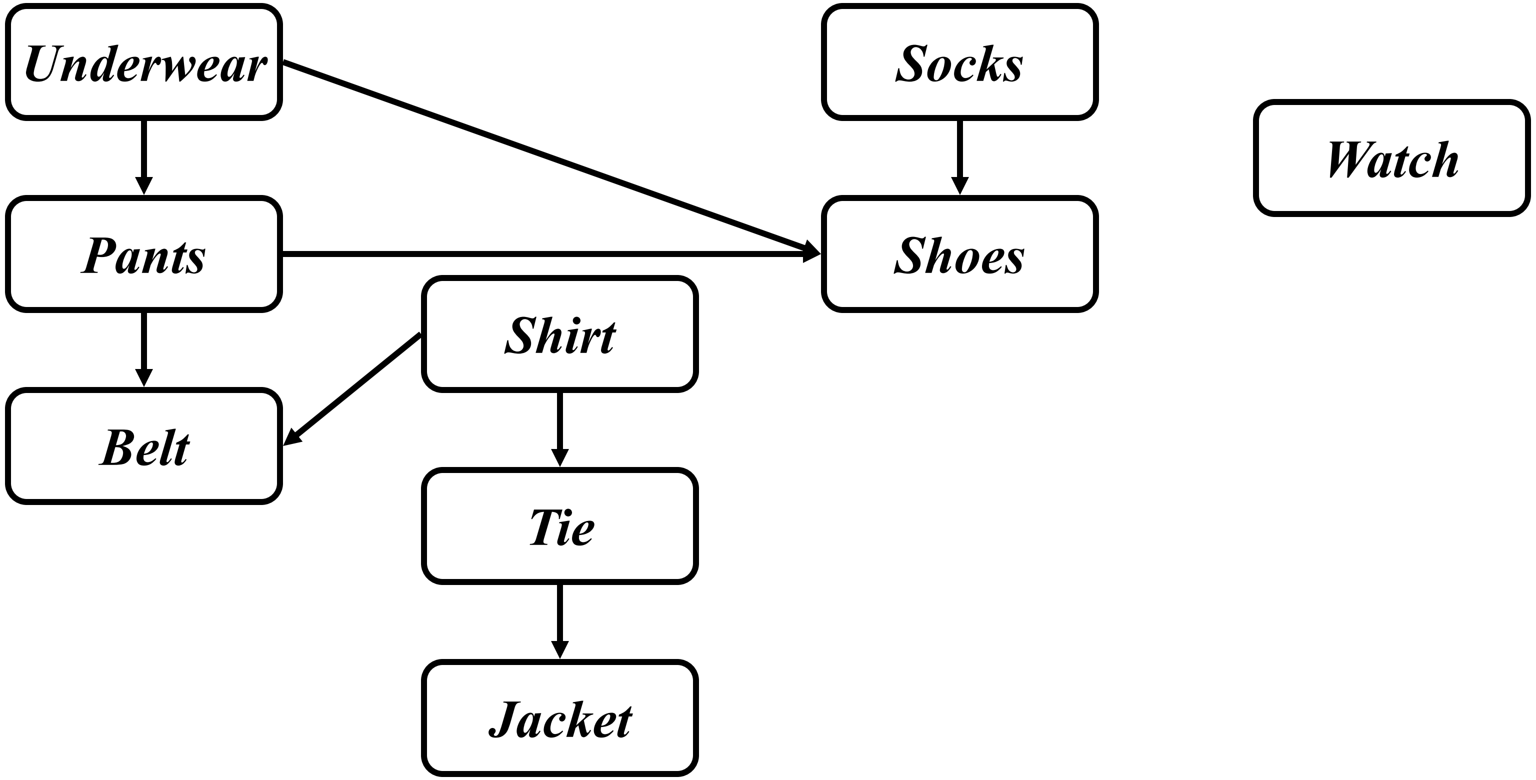
What are allowable orderings I can take all these CS classes?

