Using DFS for Topological Sorting and Strongly Connected Components

CS 4102: Algorithms

Fall 2021

Mark Floryan and Tom Horton

DFS

CLRS Section 22.3 on DFS

Readings

- CLRS:
  - Section 22.3 on DFS
  - Later/eventually:
    - Section 22.4 on Topological Sort
    - Section 22.5 on Strongly Connected Components

DFS: the Strategy in Words

- Depth-first search: Strategy
  - Go as deep as can visiting un-visited nodes
    - Choose any un-visited vertex when you have a choice
  - When stuck at a dead-end, backtrack as little as possible
    - Back up to where you could go to another unvisited vertex
  - Then continue to go on from that point
  - Eventually you'll return to where you started
    - Reach all vertices? Maybe, maybe not

Observations about the DFS Strategy

- Note: we must keep track of what nodes we've visited
- DFS traverses a subset of E (the set of edges)
  - Creates a tree, rooted at the starting point: the Depth-first Search Tree (DFS tree)
  - Each node in the DFS tree has a distance from the start. (We often don't care about this, but we could.)
- At any point, all nodes are either:
  - Un-discovered
  - Finished (you backed up from it), or
  - Discovered (i.e. visited) but not finished
    - On the path from the current node back to the root
    - We might back up to it
  - (Later we'll call these states: white, black and gray respectively)

DFS Strategy 1: Use a stack

- Maintain a Stack (Let’s call it S)

- Start at some node ‘s’ (push ‘s’ to S and mark as visited)
- While S not empty
  - Pop a node ‘n’ from S
  - Process ‘n’ if necessary (depending on problem you are solving)
  - For each non-visited neighbor of ‘n’
    - Mark neighbor as visited
    - Push neighbor onto S
  - Repeat

- Sound familiar? Same as BFS but uses stack instead of queue!
- Or we can implement recursively (see next slide)

DFS Strategy #2

- Use a recursive function to “visit” each node
  - Need a non-recursive function to initialize and make first call

- Before we look at this code… Important!
  - Best to think of DFS is a strategy as well as a single, particular bit of pseudo-code
  - We often add things to DFS code to solve problems
    - Code shown next is very minimal
  - ”Swiss Army Knife” of graph algorithms?

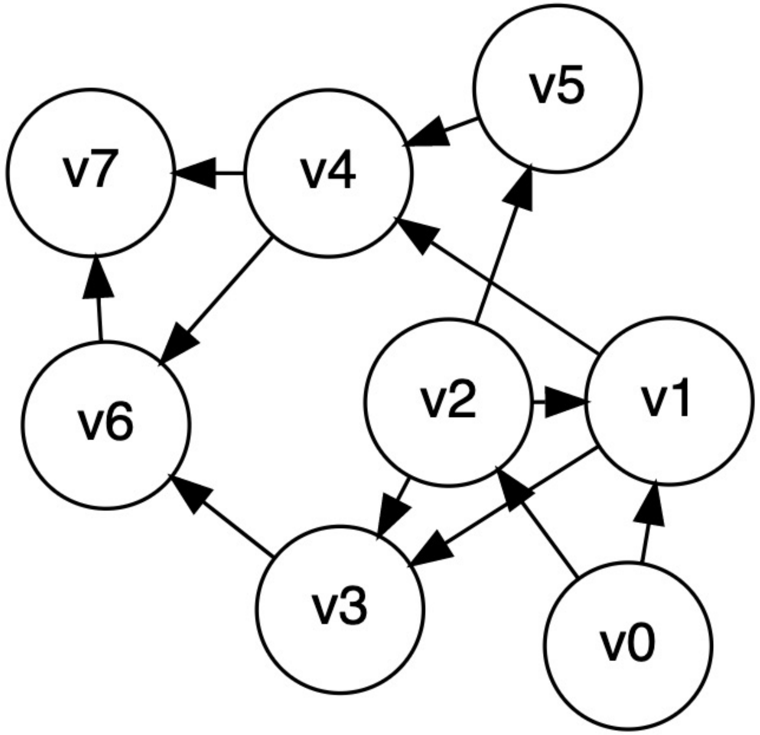
DFS Strategy 2: Recursion

```
def dfs(graph, start):                          //Main loop, inits and calls
    visited = {}
    dfs_recurse(graph, start, visited)

def dfs_recurse(graph, curnode, visited):       //sometimes called dfs_visit()
    visited[curnode] = True
    alist = graph.get_adjlist(curnode)          //get the neighbors of curnode
    for v in alist:
        if v not in visited:
            print("  dfs traversing edge:", curnode, v)
            dfs_recurse(graph, v, visited)
    # end for-all adjacent vertices
    return
```

depth-first search, example

- Let’s start at V0



DFS to Process all Vertices in a Graph

- Purpose: do all required initializations, then call dfs_recurse() as many times as needed to visit all nodes.
  - May create a DFS forest.
- Can be used to count connected components
  - Could remember which nodes are in each connected component

```
def dfs_sweep(graph, start):
    visited = {}

     # loop repeats DFS on every unvisited node
    for v in graph:
      if v not in visited:
          dfs_recurse(graph, v, visited)
```

Using DFS to Find if a Graphic is Acyclic

- Does a graph have a cycle?
  - DFS is great for this
  - But, slightly harder if graph is undirected

- Use DFS tree: classify edges and nodes as you process them
  - Nodes:
    - White: unvisited
    - Black: done with it, backed up from it (never to return)
    - Gray: Have reached it; exploring its adjacent nodes; but not done with it

CLRS's DFS Algorithm (non-recursive part)

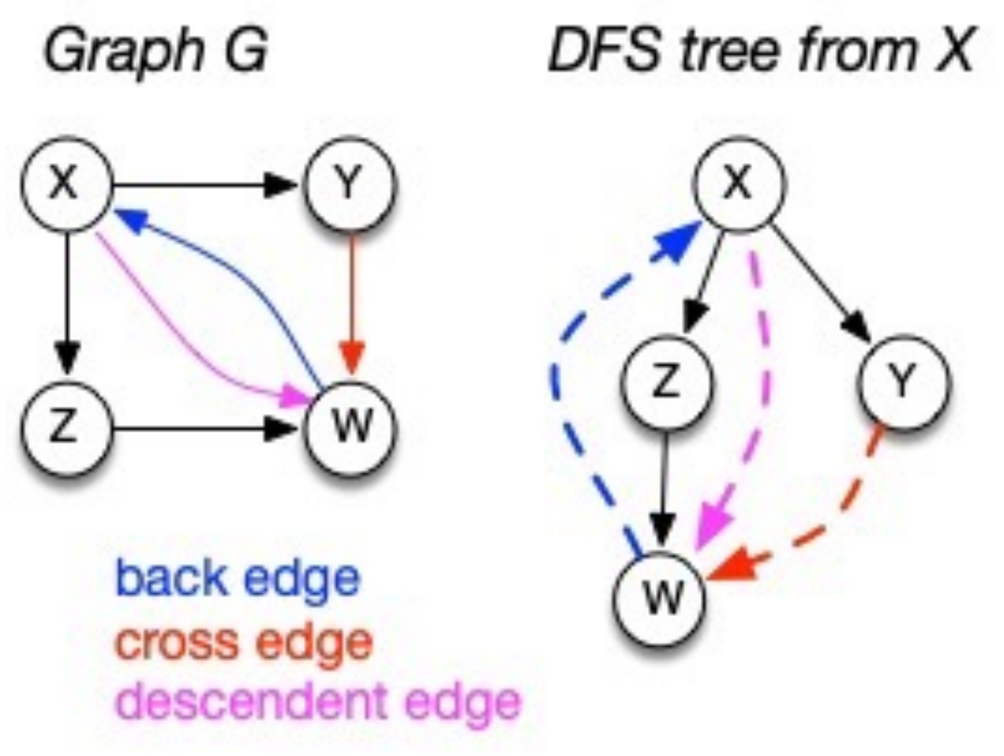
```
DFS(G)    // we called this dfs_sweep() earlier
for each vertex u in G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u in G.V
    if u.color == WHITE  // if unseen
        DFS-VISIT(G, u)  // explore paths out of u
```

