Using DFS for Topological Sorting and Strongly Connected Components

CS 4102: Algorithms
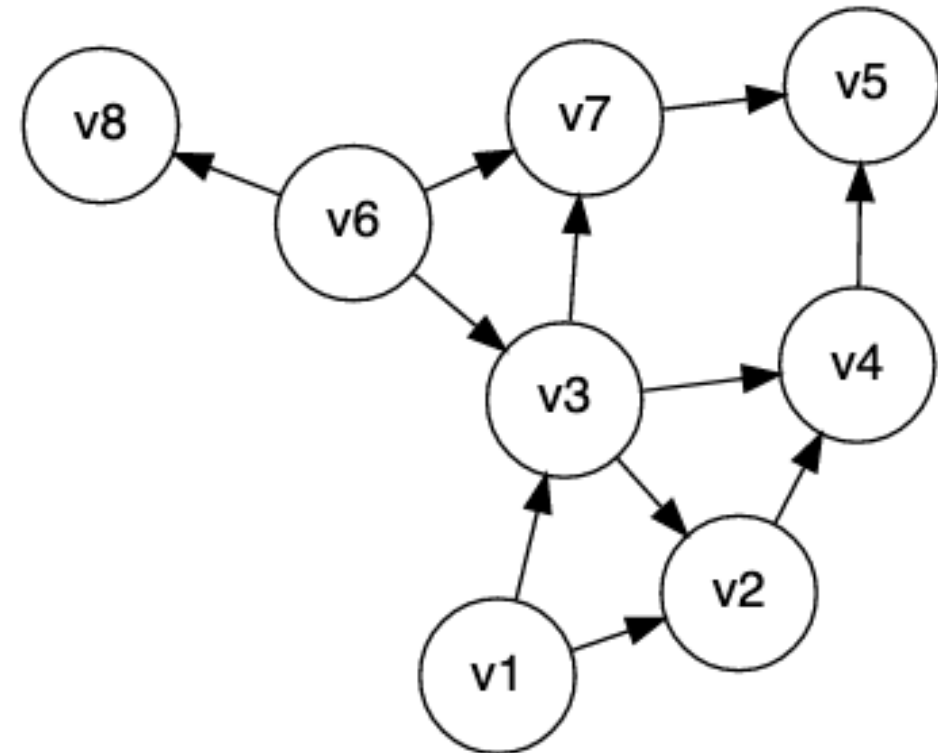
Spring 2021

Mark Floryan and Tom Horton

Topological Sorting

Readings: CLRS 22.4

Topological Sort

- Given a **directed acyclic graph**, construct