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



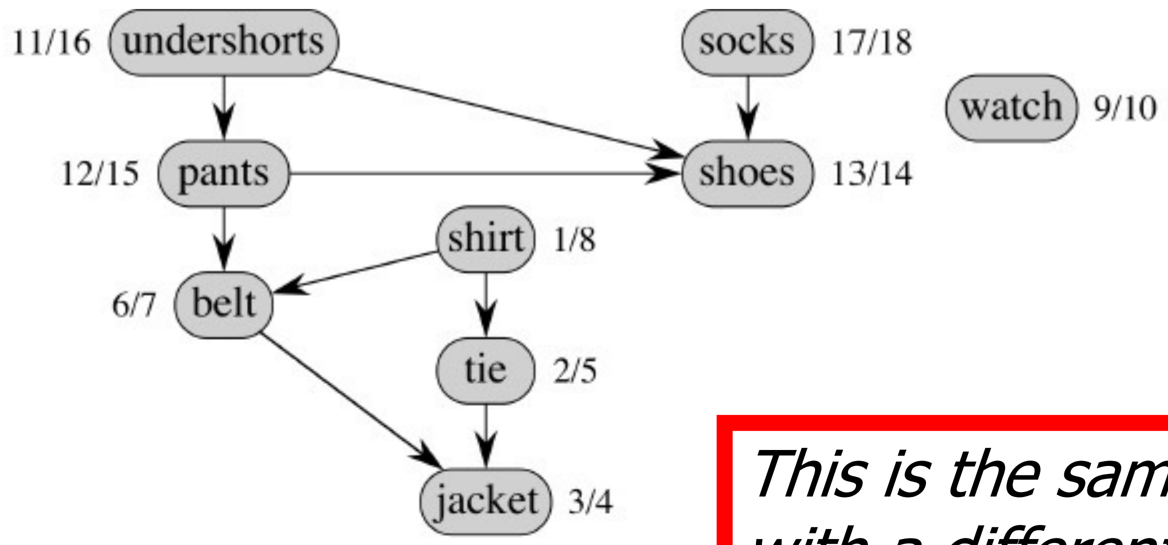
Topological Sort

Getting dressed



We Can Use DFS and Finish Times

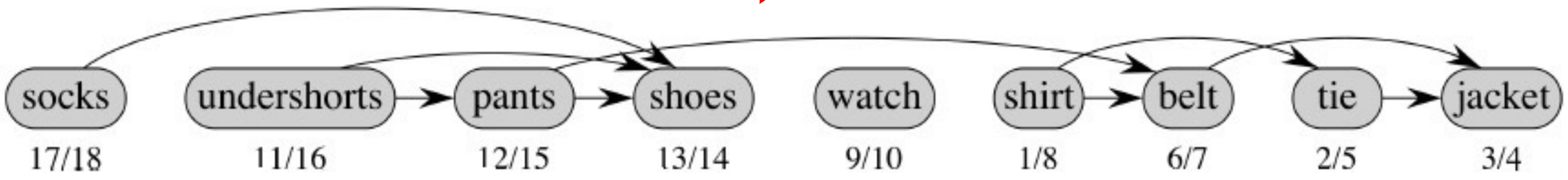
(a)



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.

