COMP 251

Algorithms & Data Structures (Winter 2022)

Graphs – TS & SCC

School of Computer Science
McGill University

Slides of (Comp321 ,2021), Langer (2014), Kleinberg & Tardos, 2005 & Cormen et al., 2009, Jaehyun Park' slides CS 97SI, Top-coder tutorials, T-414-AFLV Course, Programming Challenges books, slides from D. Plaisted (UNC) and Comp251-Fall McGill.

Announcements

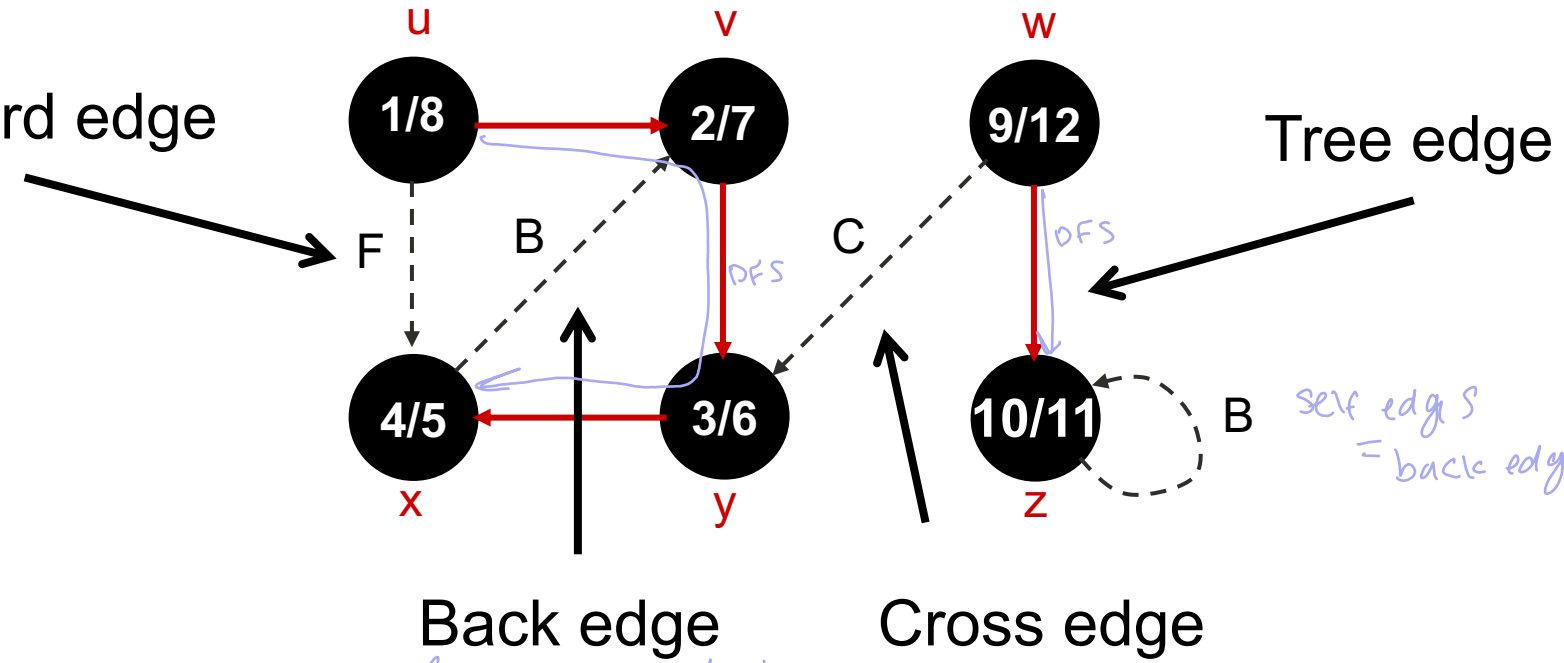
- Proof Activity.
  - #523 (Latex Tutorial)
  - #453 (submit only one proof)
- Assignment 2.
  - #503 (extension)
  - #505 (extra OH)
- Assignment 3.
  - Will be published on Friday.
- Midterm.
  - #532 (logistics)
  - (test crowdmark)

Outline

- Graphs.
  - Introduction.
  - Topological Sort. / Strong Connected Components
  - Network Flow 1.
  - Network Flow 2.
  - Shortest Path.
  - Minimum Spanning Trees.
  - Bipartite Graphs.

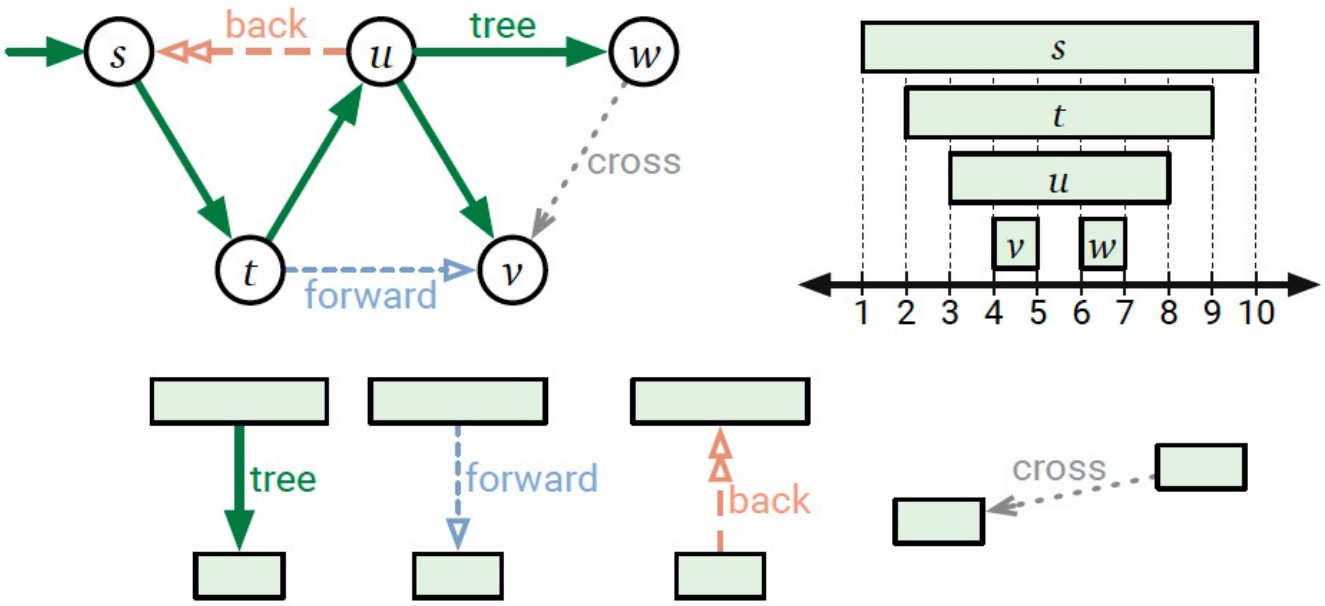
DFS – Classification of Edges

- **Tree edge:** in the depth-first forest. Found by exploring (u, v).
- **Back edge:** (u, v), where u is a descendant of v (in the depth-first tree).
- **Forward edge:** (u, v), where v is a descendant of u, but not a tree edge.
- **Cross edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.



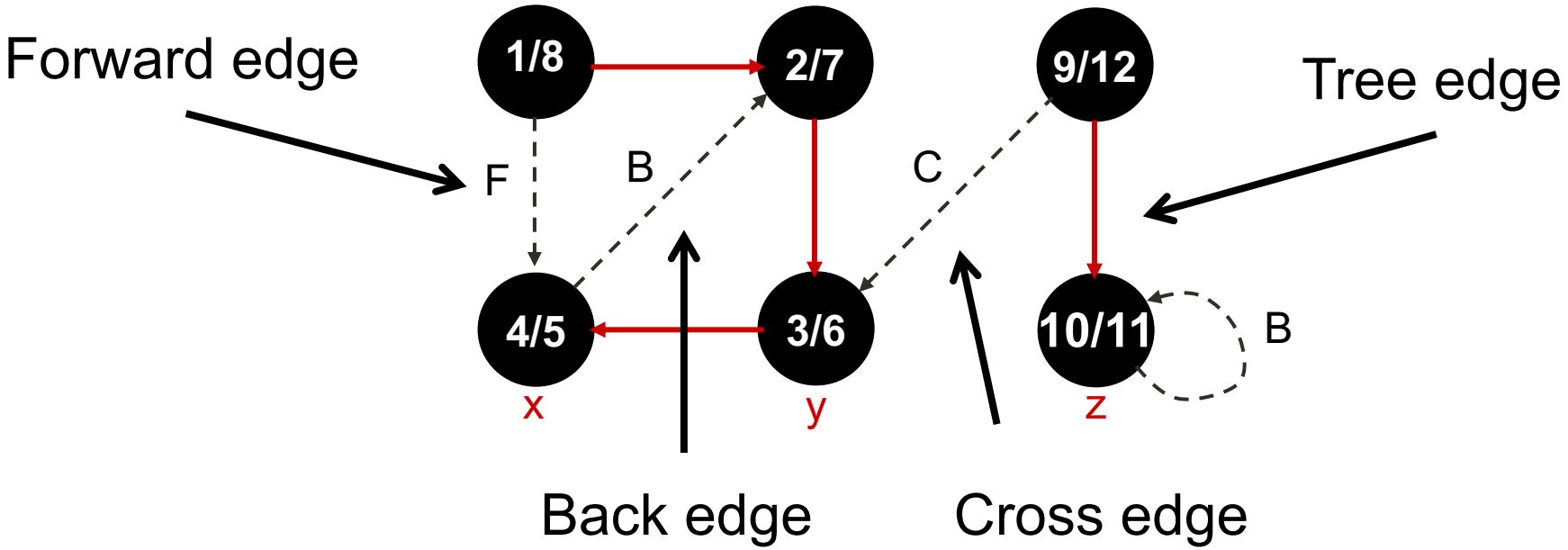
DFS – Classification of Edges

- **Tree edge:** in the depth-first forest. Found by exploring (*u, v*).
- **Back edge:** (*u, v*), where *u* is a descendant of *v* (in the depth-first tree).
- **Forward edge:** (*u, v*), where *v* is a descendant of *u*, but not a tree edge.
- **Cross edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.



DFS – Classification of Edges

- Edge type for edge (u, v) can be identified when it is first explored by DFS.
- Identification is based on the color of v.
  - White – tree edge, Gray – back edge, Black – forward or cross edge.