a linear ordering of the vertices such that if there is an edge from *u* to *v*, then *u* appears before *v* in the ordering.
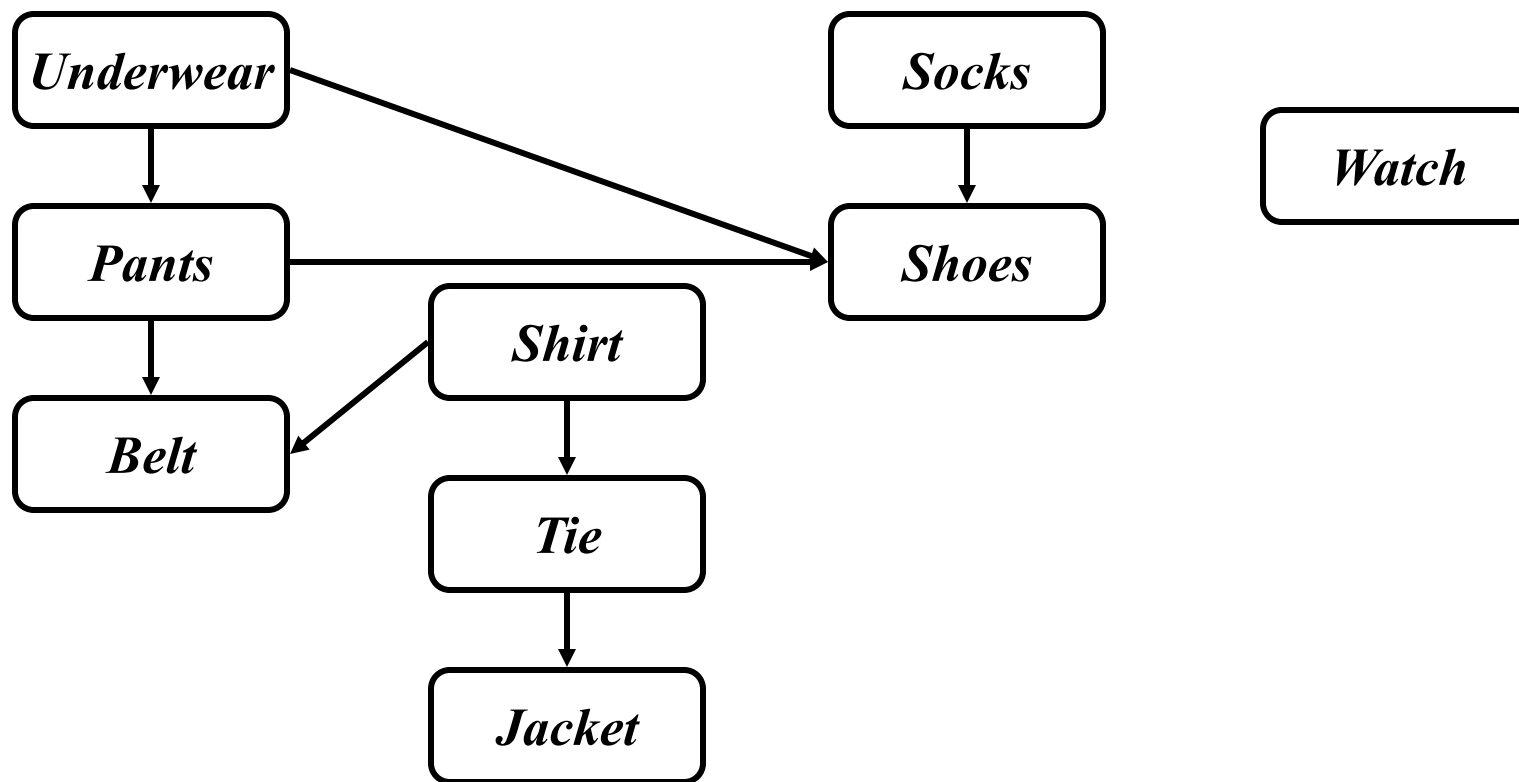
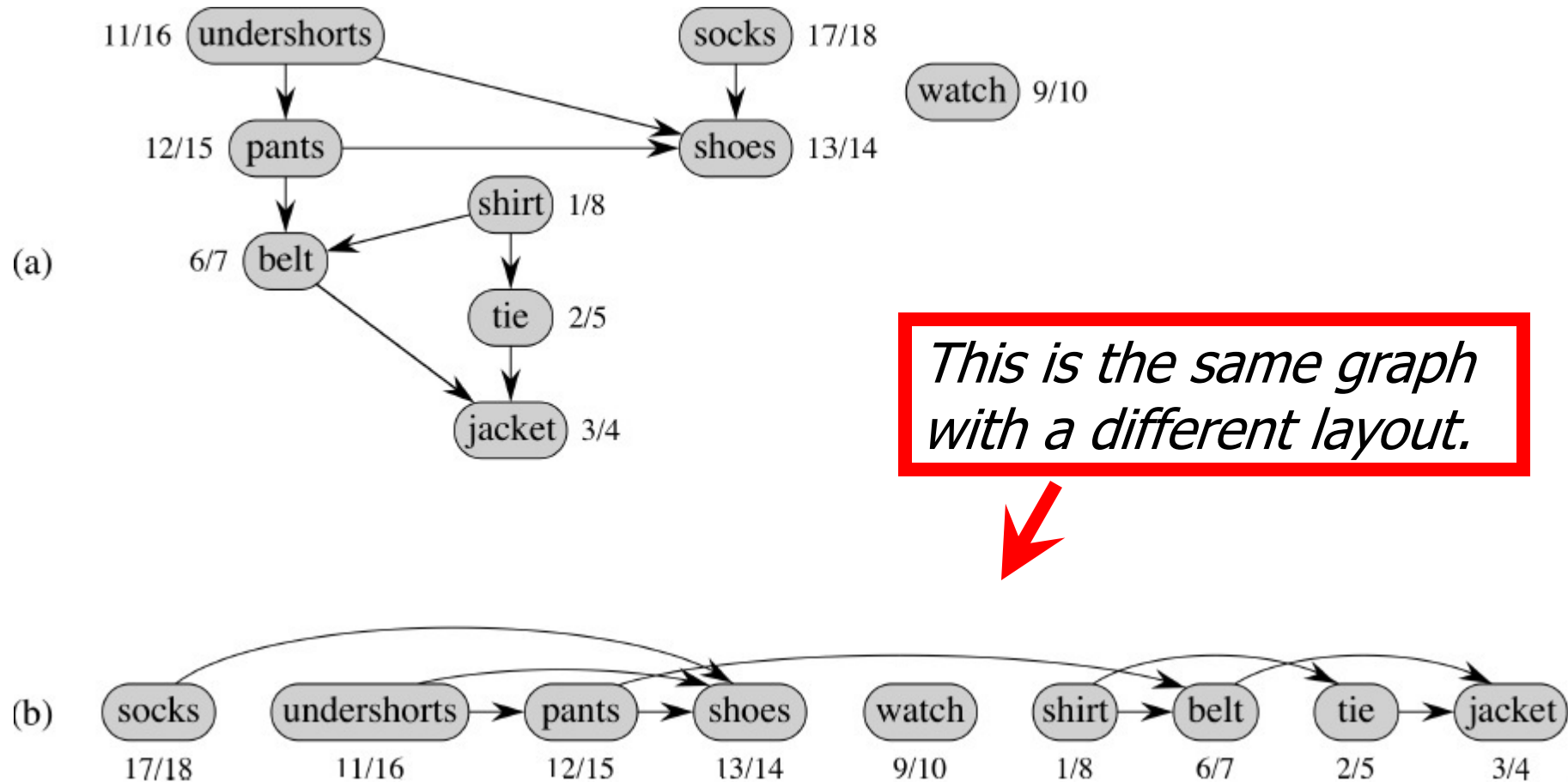- One valid topological sort is:
  V1 V6 V8 V3 V2 V7 V4 V5