(b)



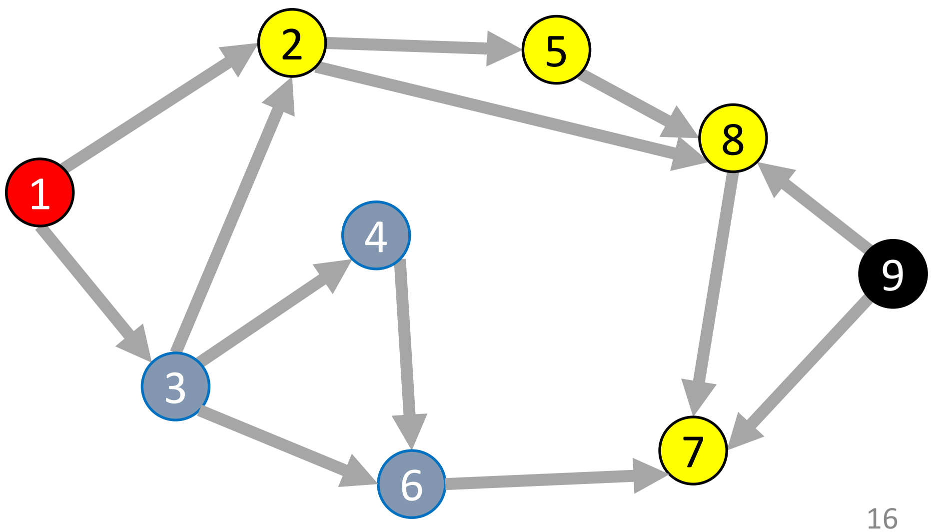
Topologically sorted vertices appear in reverse order of their finish times!

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

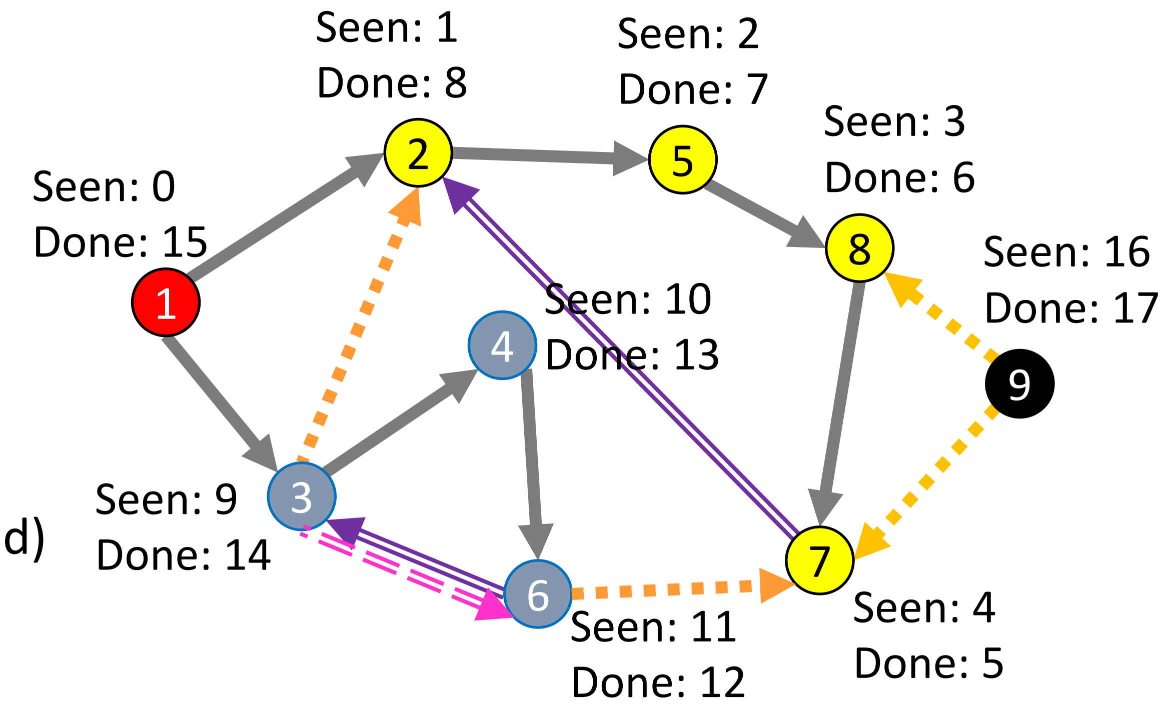


DFS: Topological sort

```
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

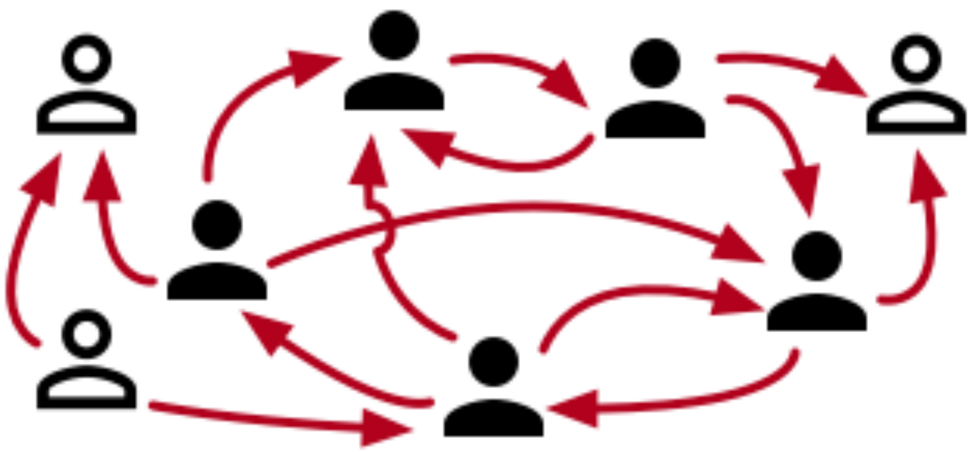
Common need: decompose a digraph into its SCCs