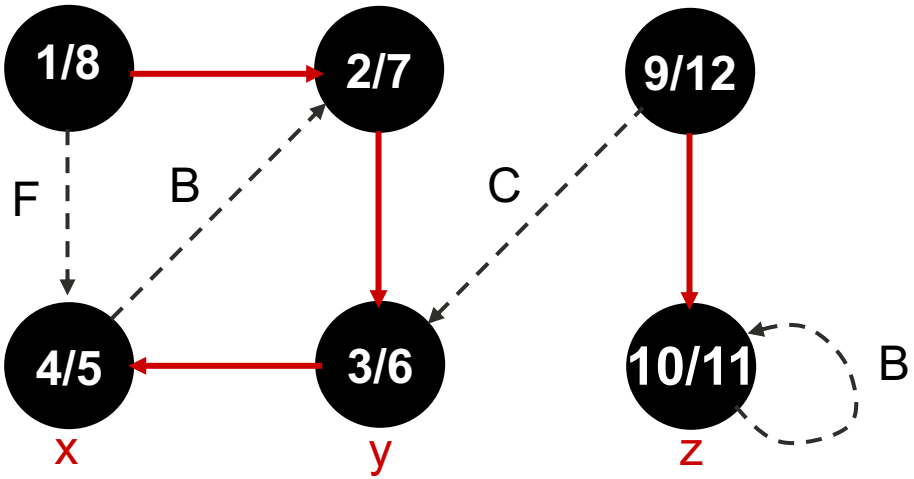
DFS – Classification of Edges - Comments

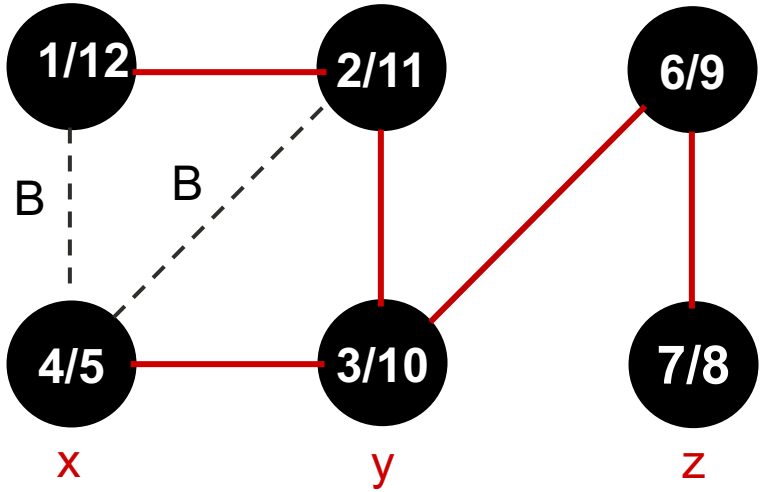
- The actual classification of edges depends on the order in which DFS considers vertices and the order in which DFS considers the edges leaving each vertex.
- An undirected graph may entail some ambiguity in how we classify edges, since (u,v) and (v,u) are really the same edge.
  - we classify the edge as the first type in the classification list that applies.
  - we classify the edge according to whichever of (u,v) or (v,u) the search encounters first.

Theorem
In DFS of a connected undirected graph, we get only tree and back edges. No forward or cross edges.

DFS – Classification of Edges - Comments

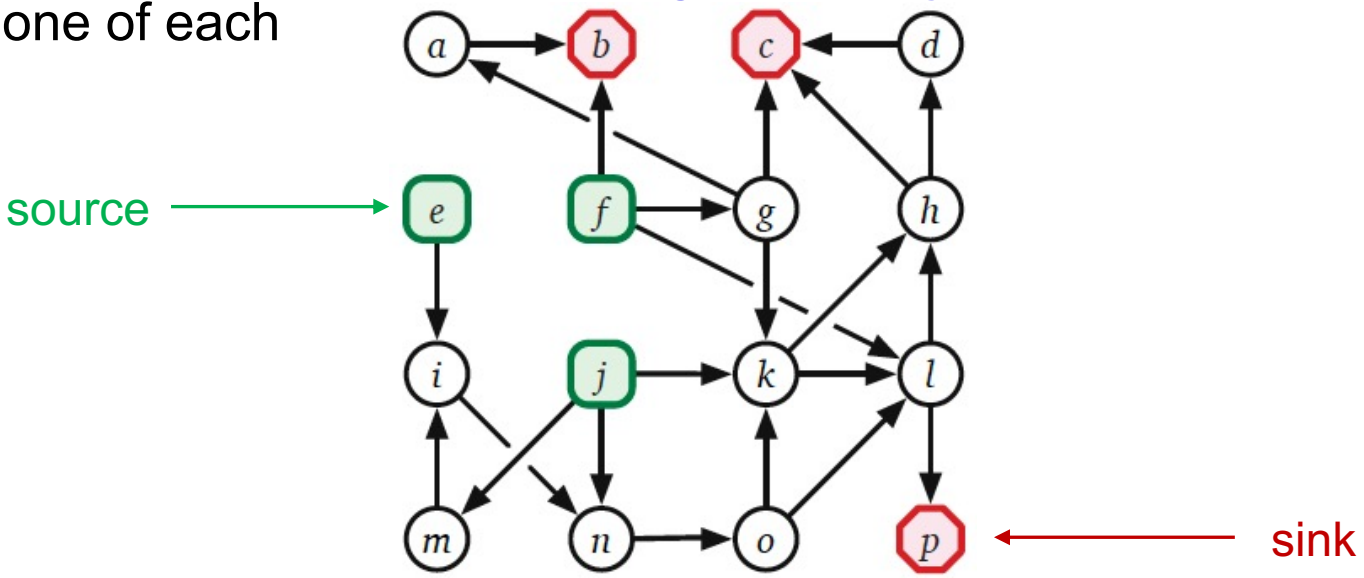
Theorem
In DFS of a connected undirected graph, we get only tree and back edges. No forward or cross edges.





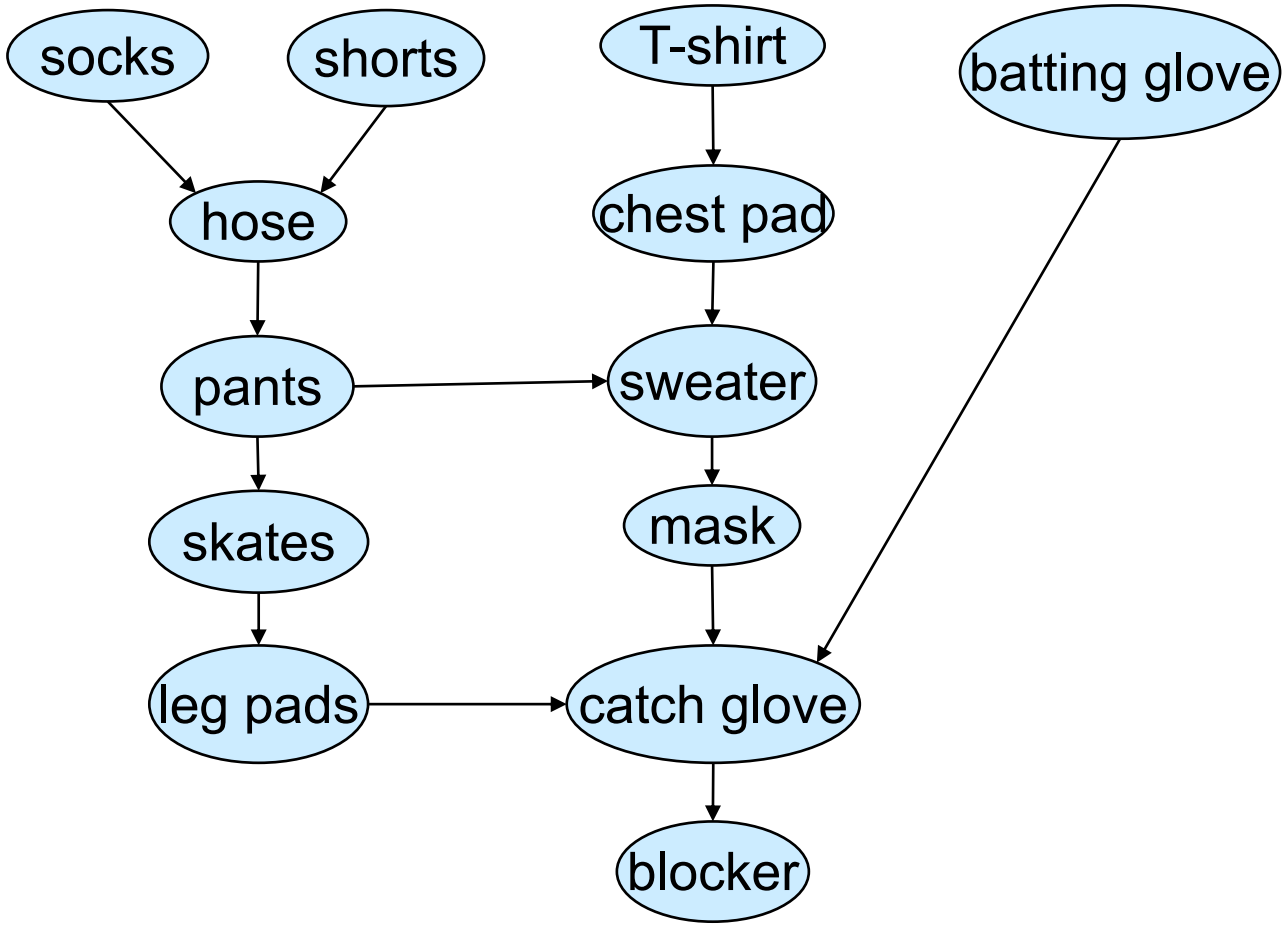
Directed Acyclic Graph

- DAG – Directed graph with no cycles.
- DAGs can be used to encode precedence relations or dependencies in a natural way.
  - **Source:** Any vertex in a dag that has no incoming vertices
  - **Sink:** any vertex with no outgoing edges
  - **Every DAG has at least one source and one sink**, but may have more than one of each



Directed Acyclic Graph - Example

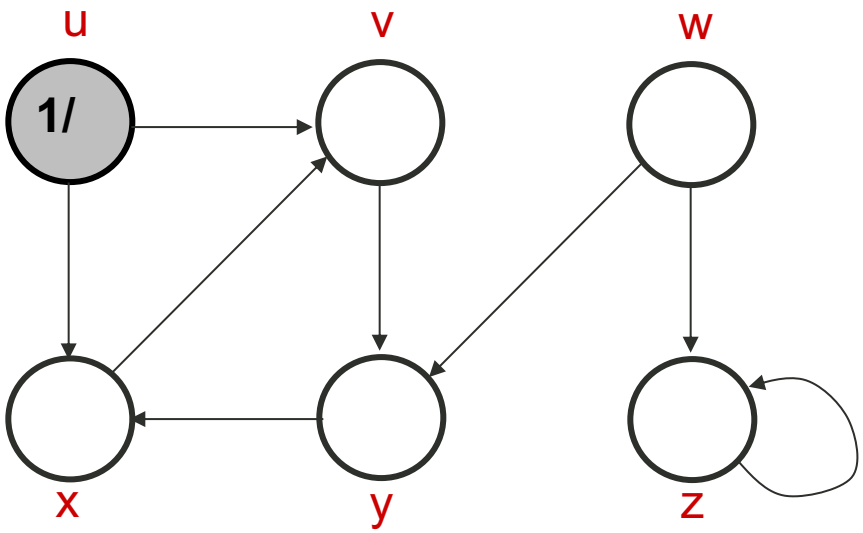
- DAG of dependencies for putting on goalie equipment.



White-path. Theorem

time of discovery

v is a descendant of *u* if and only if at time *d*[*u*], there is a path $u \rightsquigarrow v$ consisting of only white vertices (Except for *u*, which was *just* colored gray).



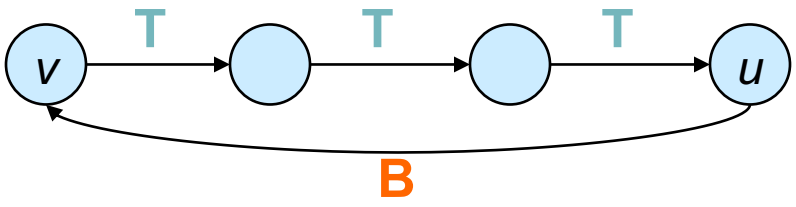
v, y, and x are descendants of u.

DAG- Characterization

Lemma 1
A directed graph *G* is acyclic iff a DFS of G yields no back edges.

Proof:

- (⇒) Show that back edge ⇒ cycle.
  - Suppose there is a back edge (*u, v*). Then *v* is ancestor of *u* in depth-first forest.
  - Therefore, there is a path $v \rightsquigarrow u$, so $v \rightsquigarrow u \rightsquigarrow v$ is a cycle.