CLRS's DFS Algorithm (recursive part)

```
DFS-VISIT(G, u)     // we called this dfs_recurse() earlier
time = time + 1  // white vertex u has just been discovered
u.d = time  // discovery time of u
u.color = GRAY  // mark as seen
for each v in G.Adj[u]  // explore edge (u, v)
    if v.color == WHITE   // if unseen
        v.π = u
        DFS-VISIT(G, v)  // explore paths out of v (i.e., go "deeper")
u.color = BLACK  // u is finished
time = time + 1
u.f = time  // finish time of u
```

Depth-first search tree

- As DFS traverses a digraph, edges classified as:
  - tree edge, back edge, descendant edge, or cross edge
  - If graph undirected, do we have all 4 types?



Using Non-Tree Edges to Identify Cycles

- From the previous graph, note that:
- Back edges (indicates a cycle)
  - dfs_recurse() sees a vertex that is gray
  - This back edge goes back up the DFS tree to a vertex that is on the path from the current node to the root
- Cross Edges and Descendant Edges (not cycles)
  - dfs_recurse() sees a vertex that is black
  - Descendant edge: connects current node to a descendant in the DFS tree
  - Cross edge: connects current node to a node in another subtree – not a descendant of current node

Non-tree Edges in DFS

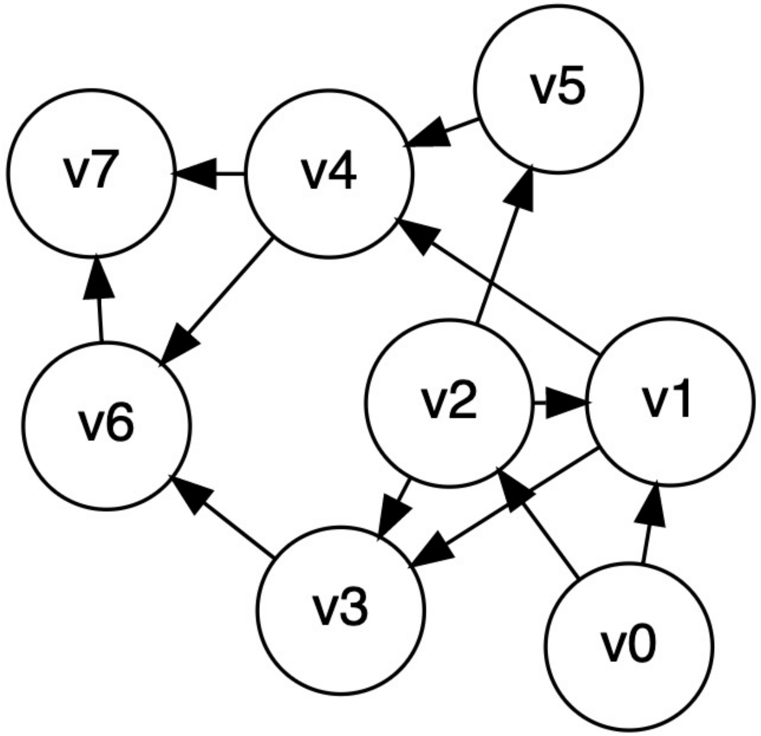
- Question 1: Finding back edges for an undirected graph is not **quite** this simple:
  - The parent node of the current node is gray
  - Not a cycle, is it? It's the same edge you just traversed
  - Question: how would you modify our code to recognize this?
- Question 2:
  - In digraph, how could you modify the code to distinguish cross edges from descendant edges?
  - Need to record the "time" at which a node was discovered (set to "gray") and finished (set to "black")
  - Also, have a "time counter", say, ctr
    - Set d[v] = ctr++ as discovery time
    - Set f[v] = ctr++ as finish time

Time Complexity of DFS

- For a digraph having V vertices and E edges
  - Each edge is processed once in the while loop of dfs_recurse() 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 2*E nodes in all the lists
  - The non-recursive dfs_sweep() 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
    - Note: This is considered "linear" for graphs since there are two size parameters for graphs.
  - Extra space is used for color array.
    - Space complexity is $\theta(V)$