- 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

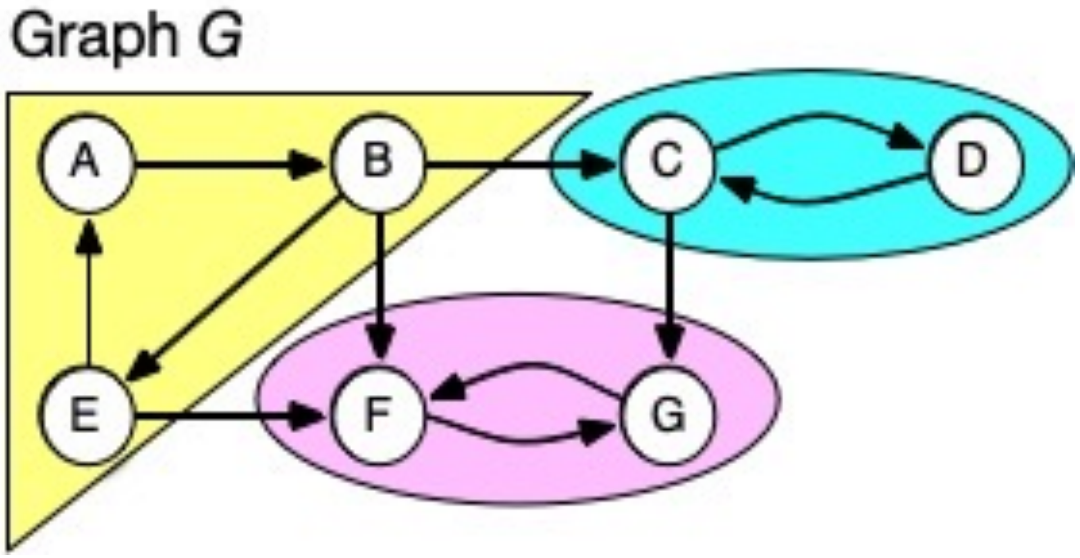
In this example, the group of solid-colored users is an SCC

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

SCC Example

Example: digraph below has 3 SCCs

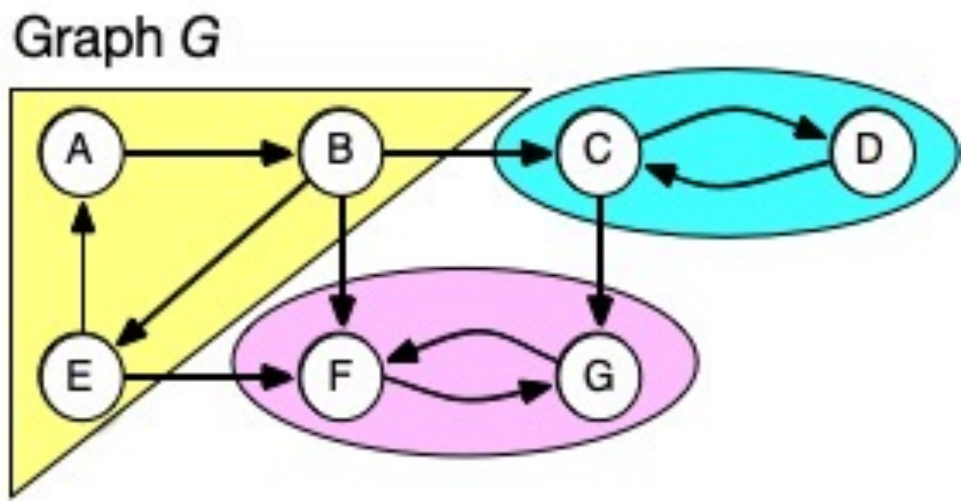
- Note here each SCC has a cycle. (Possible to have a single-node SCC.)