- What are allowable orderings I can take all these CS classes?
  - Note there are many possible orderings
  - Unlike sorting a list

Getting Dressed

This is the same graph with a different layout.

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

Topological Sort Algorithm

- Strategy: modify the two DFS functions so that they order nodes by finish-time in reverse order. This slide: DFS "Sweep".

DFS(G)
0 toposort-list = [ ] // empty list
1 for each vertex u in G.V
2   u.color = WHITE
3   u.π = NIL
4 time = 0
5 for each vertex u in G.V
6   if u.color == WHITE // if unseen
7 DFS-VISIT(G, u) // explore paths out of u
8 // toposort-list contains the result

DFS-VISIT(G, u)

1   time = time + 1 // white vertex u has just been discovered

2   u.d = time // discovery time of u

3   u.color = GRAY // mark as seen

4   for each v in G.Adj[u] // explore edge (u, v)

5   if v.color == WHITE   // if unseen

6   v.π = u

7   DFS-VISIT(G, v) // explore paths out of v (i.e., go "deeper")

8   u.color = BLACK // u is finished

9   time = time + 1

10 u.f = time // finish time of u

11 toposort-list.prepend(u)

- Topological sort is a type of sort
  - Implies an ordering
  - Can sort backwards, of course

- Forward topological order
  - If edge **vw** in graph, then topo[**v**] < topo[**w**]
- Reverse topological order
  - If edge **vw** in graph, then topo[**v**] > topo[**w**]

- And, every directed graph has a transpose, which means... (see next slide)

What's an Edge Mean?