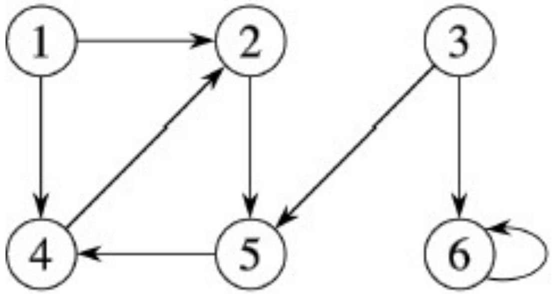
depth-first search, example

- Let’s start at V0

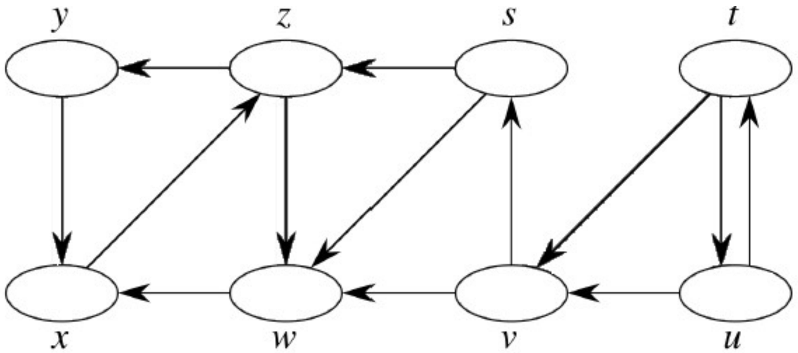


DFS Examples

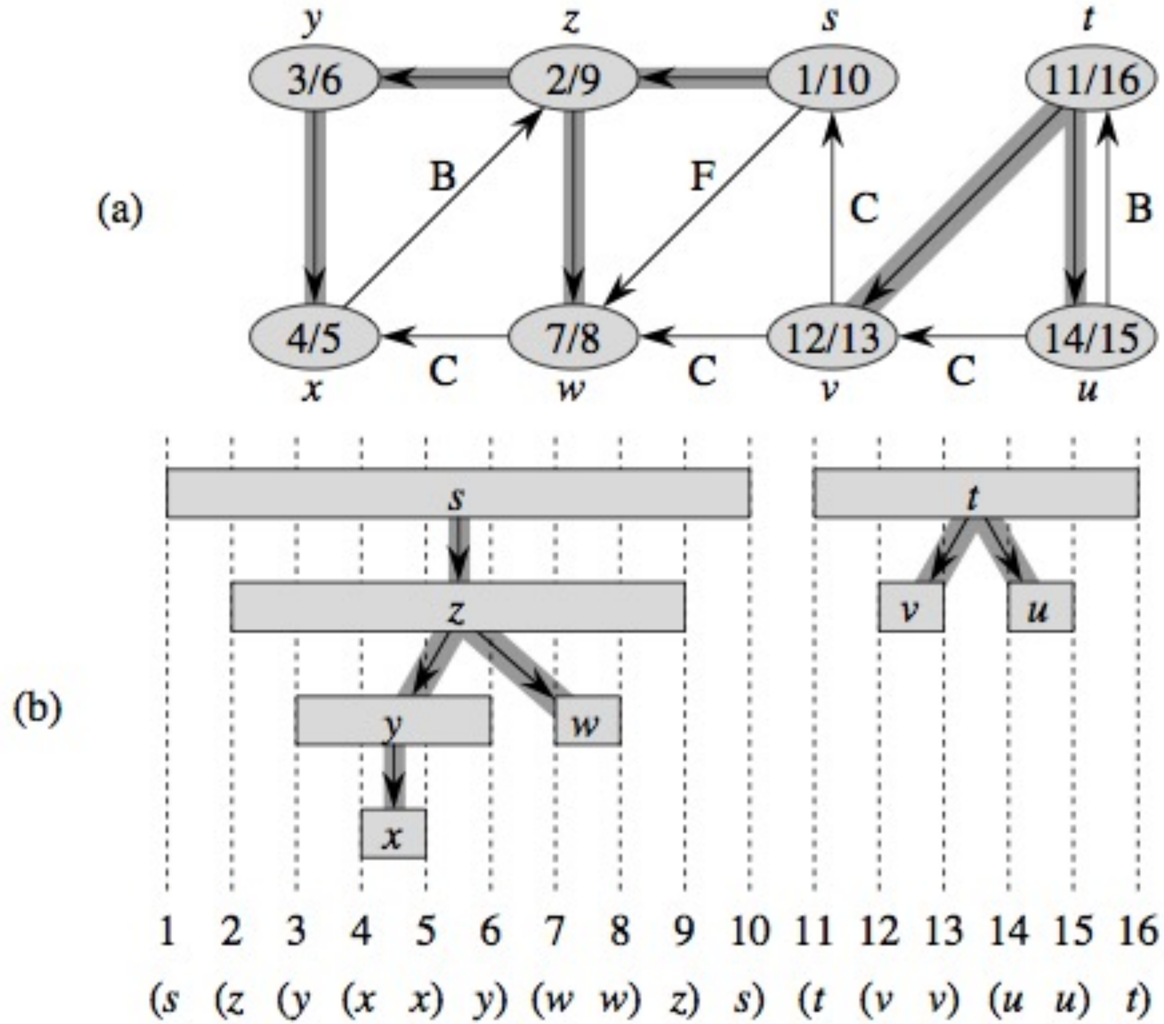
- Source vertex: 1



- Source vertex: s

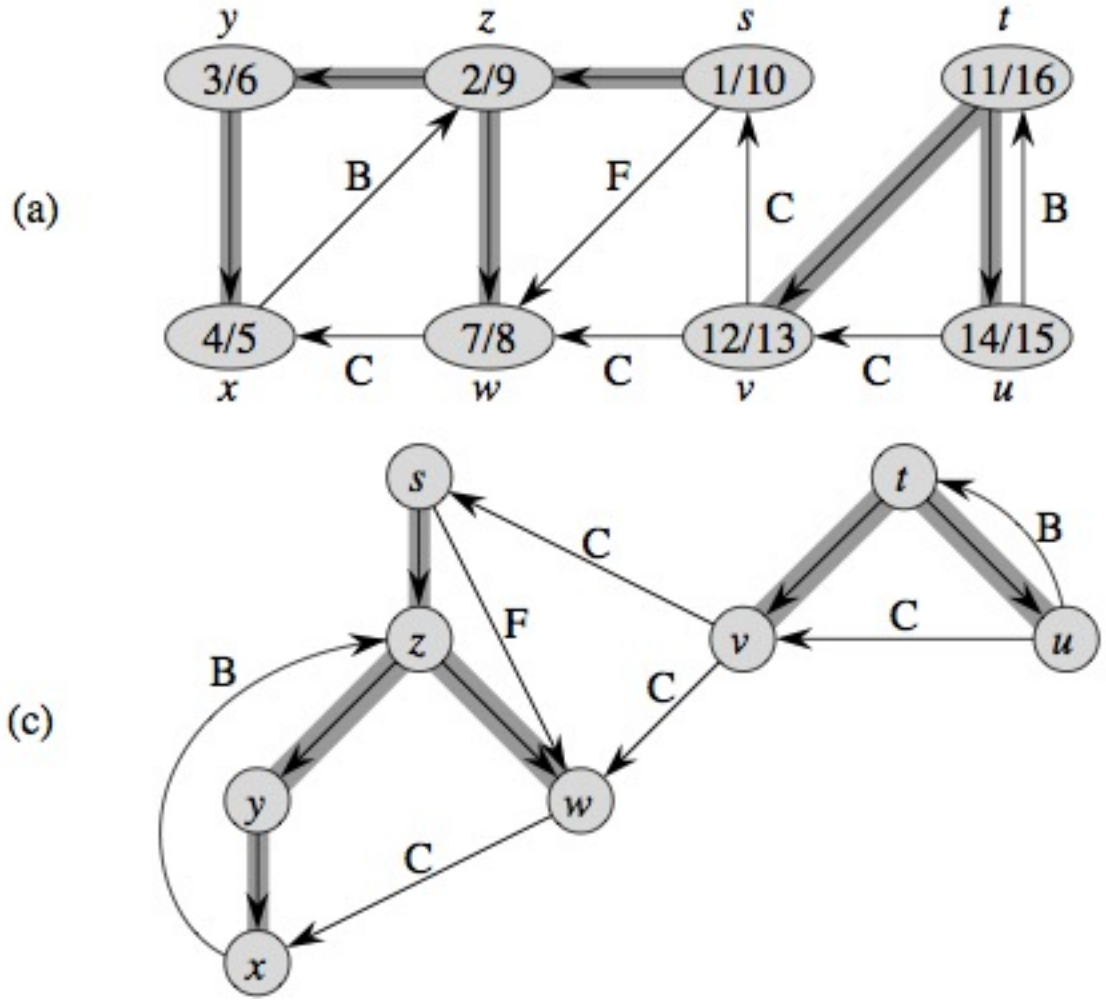


Properties of DFS Search, DFS Trees