- Note connections to other SCCs, but no path leaves a SCC and comes back
- Note there's a unique set of SCCs for a given digraph

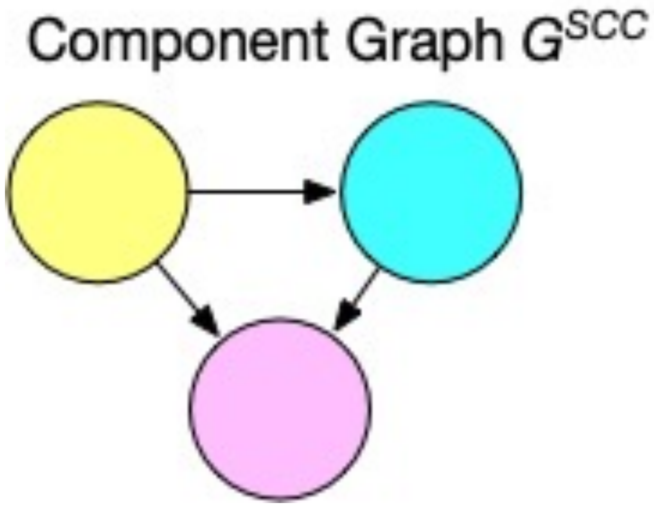


Component Graph

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 <u>G's</u> DFS tree,…)
4. The DFS forest produced in Step 3 is the set of SCCs

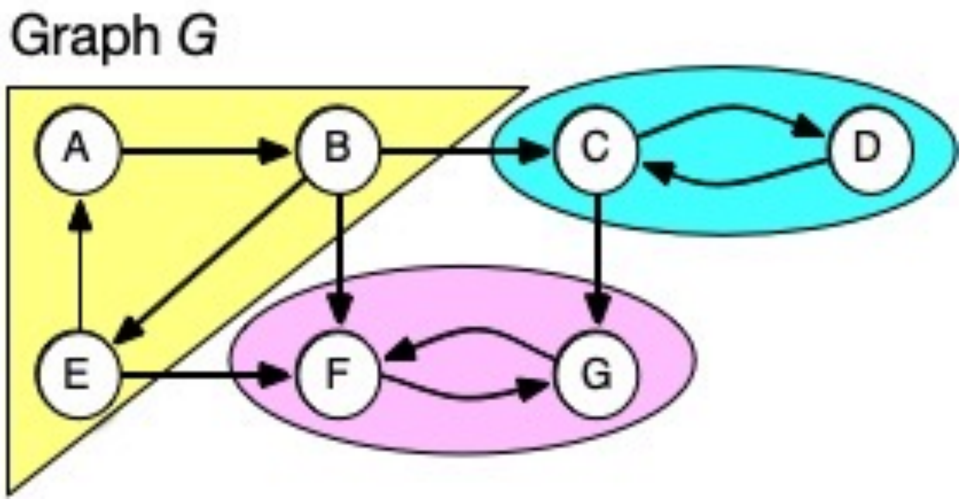
Why Do We Care about the Transpose?