- ## What does our graph model?
  - Edge **uv** means do **u** first, then **v**. Or, ...
  - Edge **uv** means task **u** depends on v (i.e. **v** must be done first)
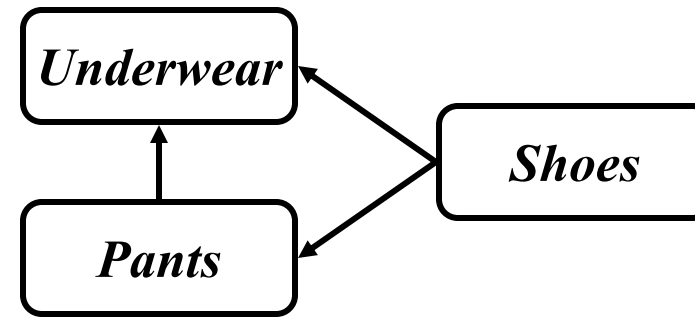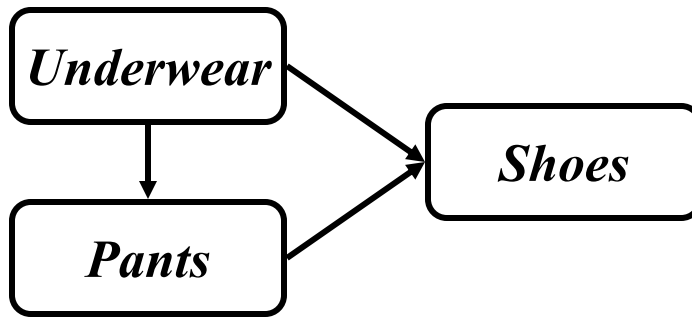
  - The latter is called a dependency graph
  - "forward in time" vs. "depend on this one"

- ## Big deal? No, we can order vertices in reverse topological order if needed

Strongly Connected Components in a Digraph

Readings: CLRS 22.5, 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

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
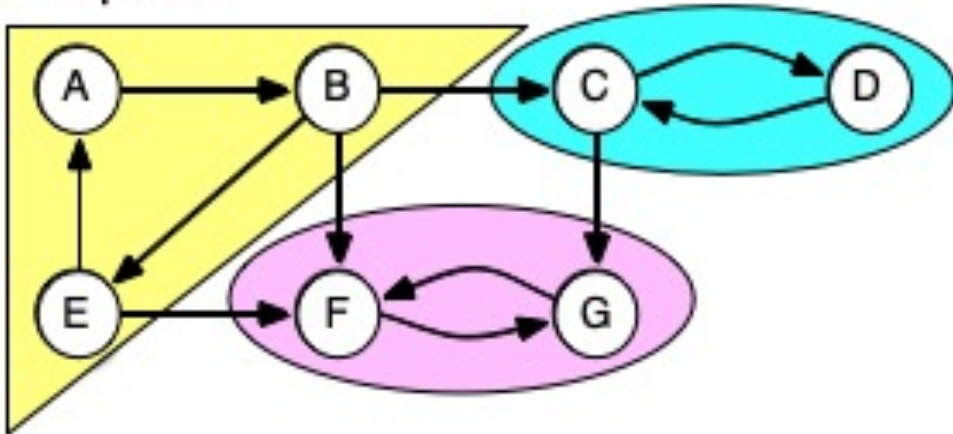 - Note there's a unique set of SCCs for a given digraph
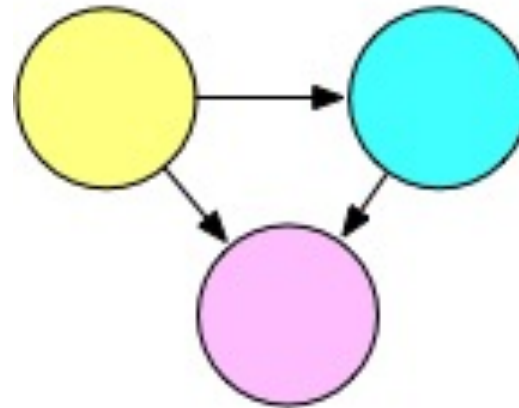