- "Parentheses Structure". See pp. 606-609

Properties of DFS Search, DFS Trees



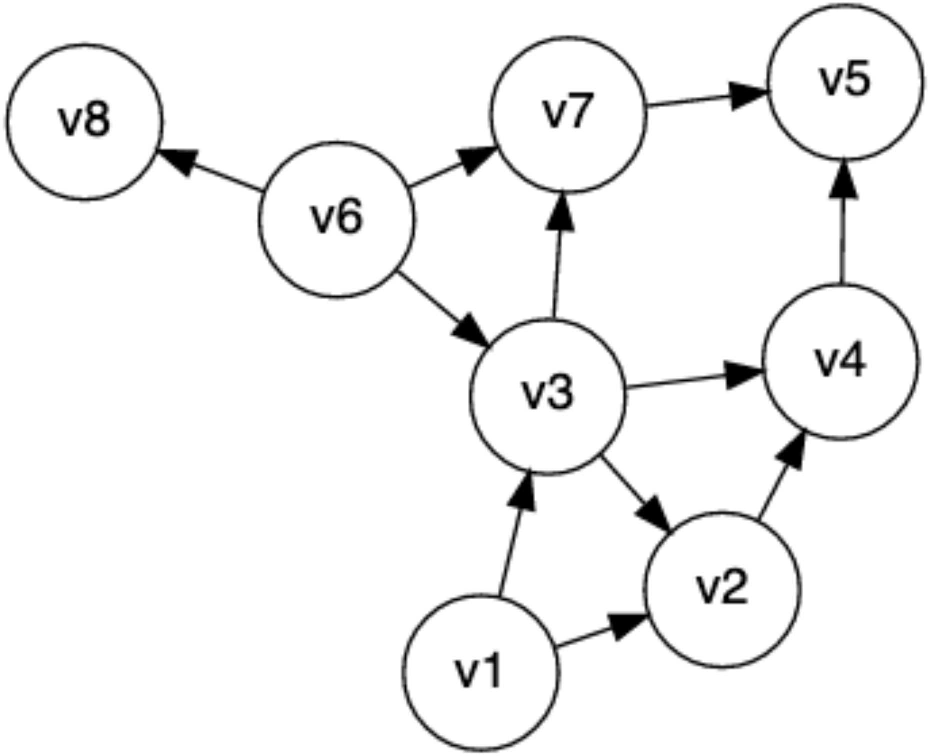
- Edge Classification. See pp. 606-609

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.

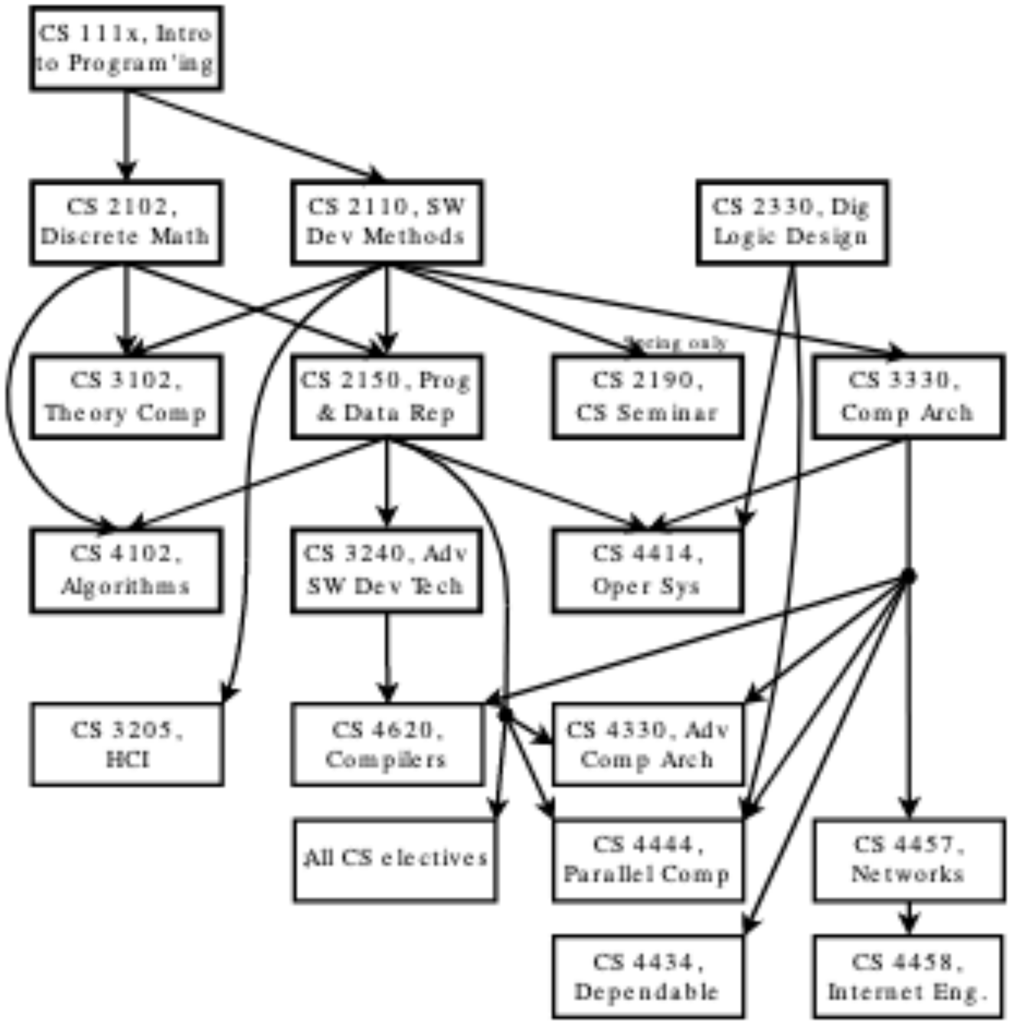


- One valid topological sort is:

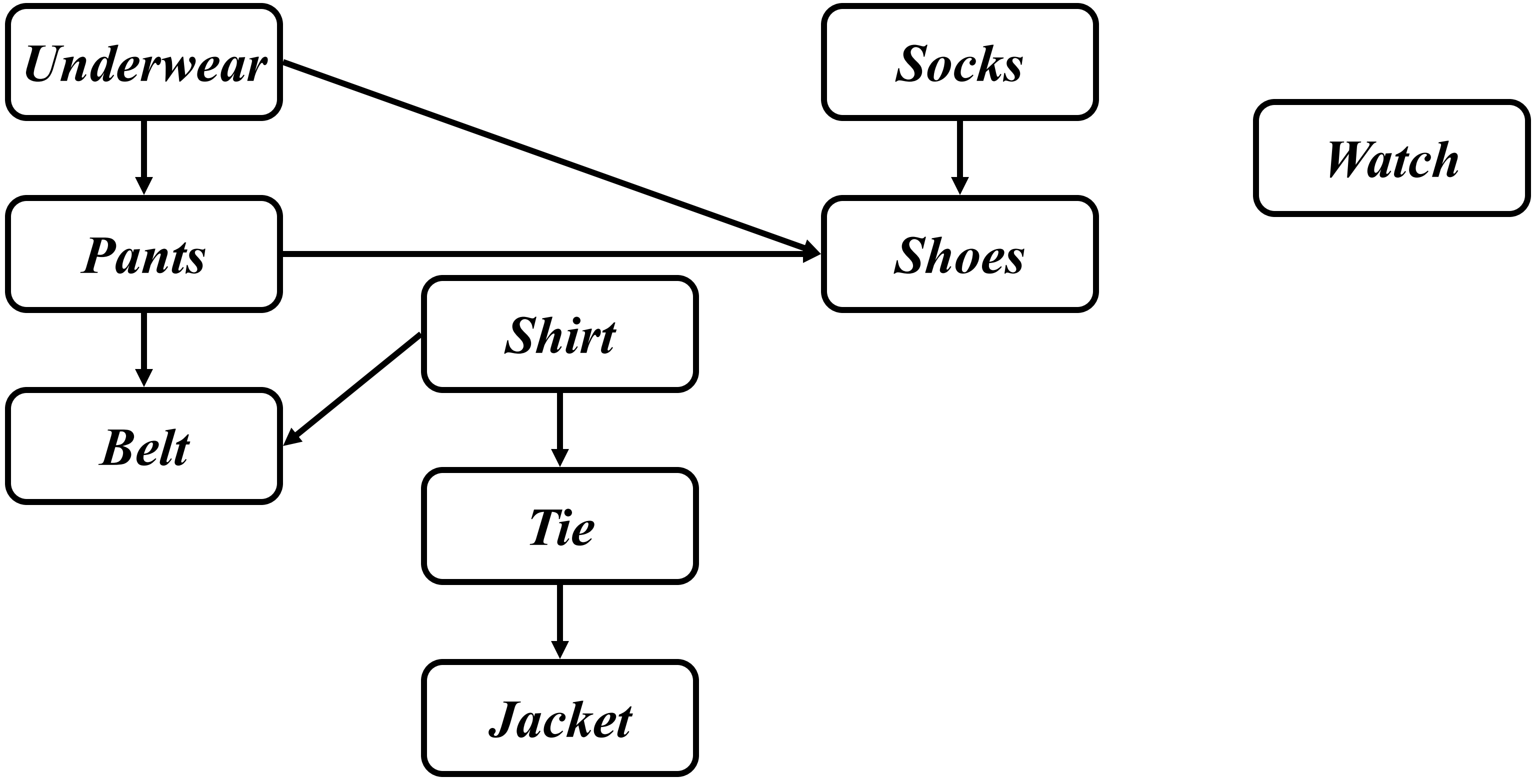
  V1 V6 V8 V3 V2 V7 V4 V5

Topological Sort

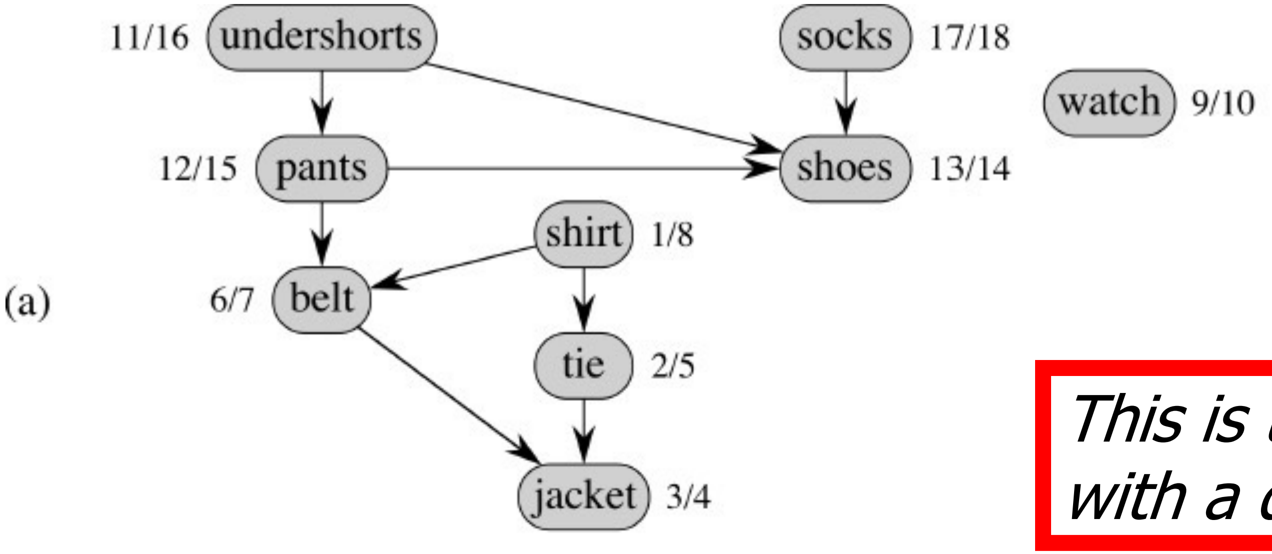
- What are allowable orderings I can take all these CS classes?
  - Note there are many possible orderings
  - Unlike sorting a list



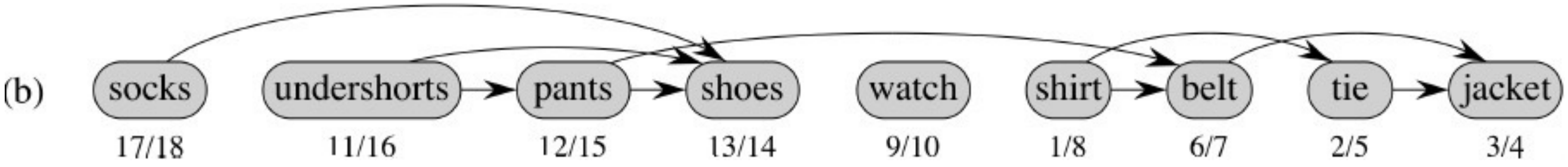
Getting Dressed



We Can Use DFS and Finish Times



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

Topological Sort Algorithm

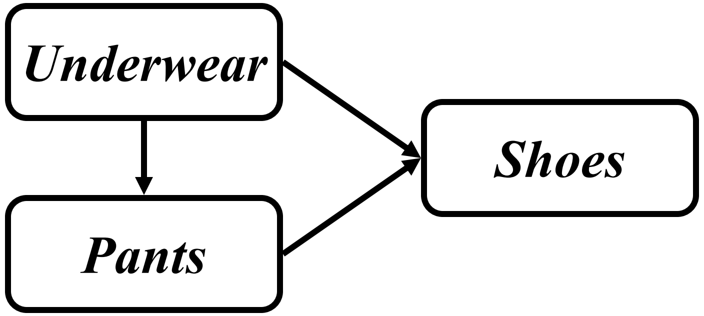
```
DFS-VISIT(G, u)
time = time + 1  // white vertex u has just been discovered
u.d = time  // discovery time of u
u.color = GRAY  // mark as seen
for each v in G.Adj[u]  // explore edge (u, v)
    if v.color == WHITE   // if unseen
        v.π = u
        DFS-VISIT(G, v)  // explore paths out of v (i.e., go "deeper")
u.color = BLACK  // u is finished
time = time + 1
u.f = time  // finish time of u
toposort-list.prepend(u)
```