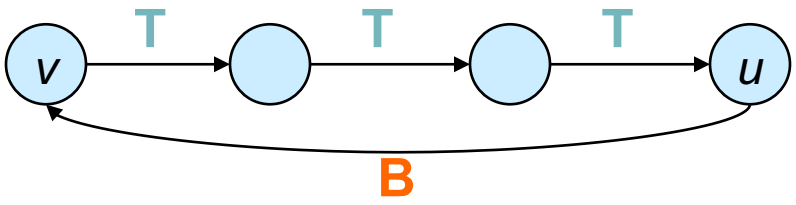


DAG- Characterization

Lemma 1
A directed graph *G* is acyclic iff a DFS of G yields no back edges.

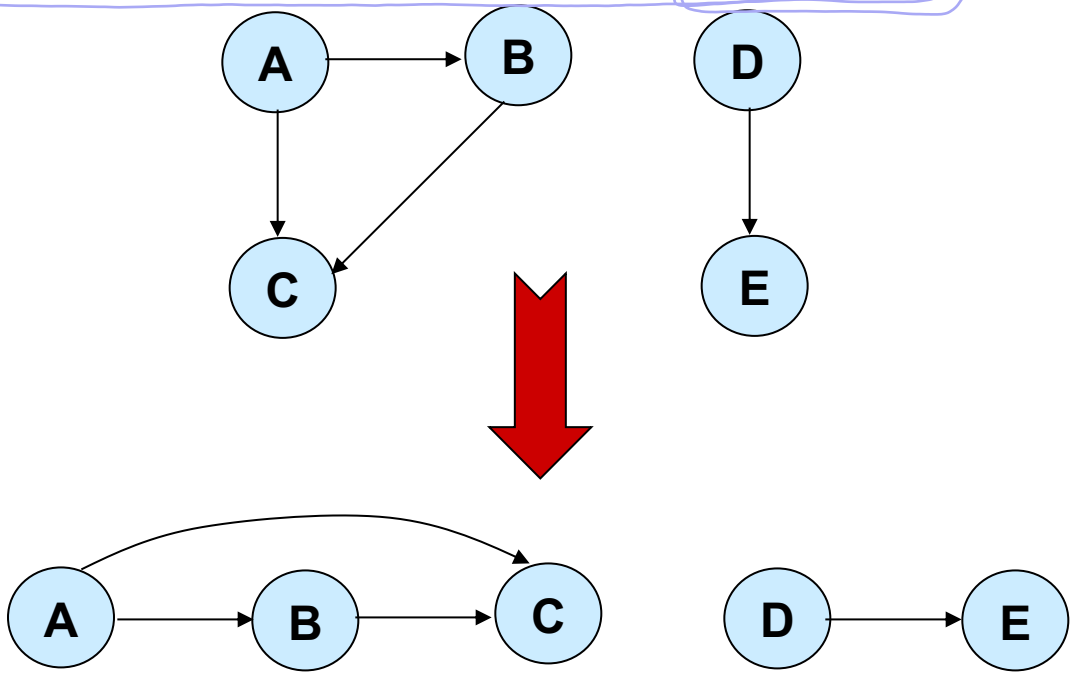
Proof (Contd.):

- ($\Leftarrow$) Show that a cycle implies a back edge.
  - Suppose that G contains a cycle c.
  - *v* : first vertex discovered in *c;* (*u, v*) : preceding edge in *c*.
  - At time *d*[*v*], vertices of *c* form a white path $v \rightsquigarrow u$.
  - By **white-path theorem**, *u* is a descendent of *v* in depth-first forest.
  - Therefore, *(u, v)* is a back edge.



Graphs – Topological Sort

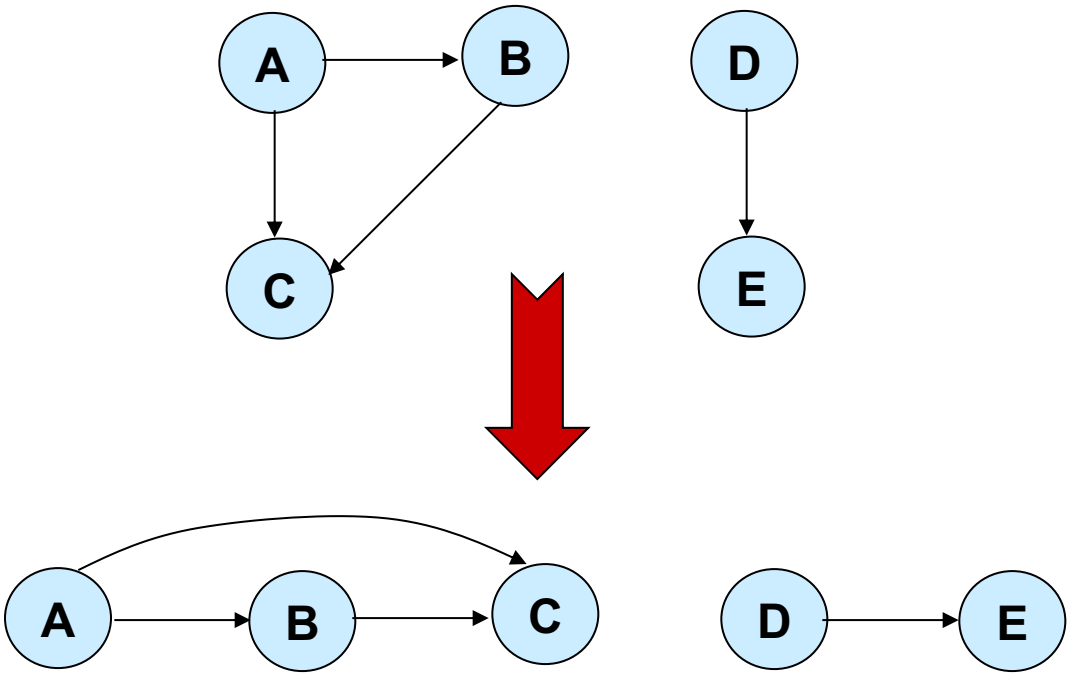
- Want to 'sort' a DAG.
  - Linear ordering of the vertices of G such that if $(u, v) \in E$, then u appears somewhere before v.
  - Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right.



Topological Sort - Algorithm

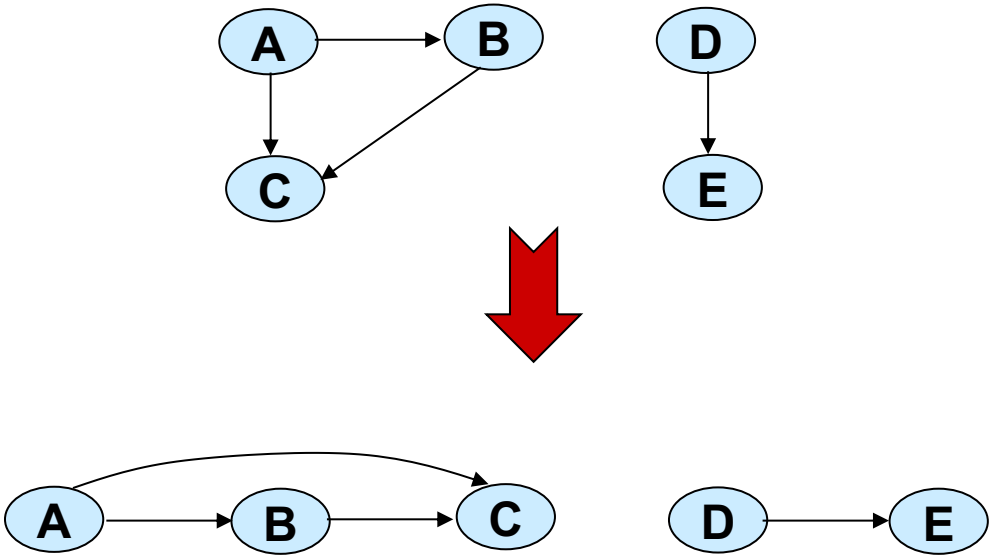
- Finding a way to get started.
  - Which node do we put at the beginning of the topological sort?

Source node!



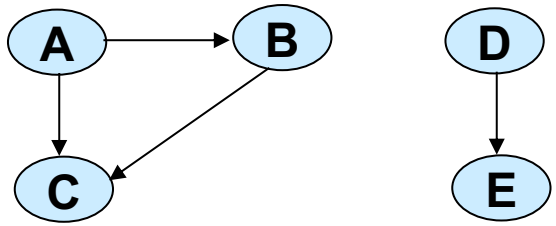
Topological Sort - Algorithm

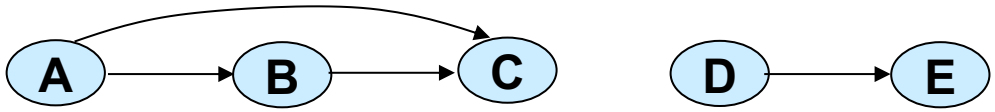
- Finding a way to get started.
  - Which node do we put at the beginning of the topological sort?
    - ANS: A source node because this node guarantees that all edges point forward.
    - Please remember that **Every DAG has at least one source.**



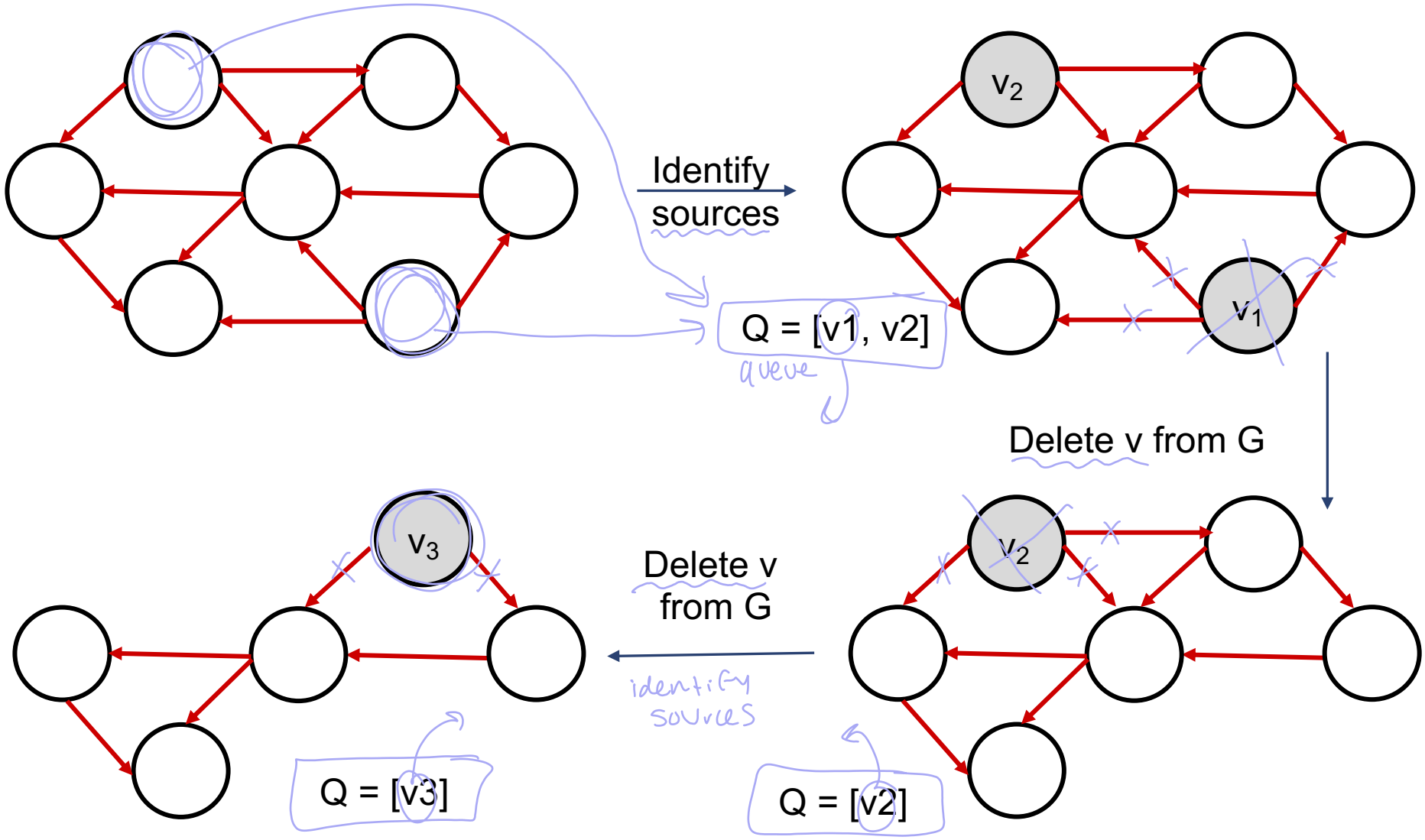
Topological Sort - Algorithm

- Find a source node v.
- Place v first in the topological ordering.
  - This is safe, since all edges out of v will point forward.
- Delete v from G.
  - This is safe, since deleting v can not create any cycles that were not there previously
- Recursively compute a topological ordering of G – {v}