Graph G
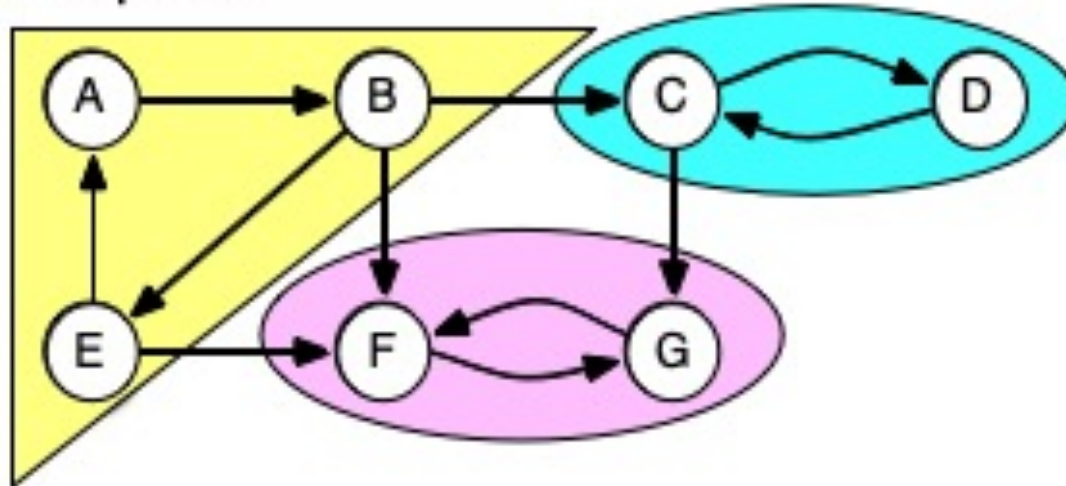
Component Graph

- 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 Graph into SCCs

- Several algorithms do this using DFS
- We'll use CLRS's choice (by Kosaraju and Sharir)
- Algorithm is:
  1. Call *DFS-sweep(G)* to find finishing times $u.f$ for each vertex u in G.
  2. Compute G^T, the transpose of diagraph 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 $u.f$. (Start with the vertex with largest finish time,…)
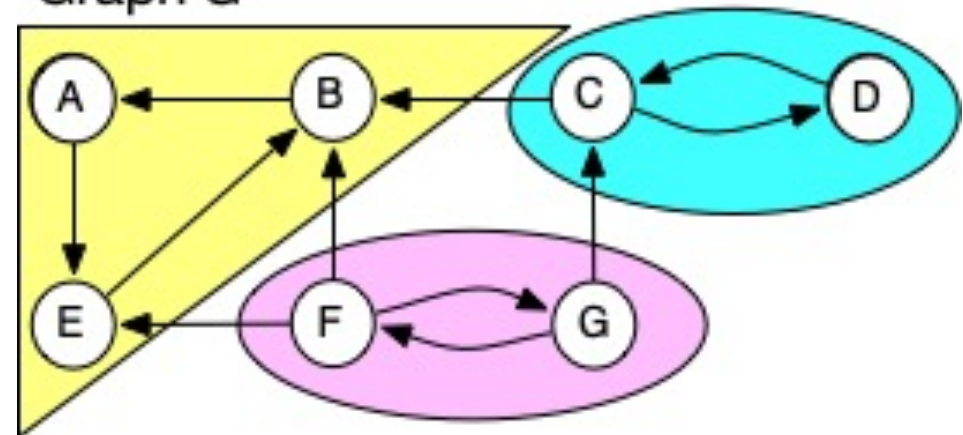 4. The DFS forest produced in Step 3 is the set of SCCs