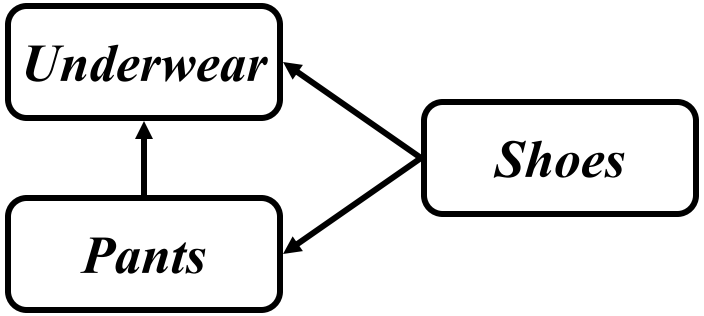
Forward vs. Reverse

- 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

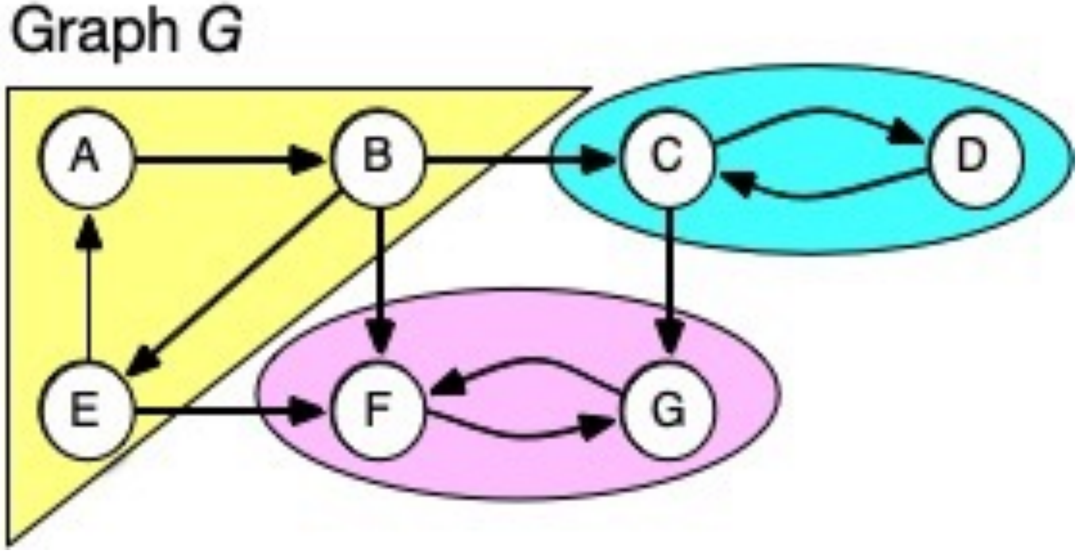
Readings: CLRS 22.5, but you can ignore the proof-y parts

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

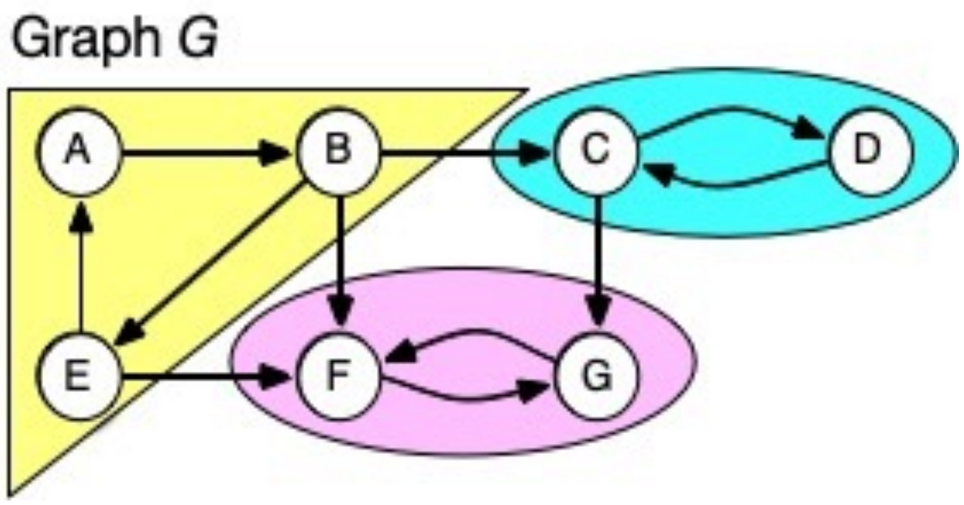
SCC Example

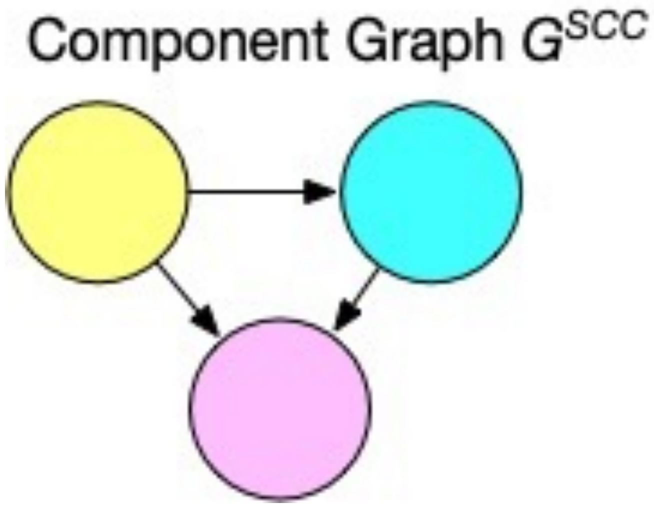
- Example: digraph below has 3 SCCs
  - Note here each SCC has a cycle. (Possible to have a single-node SCC.)
  - Note connections to other SCCs, but no path leaves a SCC and comes back
  - Note there's a unique set of SCCs for a given digraph