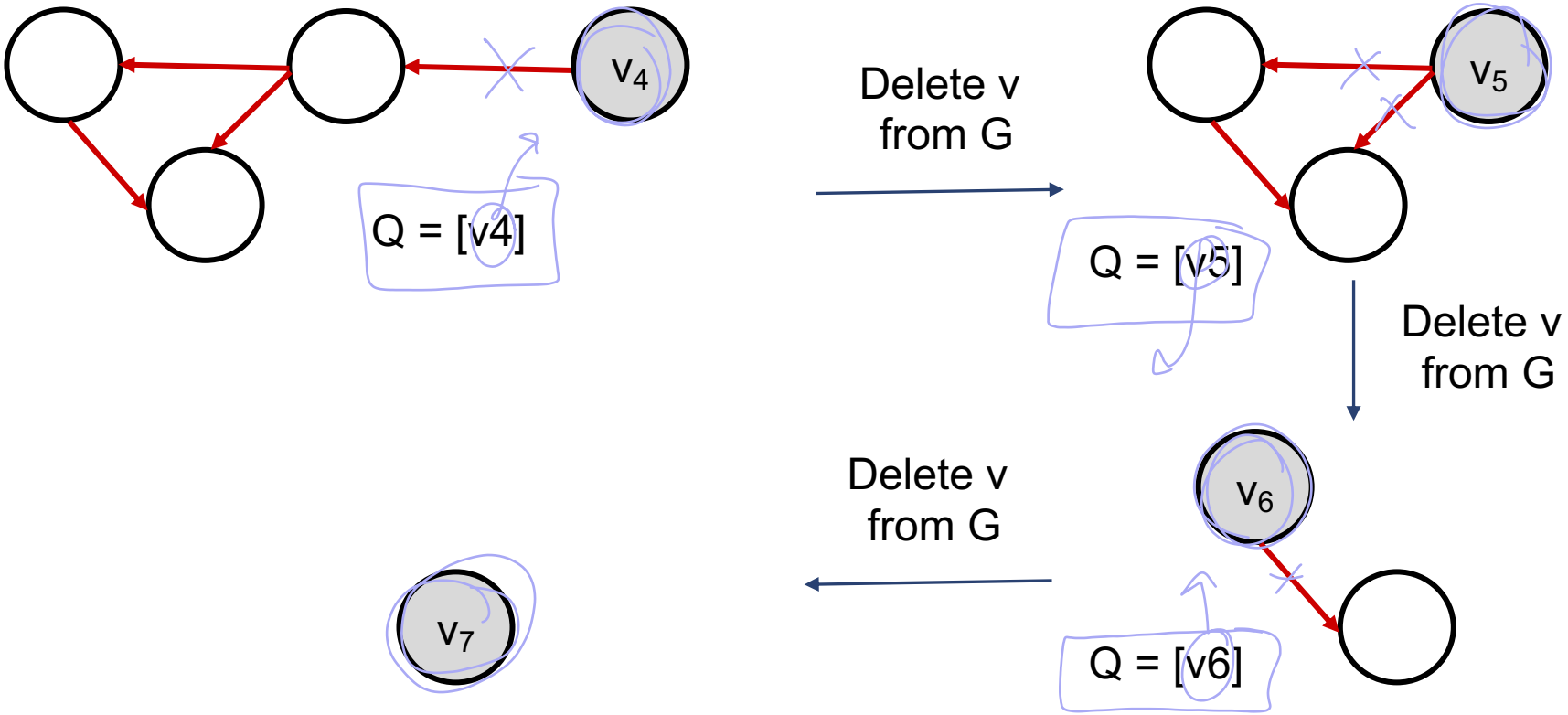




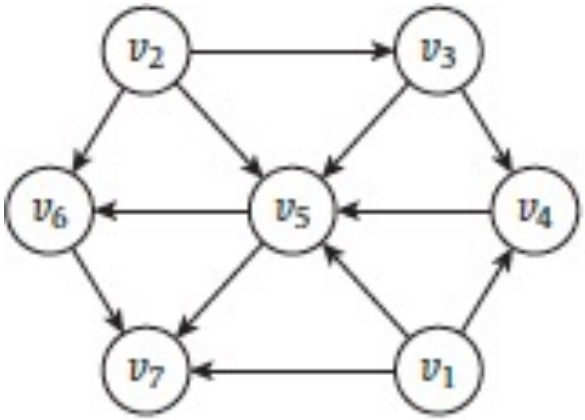
Topological Sort – Algorithm - Example

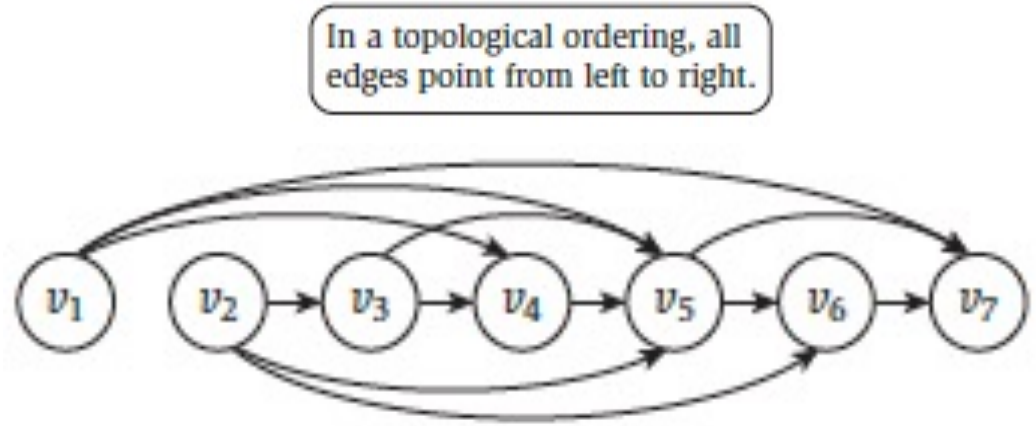


Topological Sort – Algorithm - Example



Topological Sort – Algorithm - Example





Step-1: Compute in-degree of each node → O (V + E)

Step-2: Pick all the vertices with in-degree as 0
and add them into a queue

Step-3: Remove a vertex from the queue.
Decrease in-degree by 1 for all its neighboring nodes.
If in-degree of a neighboring nodes is zero, then
add it to the queue.

} O (V + E)

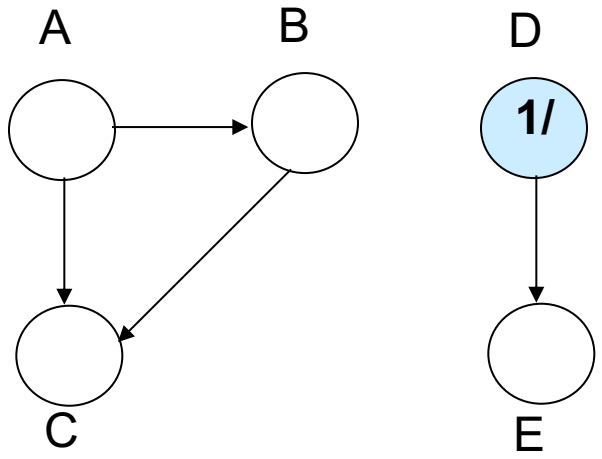
Step 5: Repeat Step 3 until the queue is empty.

Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Time: $\Theta(V + E)$.

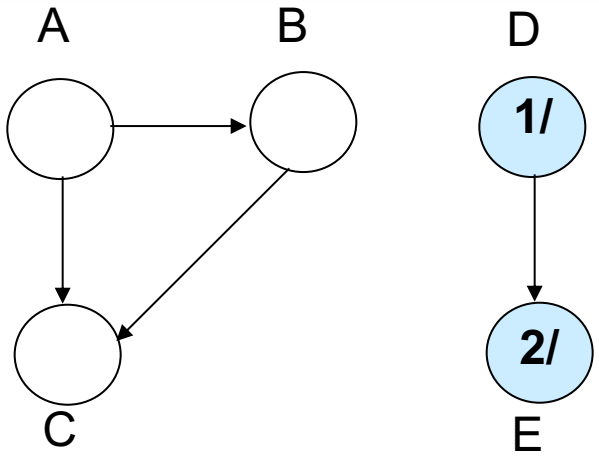


Linked List:

Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

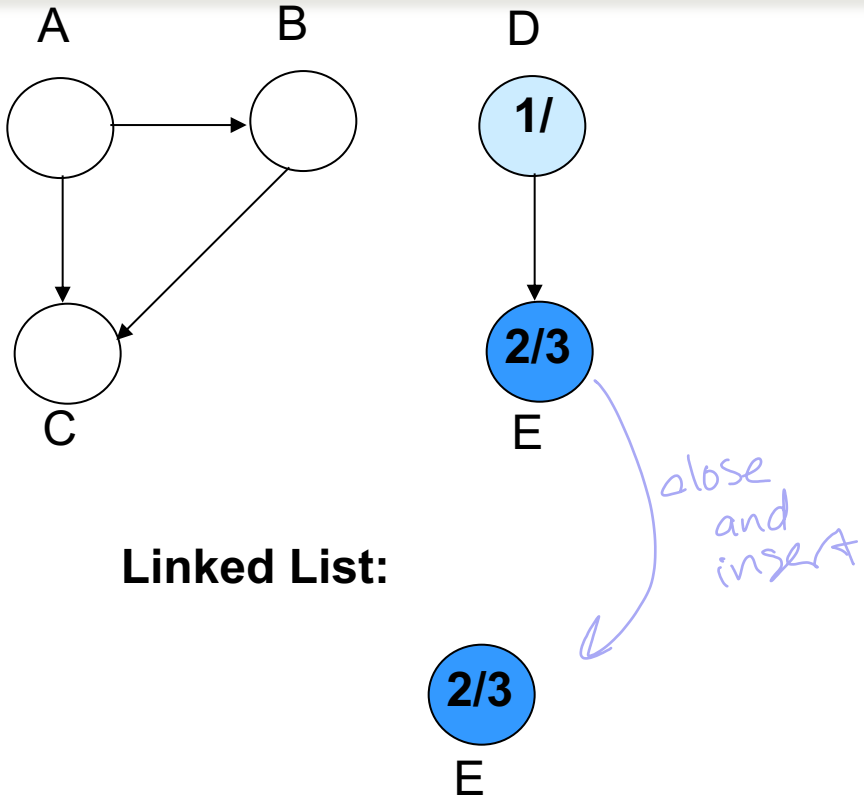


Linked List:

Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

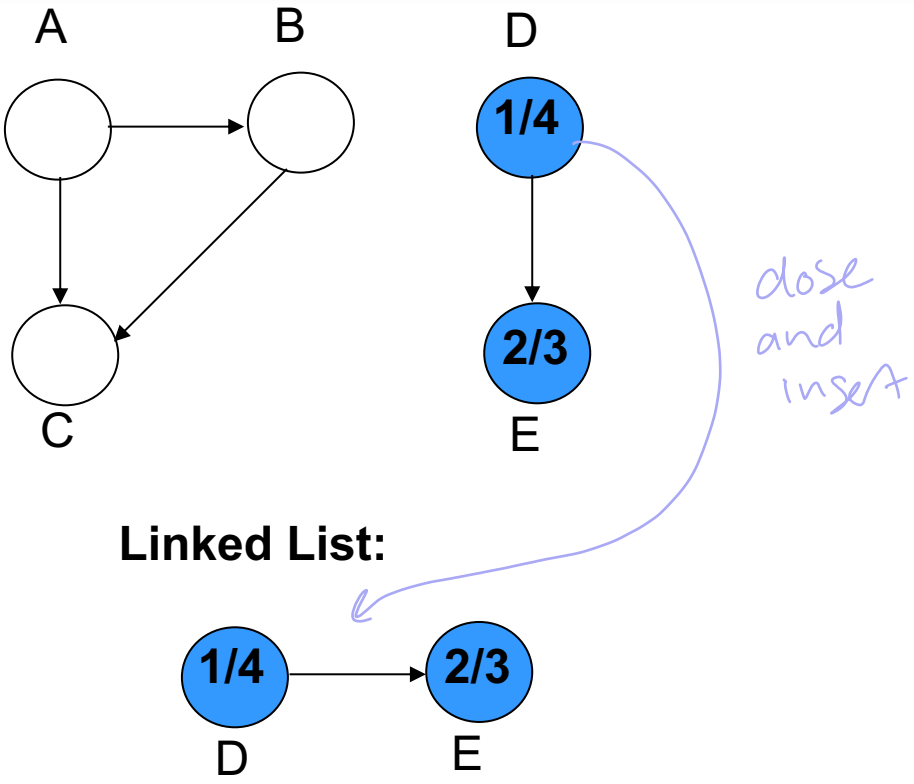
1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices



Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

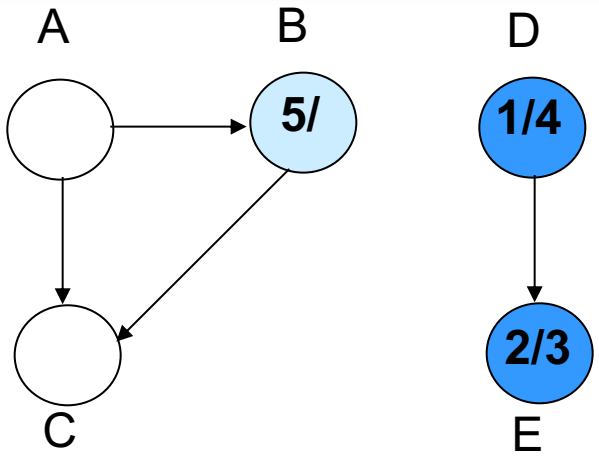
1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices



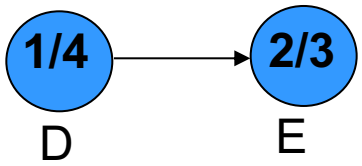
Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices



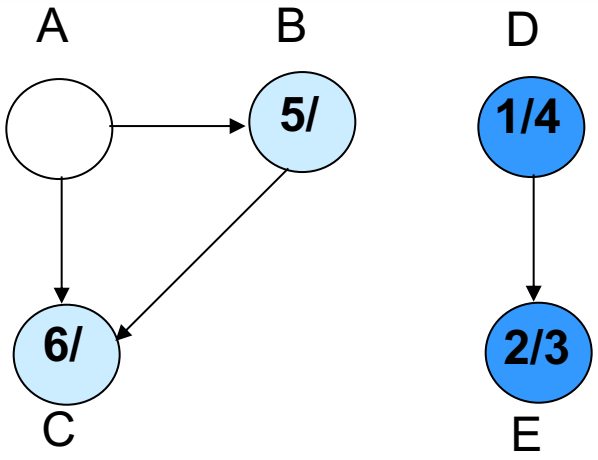
Linked List:



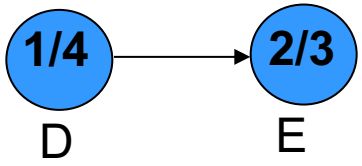
Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices



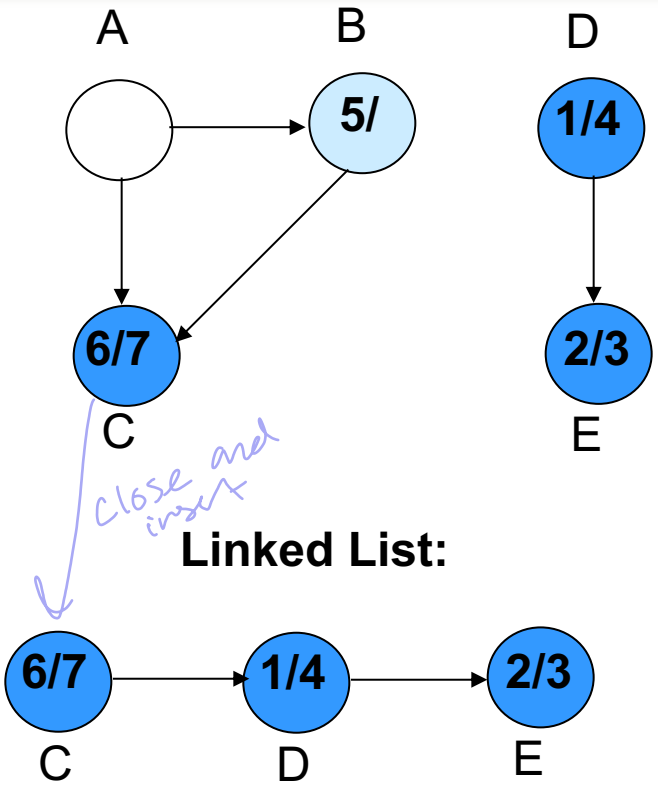
Linked List:



Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

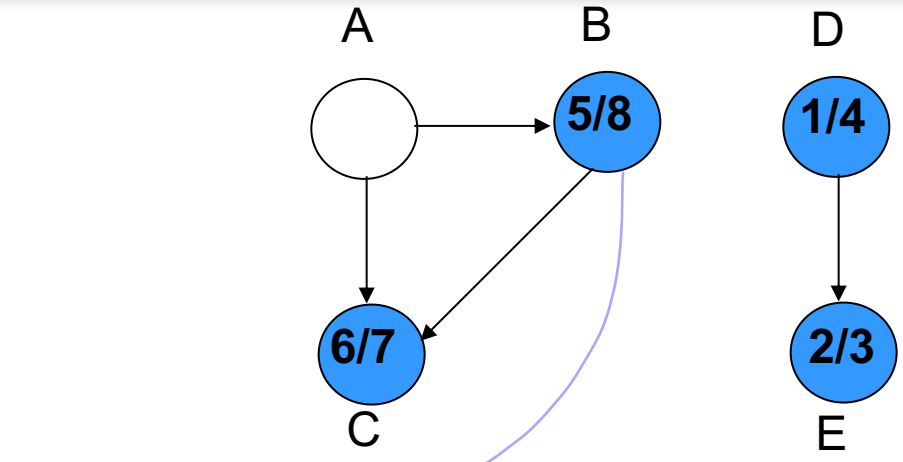
1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices



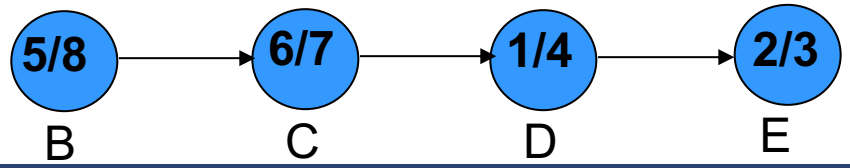
Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices



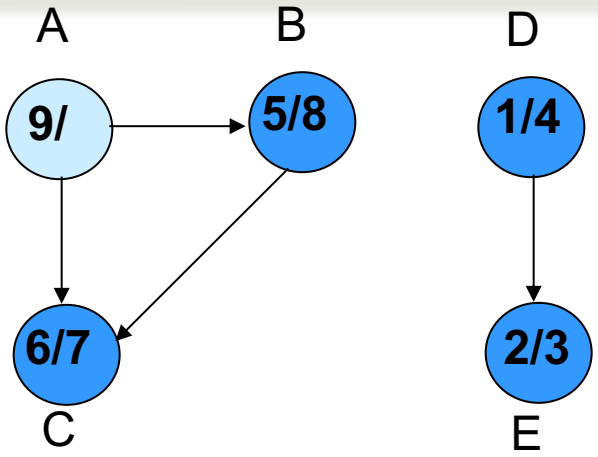
Linked List:



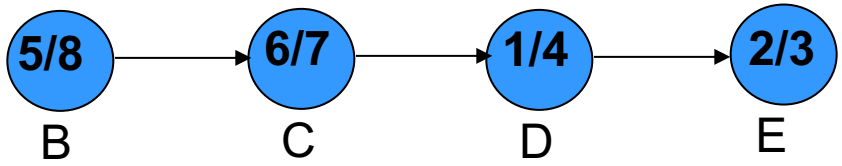
Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices



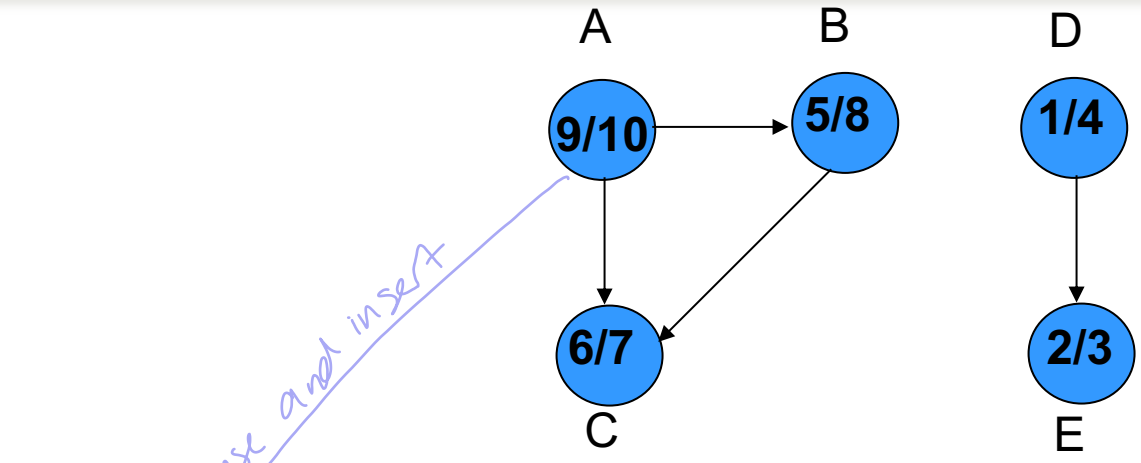
Linked List:



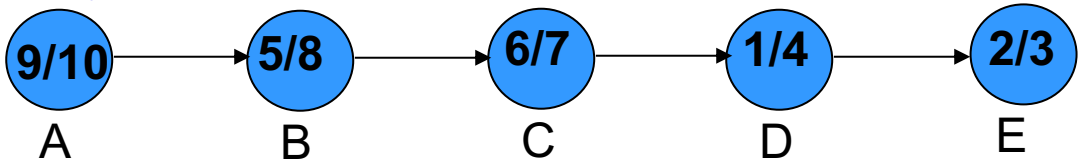
Topological Sort – Algorithm2 – Example2

Topological-Sort (*G*)