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
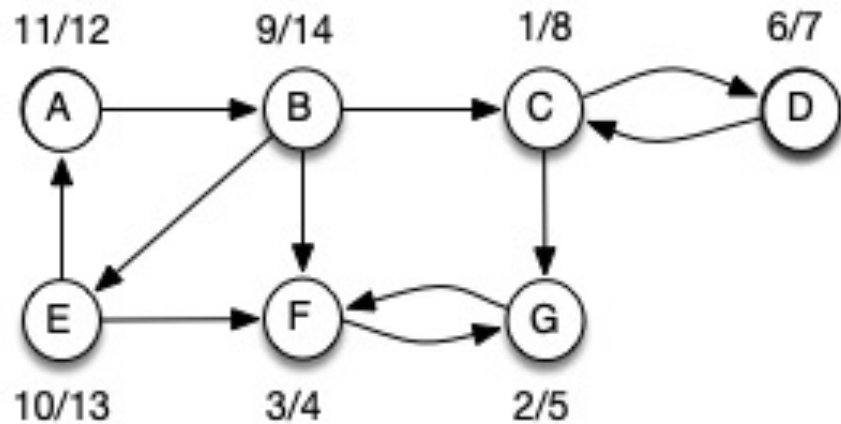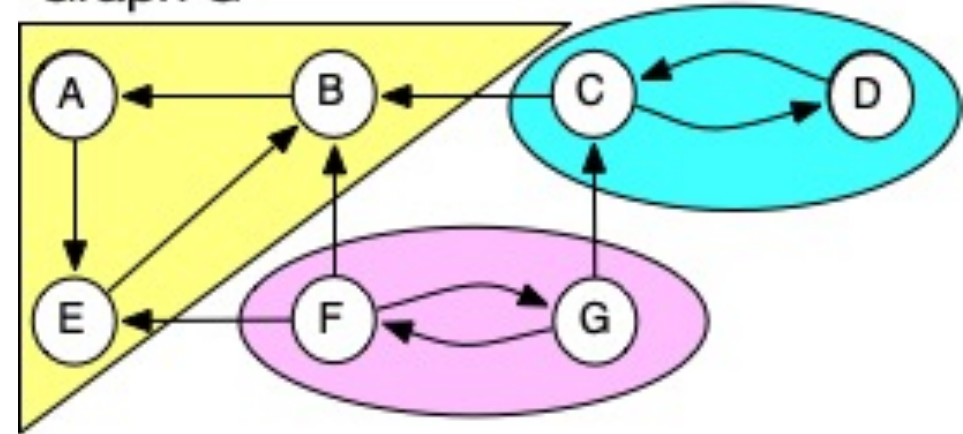
Graph G
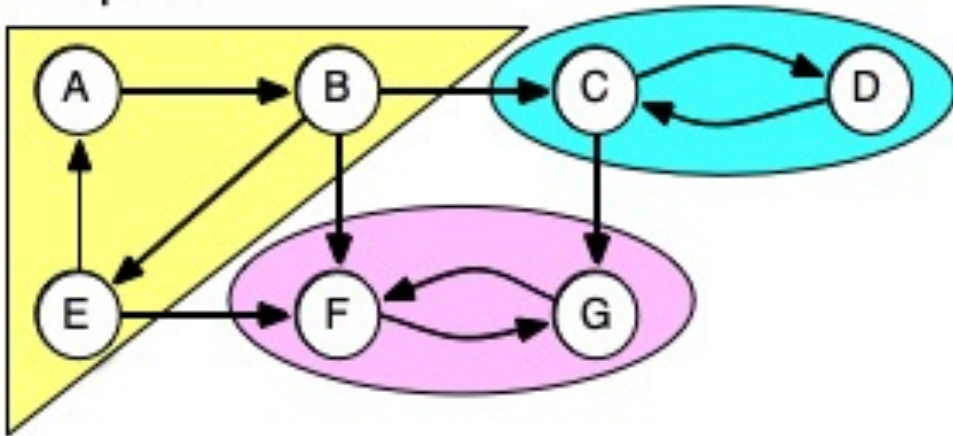
Graph G^T

Why Do We Care About Finish Times?

- Our algorithm first finds DFS finish times <u>in G</u>
- Then calls recursive DFS <u>in transpose</u> from vertex with largest finish time (here, B)
  - Reversed edges in G^T stop it visiting nodes in other SCCs
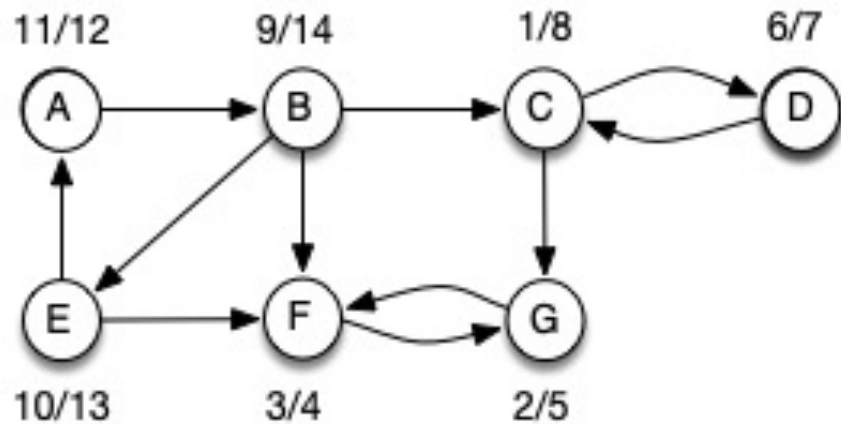