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





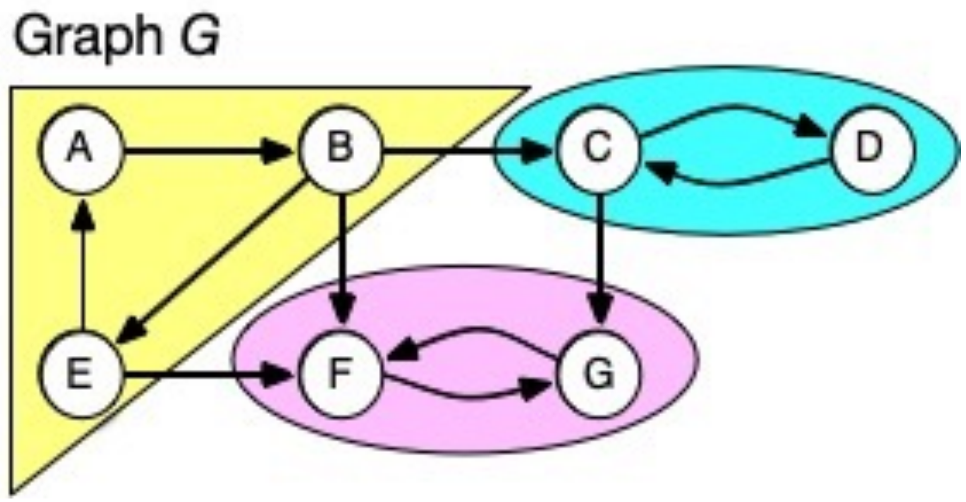
How to Decompose Graph into SCCs

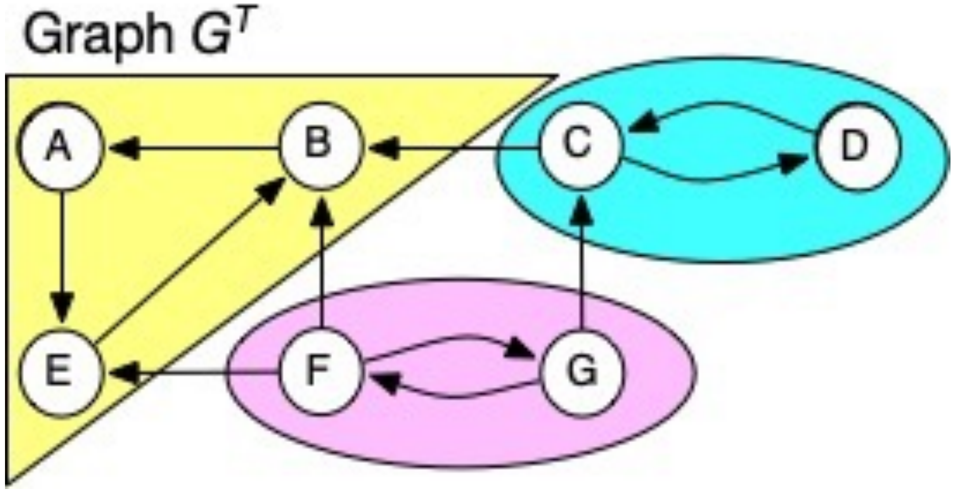
- Several algorithms do this using DFS
- We'll use CLRS's choice (by Kosaraju and Sharir)
- Algorithm is:
  1. Call *DFS-sweep(G)* to find finishing times *u.f* for each vertex *u* in *G*.
  2. Compute G^T, the transpose of diagraph *G*.

     (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?

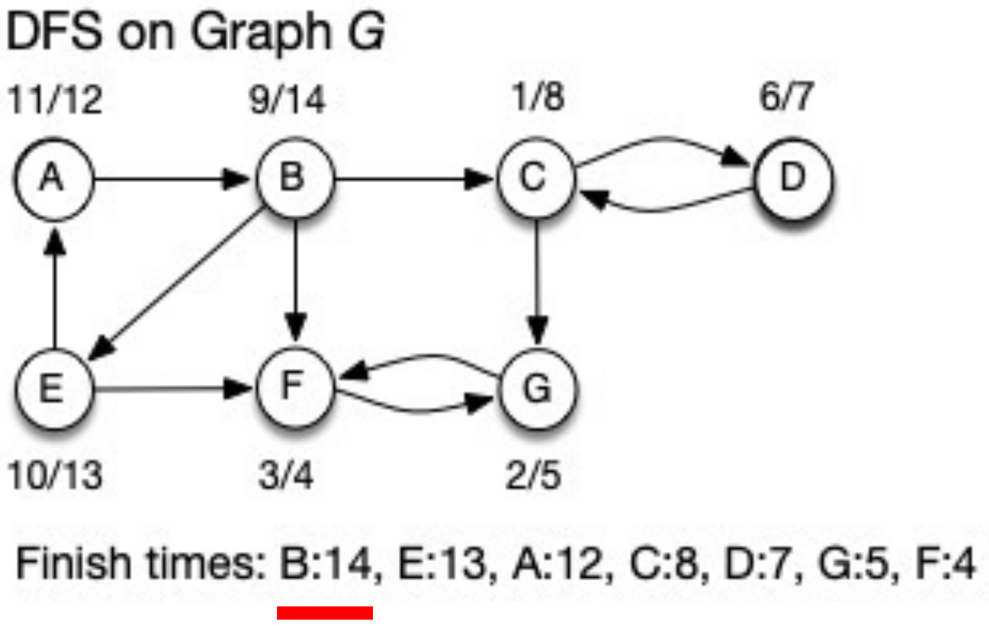
- If we call DFS on a node in an SCC, it will visit all nodes in that SCC
  - But it could leave the SCC and find other nodes ☹
  - Could we prevent that somehow?
- Note that a digraph and its transpose have the same SCCs
  - Maybe we can use the fact that edge-directions are reversed in G^T to stop DFS from leaving an SCC?
  - But this depends on the order you choose vertices to do *DFS-sweep()* in G^T

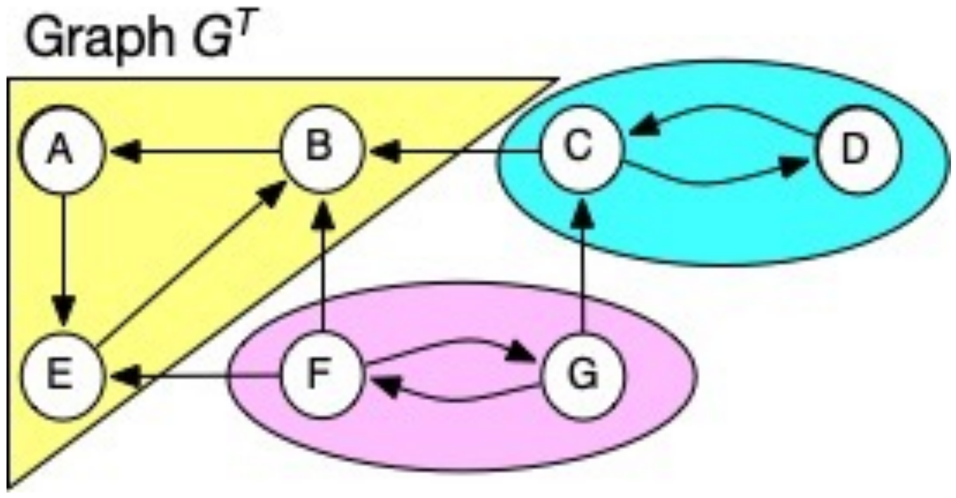




Why Do We Care About Finish Times?