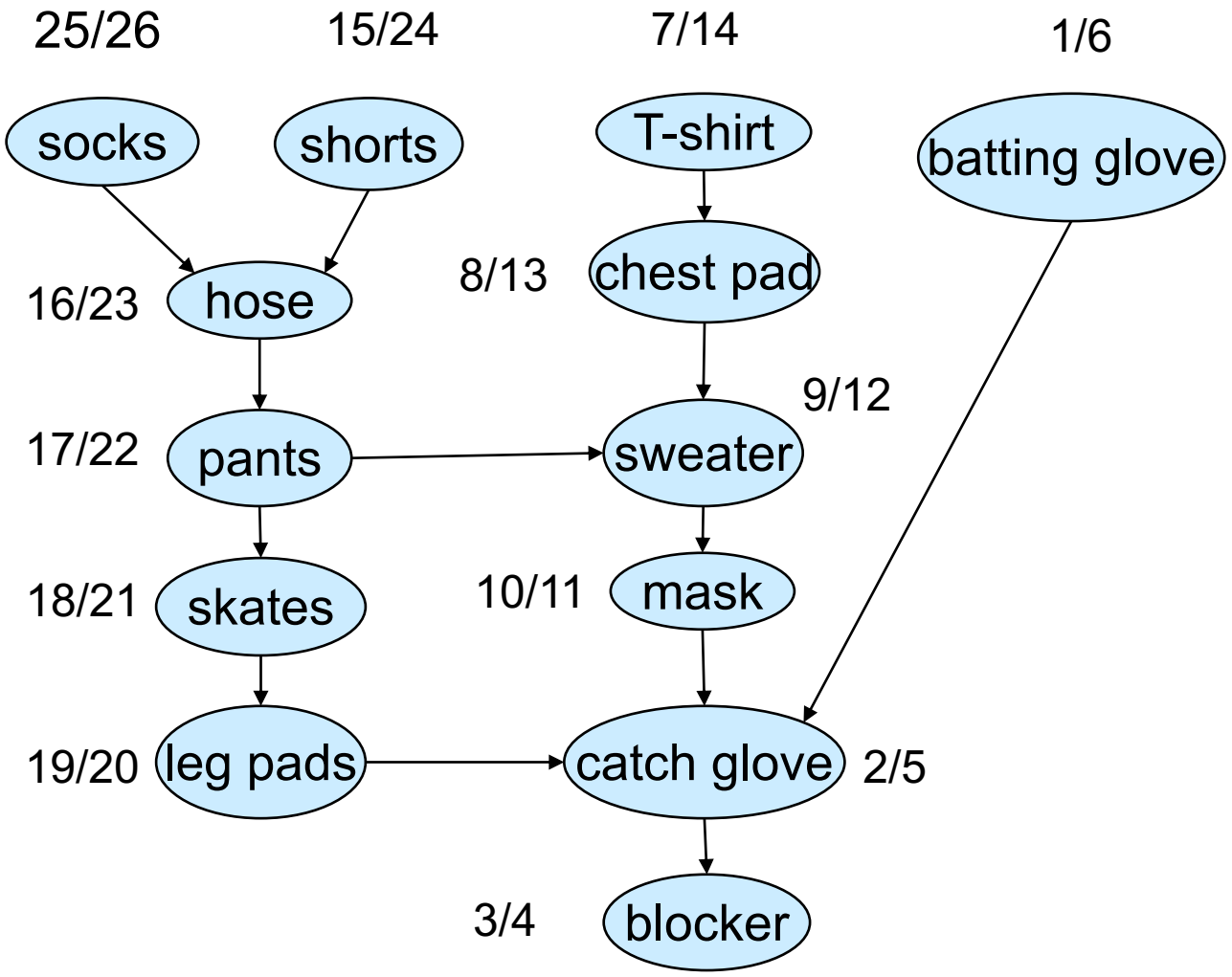
1. call DFS(*G*) to compute finishing times *f* [*v*] for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices



Linked List:



Topological Sort – Algorithm2 – Example3



26 socks
24 shorts
23 hose
22 pants
21 skates
20 leg pads
14 t-shirt
13 chest pad
12 sweater
11 mask
6 batting glove
5 catch glove
4 blocker

Topological Sort – Algorithm2 – Correctness

"Linear ordering of the vertices of G such that if $(u, v) \in E$, then u appears somewhere before v."

$\Rightarrow$ We need to show if $(u, v) \in E$, then $f[v] < f[u]$.

When we explore (u, v), what are the colors of u and v?

Assume we just discovered u, which is thus gray.

Then, what are the possible colors of v ?

- Can v be gray?
- Can v be white?
- Can v be black?

Topological Sort – Algorithm2 – Correctness

When we explore (*u, v*), what are the colors of *u* and *v*?

- Assume *u* is gray.
- Is *v* gray, too?

  No, because then *v* would be ancestor of *u*.

  $\Rightarrow$ (*u, v*) is a back edge.

  $\Rightarrow$ contradiction of **Lemma 1** (DAG has no back edges).
- Is *v* white?
  - Then becomes descendant of *u*.
  - By **parenthesis theorem**, $d[u] < d[v] < \boldsymbol{f}\,\boldsymbol{[v]} < \boldsymbol{f}\,\boldsymbol{[u]}$.
- Is *v* black?
  - Then *v* is already finished.
  - Since we are exploring (*u, v*), we have not yet finished *u*.
  - Therefore, $f[v] < f[u]$.

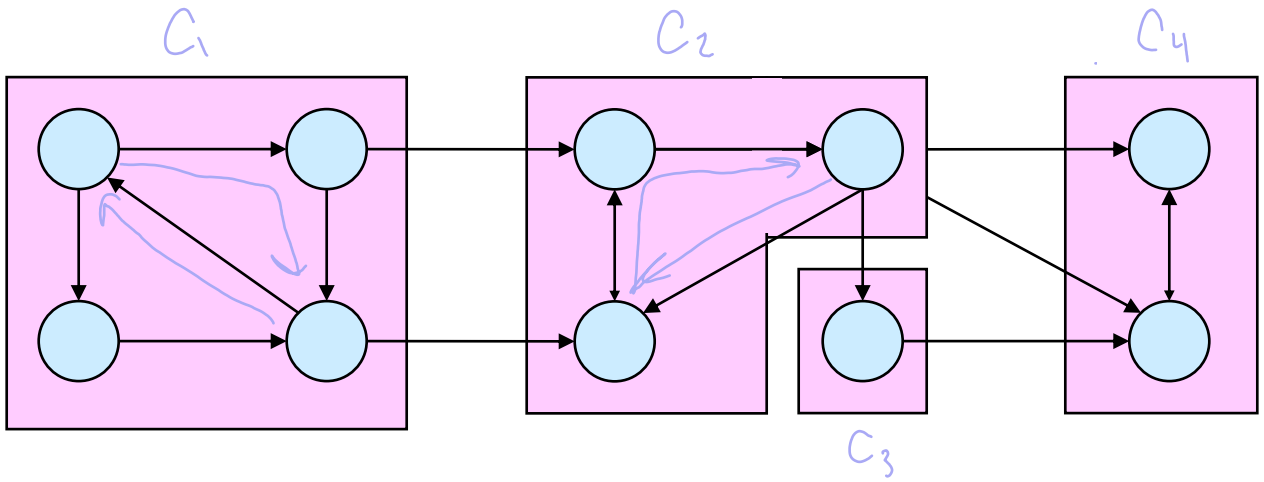
Outline

- Graphs.
  - Introduction.
  - Topological Sort. / Strong Connected Components
  - Network Flow 1.
  - Network Flow 2.
  - Shortest Path.
  - Minimum Spanning Trees.
  - Bipartite Graphs.

Graphs – Strongly Connected Components

- G is strongly connected if every pair (u, v) of vertices in G is reachable from one another.
- A **strongly connected component** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$ exist.

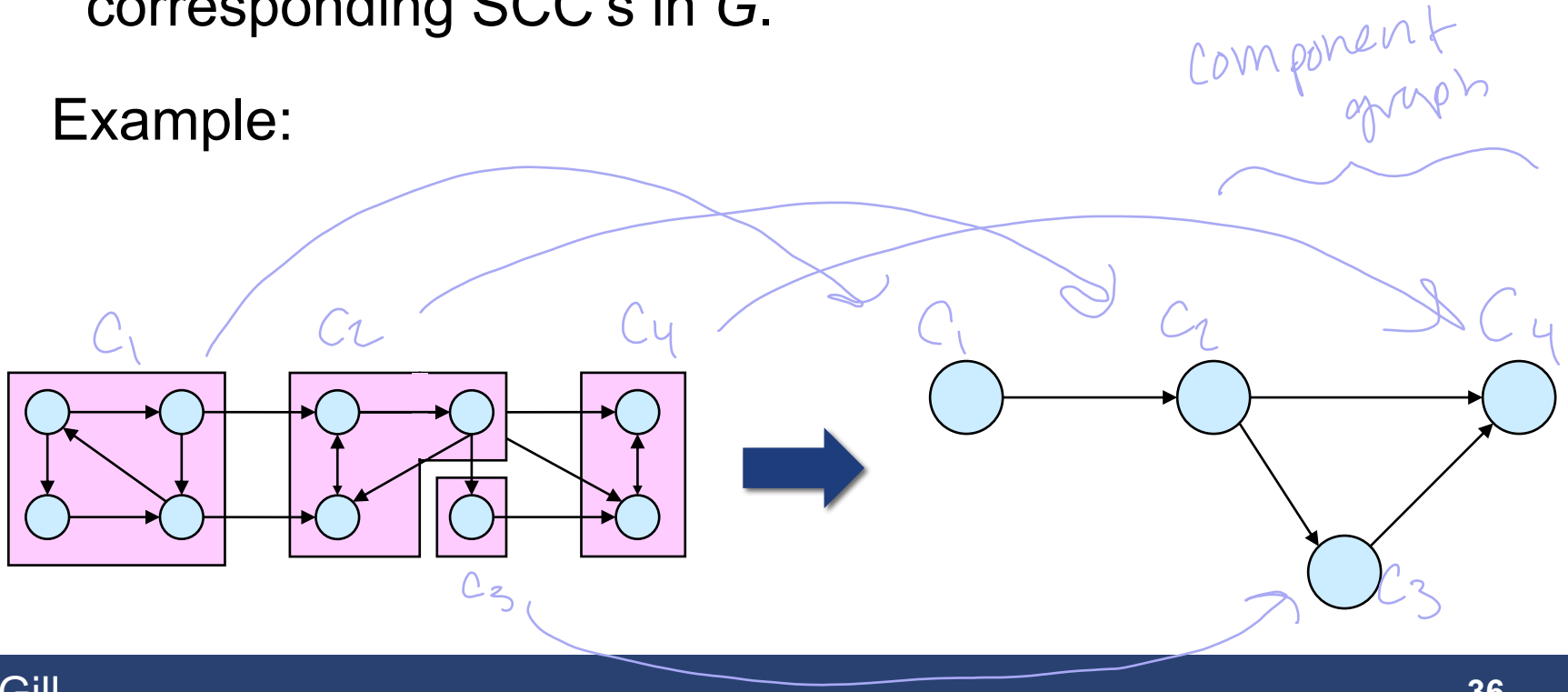
strongly connected part of a graph



Graphs – Component Graph

- $G^{SCC} = (V^{SCC}, E^{SCC})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there is an edge between the corresponding SCC's in G.