DFS on Graph G

11/12   9/14   1/8   6/7

A → B → C ⇄ D

10/13   3/4   2/5

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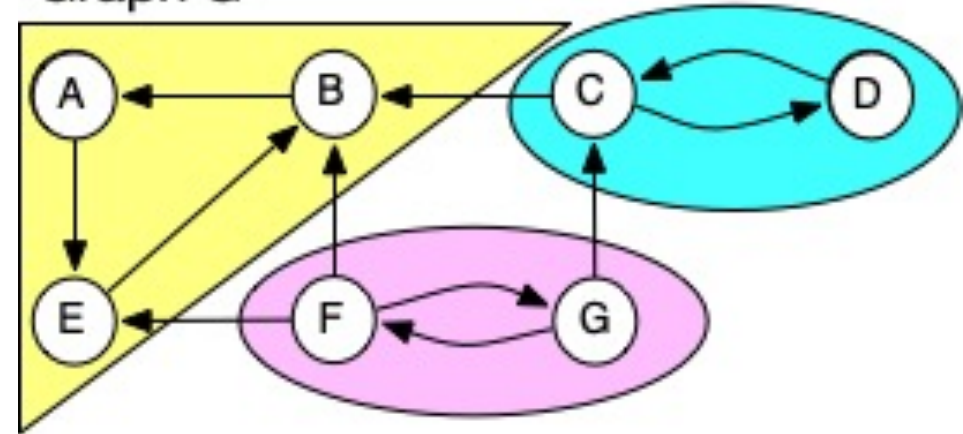
Graph G^T

- After recursive DFS <u>in transpose</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

- 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}
  - Topological sort controls the order we do things, and DFS finds all the reachable nodes in an SCC

Component Graph G^{SCC}
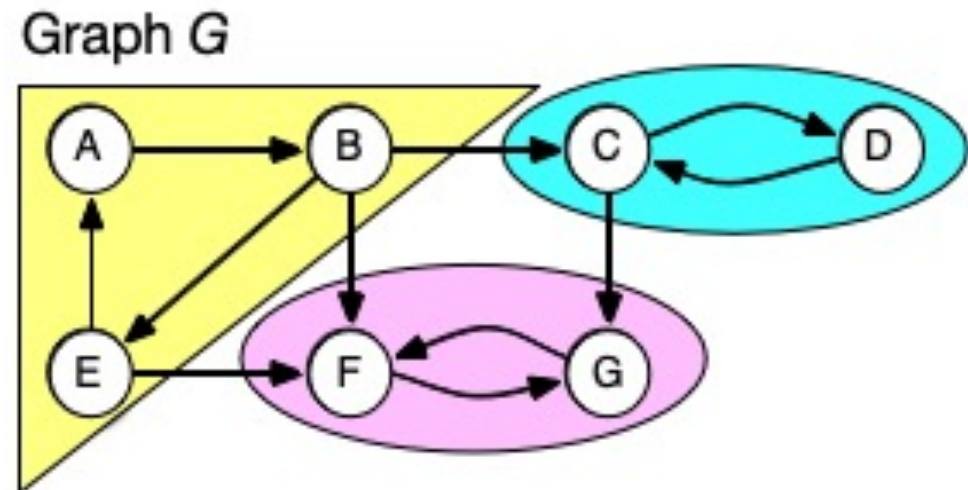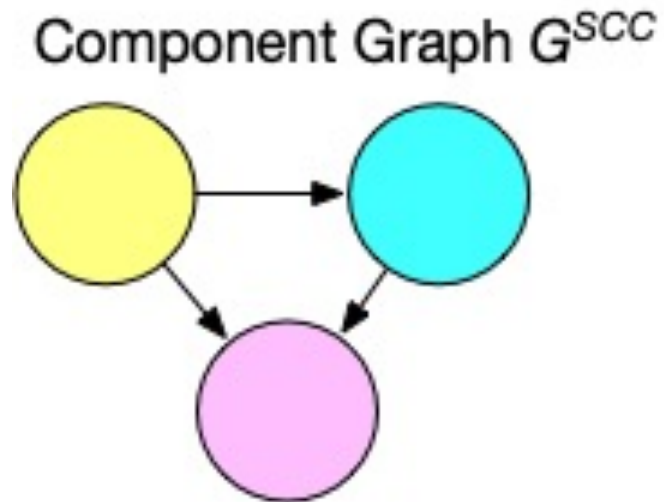
Graph G

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