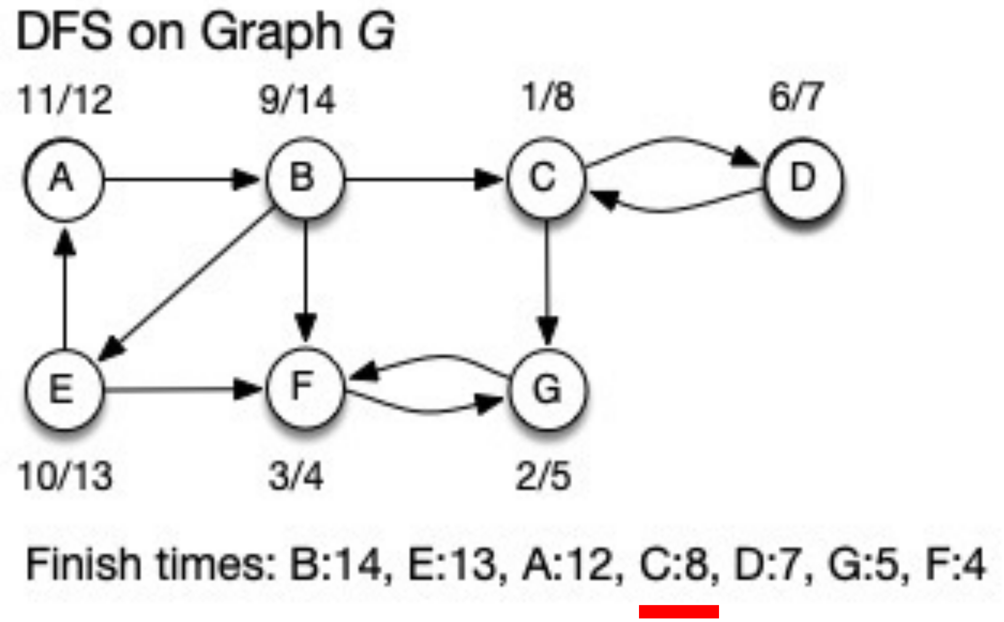
- Our algorithm first finds DFS finish times in G
- Then calls recursive DFS in transpose from vertex with largest finish time (here, B)
  - Reversed edges in G^T stop it visiting nodes in other SCCs

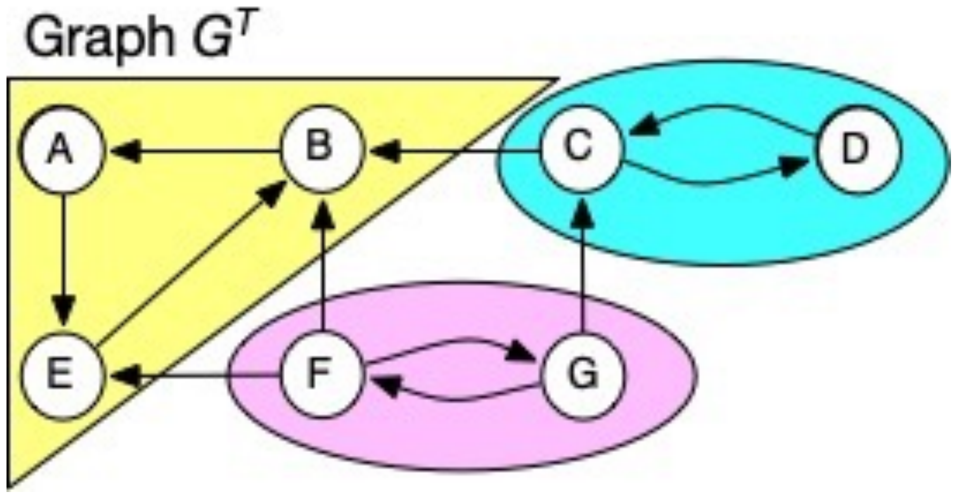




Why Do We Care About Finish Times?

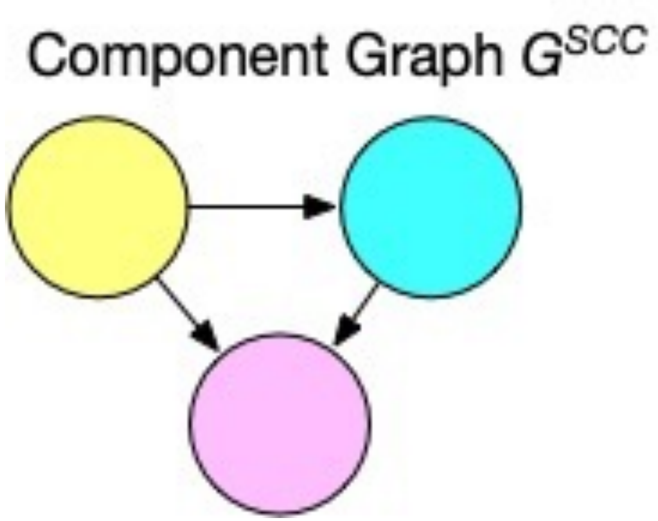
- After recursive DFS <u>in transpose</u> finds SCC containing B, next DFS will start from C
  - – Nodes in previously found SCC(s) have been visited
  - – Reversed edges in G^T stop it visiting nodes in SCCs yet to be found

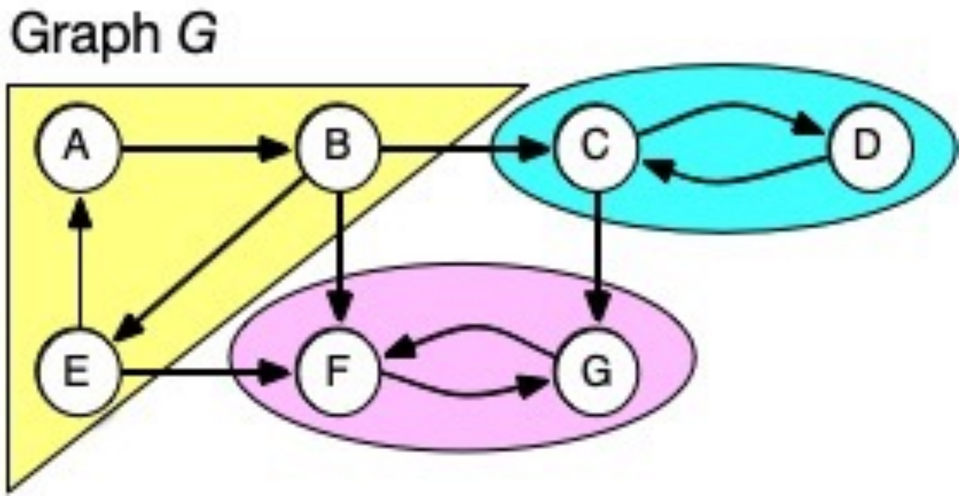




Ties to Topological Sorting

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