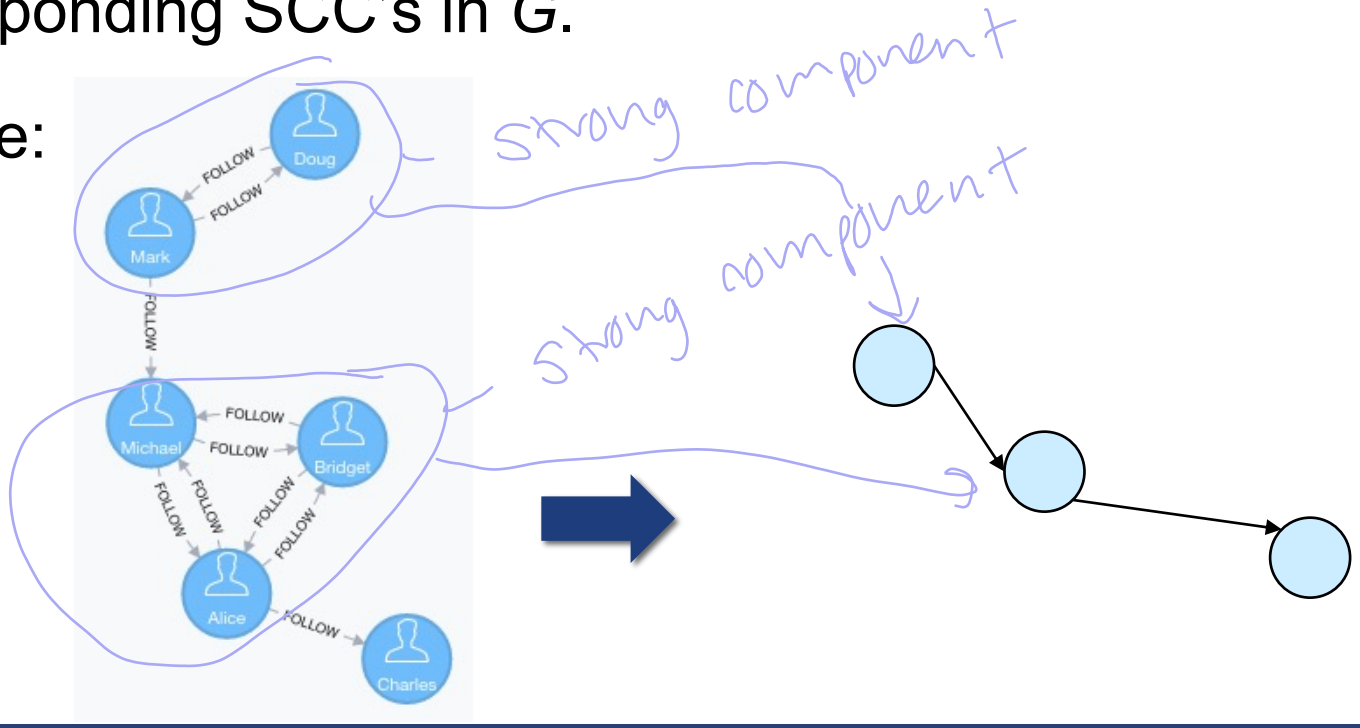
Example:



Graphs – Component Graph

- $G^{SCC} = (V^{SCC}, E^{SCC})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there is an edge between the corresponding SCC's in G.

Example:



Graphs – Component Graph

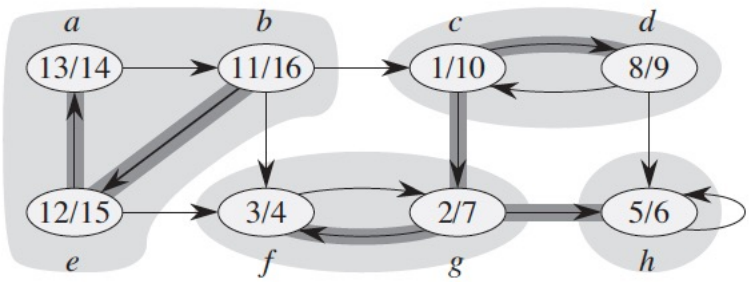
Lemma 2
Let C and C' be distinct SCC's in G, let $u, v \in C$ & $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

Proof:

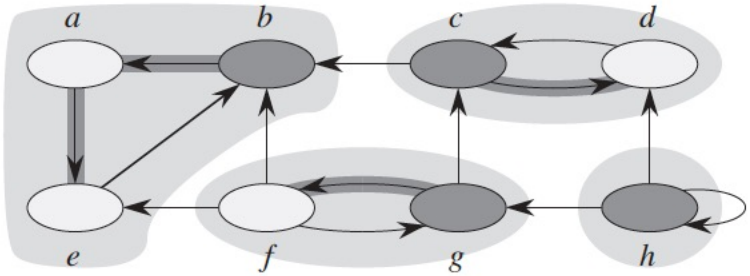
- Suppose there is a path $v' \rightsquigarrow v$ in G.
- Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G.
- Therefore, u and v' are reachable from each other, so they are not in separate SCC's.

Transpose of a Directed Graph

- G^T = **transpose** of directed G.
  - $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$.
  - G^T is G with all edges reversed.
- Can create G^T in $\Theta(V + E)$ time if using adjacency lists.
- G and G^T have the *same* SCC's.
  - u and v are reachable from each other in G if and only if reachable from each other in G^T.



G



G^T

Algorithm to determine SCCs

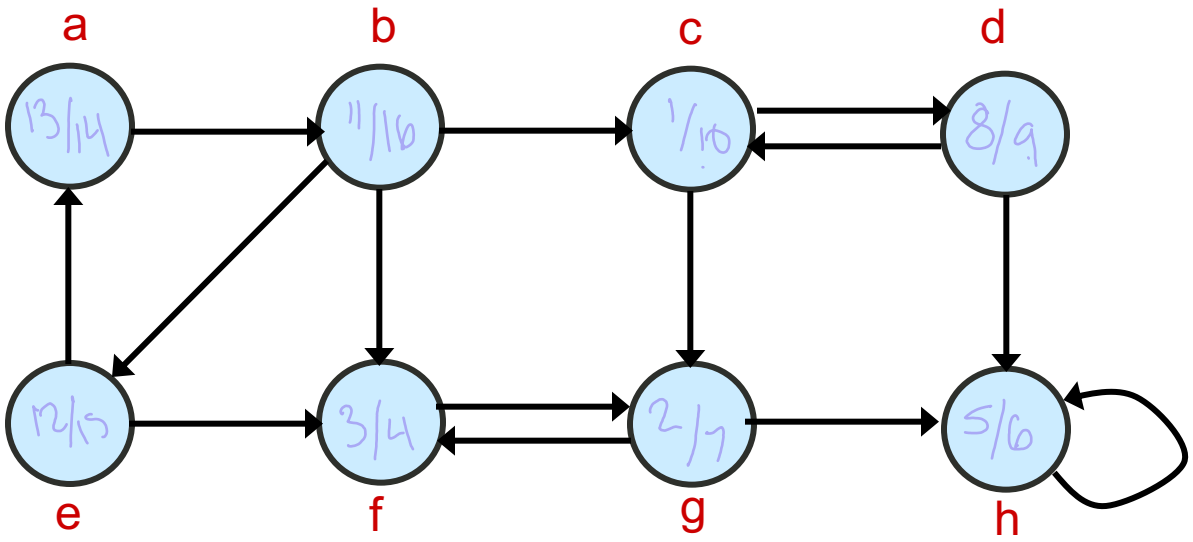
SCC(G)

1. call DFS(G) to compute finishing times $f[u]$ for all u
2. compute G^T
3. call DFS(G^T), but in the main loop, consider vertices in order of decreasing $f[u]$ (as computed in first DFS)
4. output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

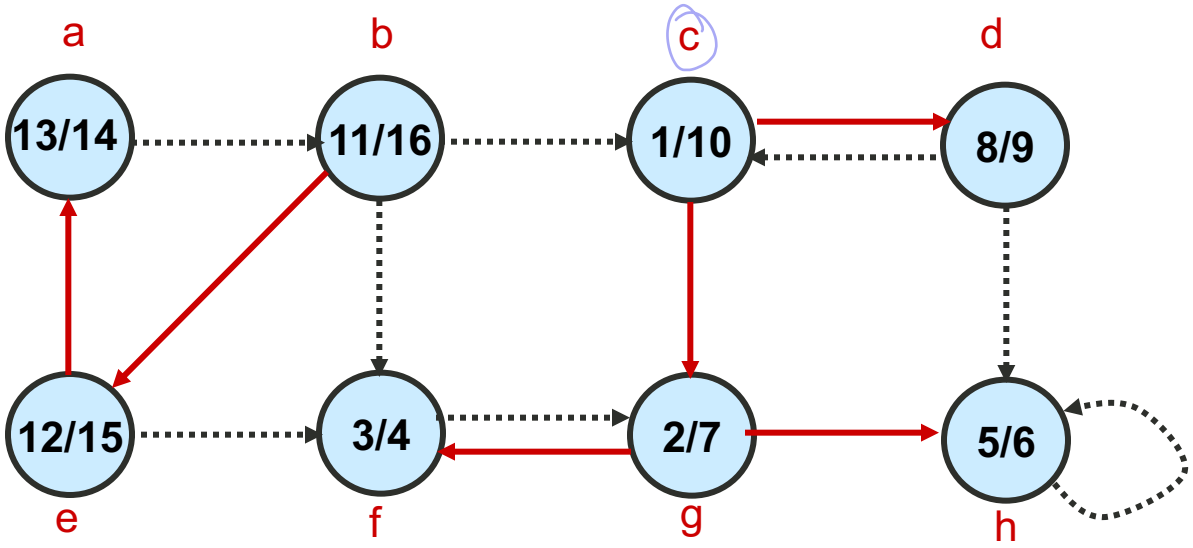
Time: $\Theta(V + E)$.