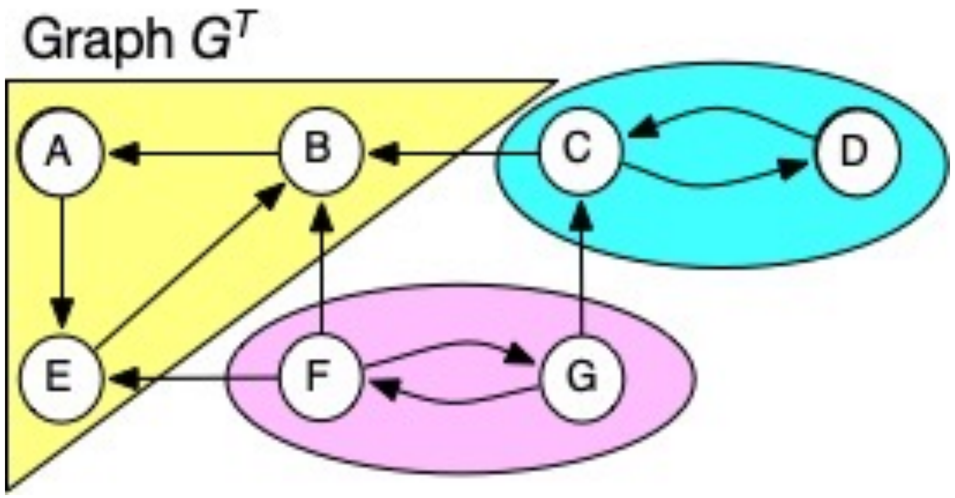
If we call DFS on a node in an SCC, it will visit all nodes in that SCC

- 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



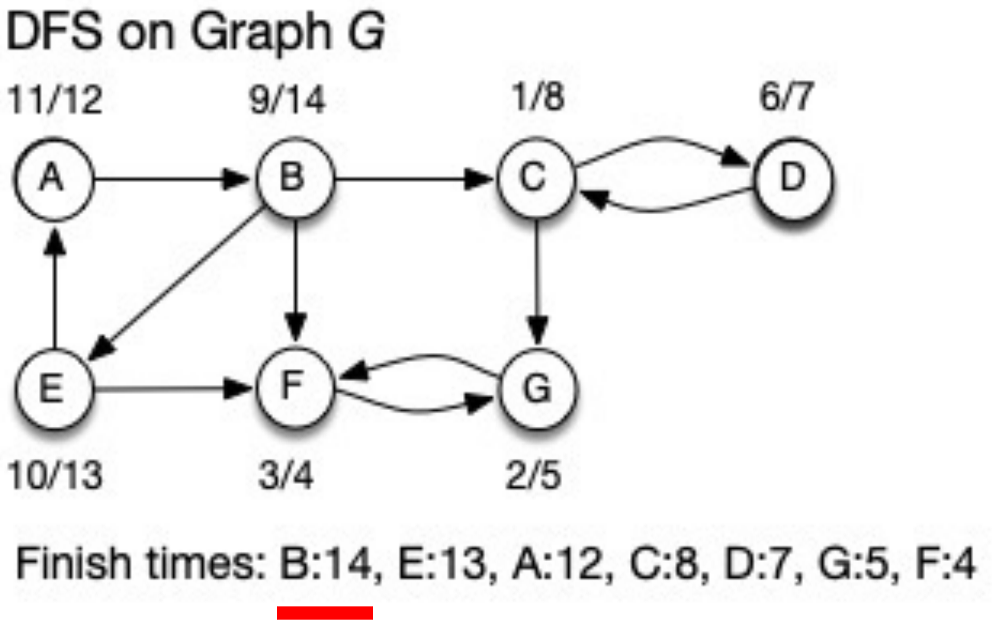


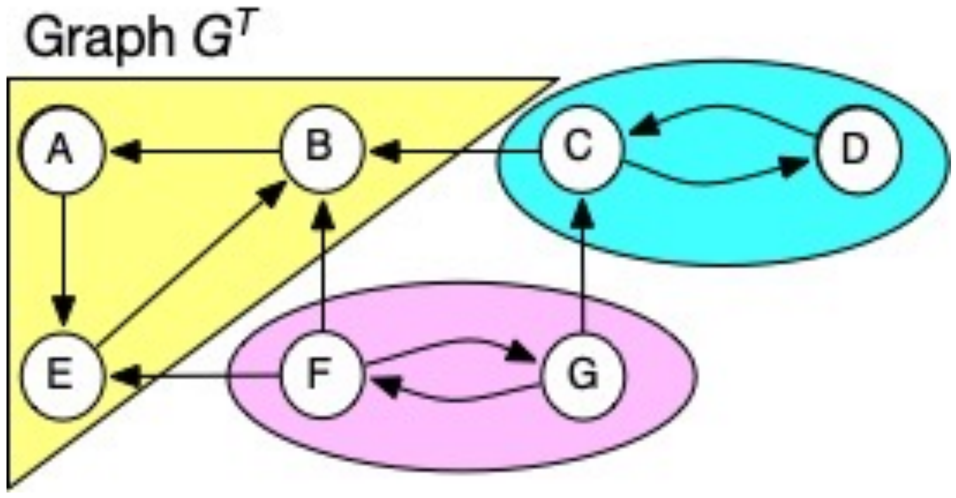
Why Do We Care About Finish Times?

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

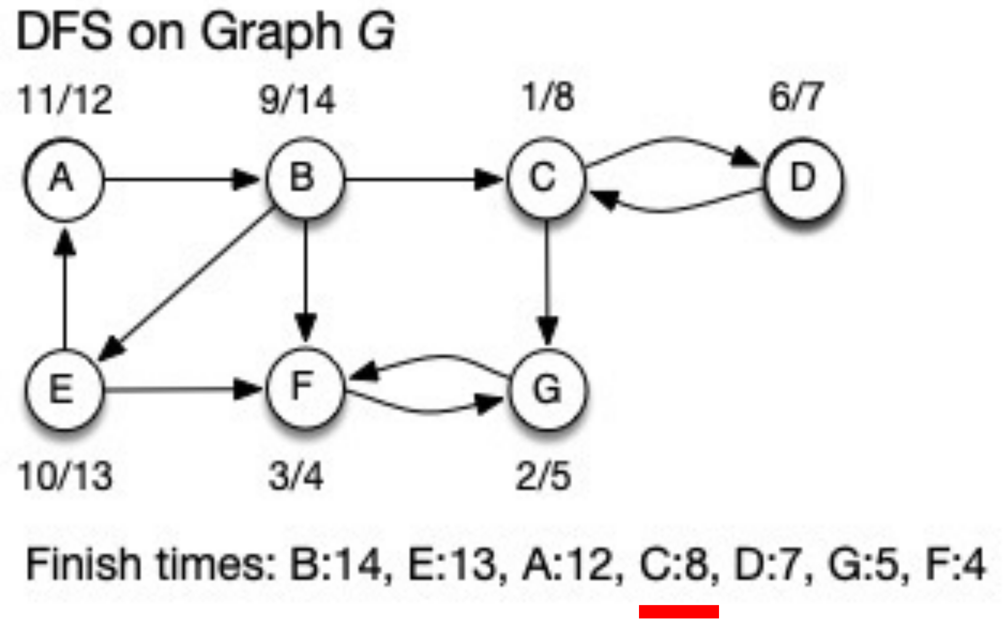


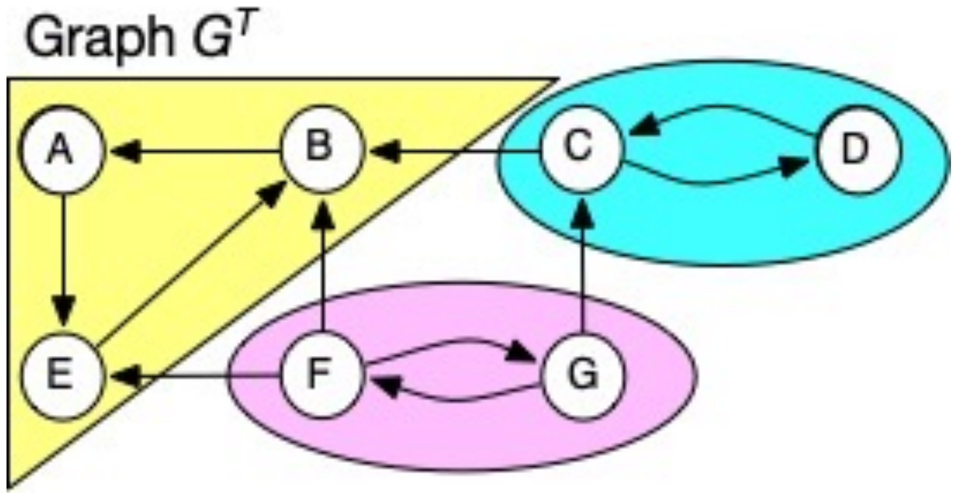


Why Do We Care About Finish Times?

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



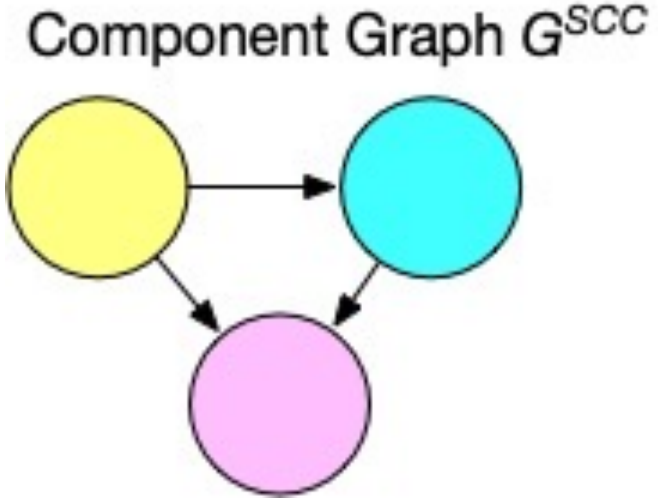


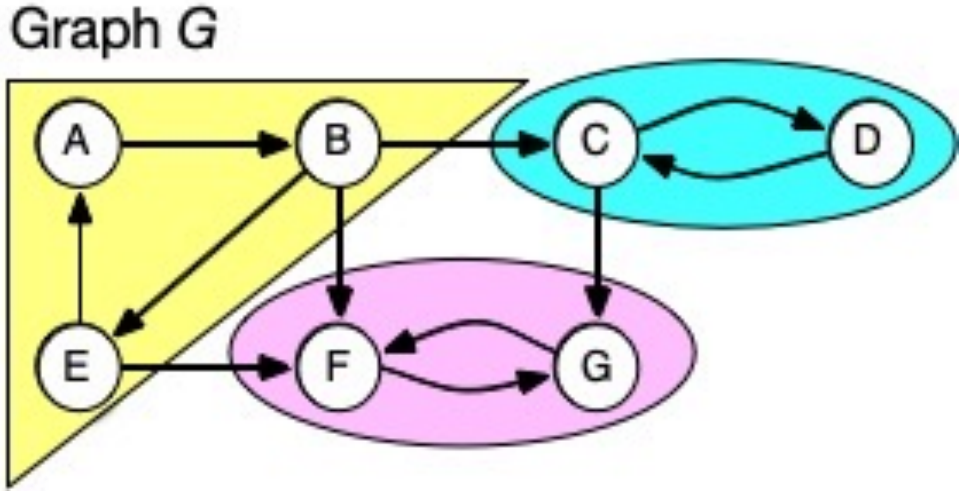
Ties to Topological Sorting

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