SCCs - Example

G

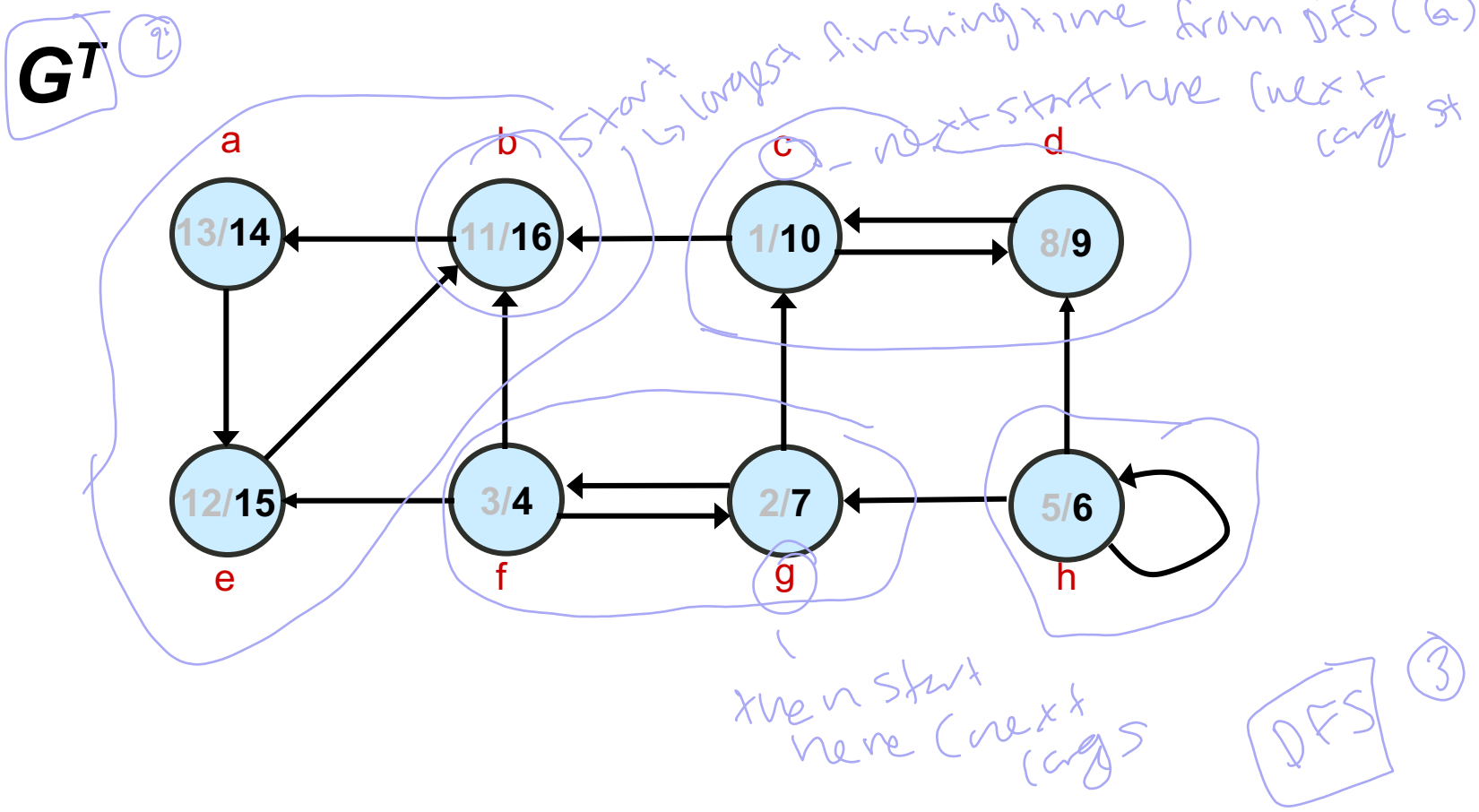


SCCs - Example



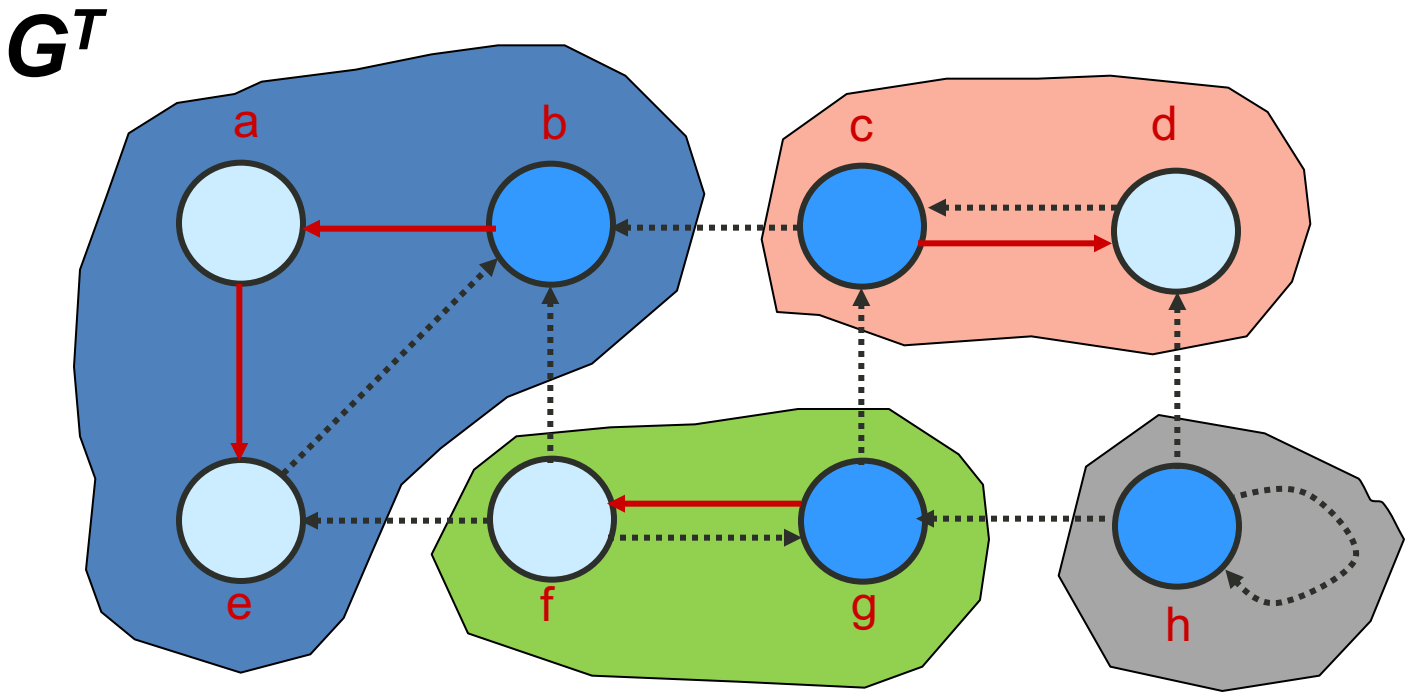
After the first DFS. We computed all finishing times in G.

SCCs - Example



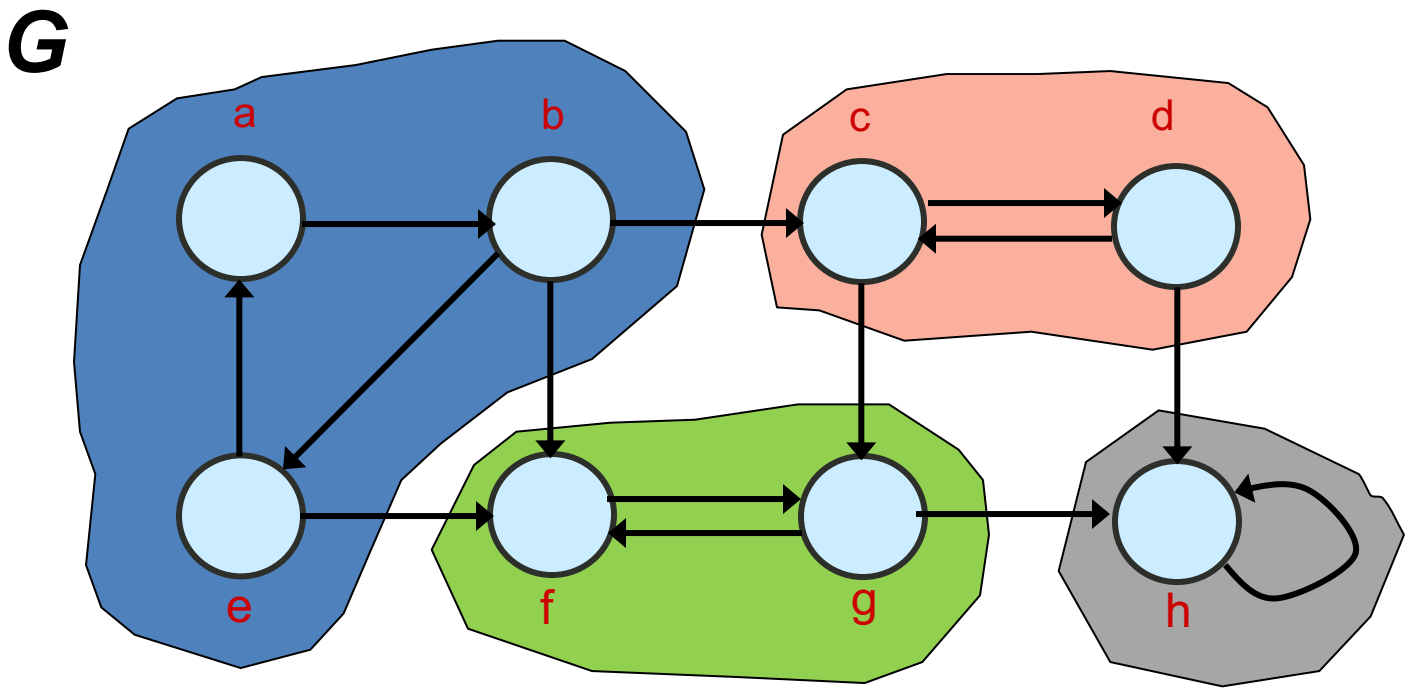
Then, we compute the transpose G^T of G and sort the vertices with the finishing time calculated in G.

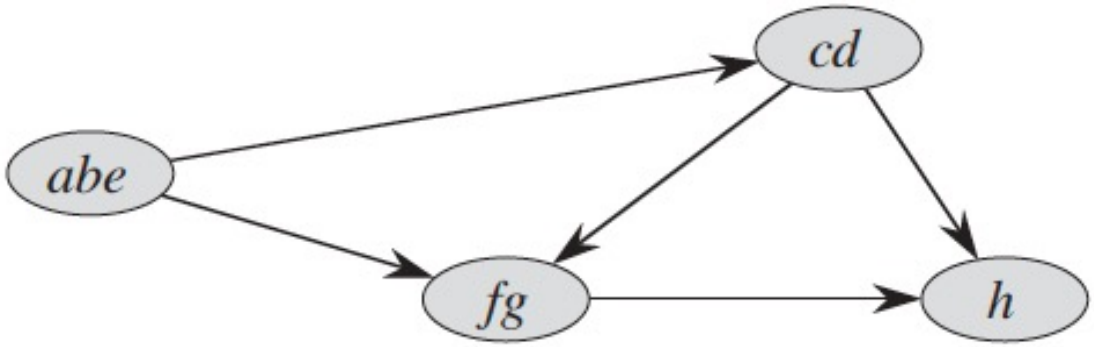
SCCs - Example



(b (a (e e) a) b) (c (d d) c) (g (f f) g) (h)

SCCs - Example





SCCs – How does it work?

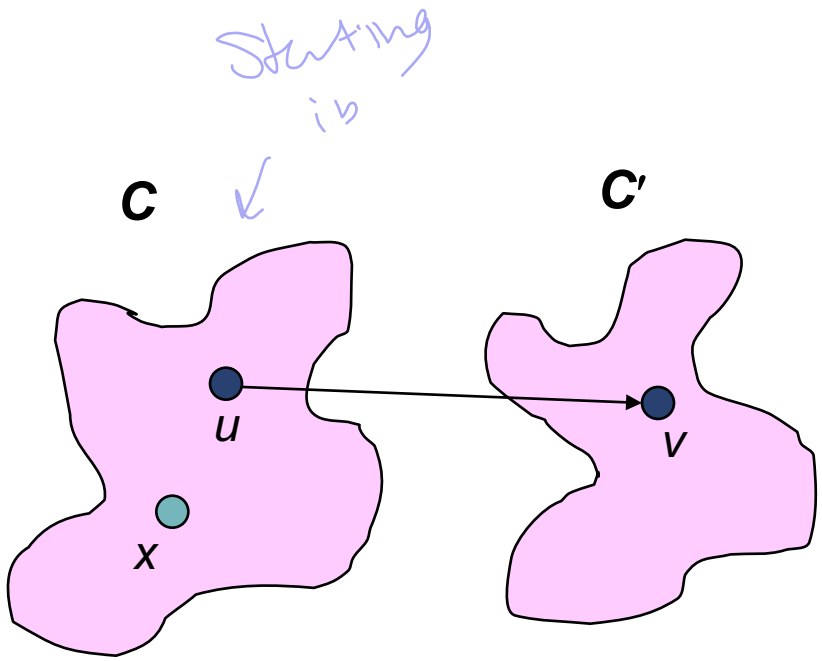
- **Idea:**
  - By considering vertices in second DFS in decreasing order of finishing times from first DFS, we are visiting vertices of the **component graph** in topologically sorted order.
  - Because we are running DFS on G^T, we will not be visiting any v from a u, where v and u are in different components.
- **Notation:**
  - $d[u]$ and $f[u]$ always refer to ***first*** **DFS**.
  - Extend notation for d and f to sets of vertices $U \subseteq V$:
    - $d(U) = \min_{u \in U}\{d[u]\}$ (earliest discovery time)
    - $f(U) = \max_{u \in U}\{f[u]\}$ (latest finishing time)

SCCs and DFS finishing times

Lemma 3
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Proof:

- Case 1: $d(C) < d(C')$
  - Let x be the first vertex discovered in C.
  - At time $d[x]$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.
  - By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.
  - By the parenthesis theorem, $f[x] = f(C) > f(C')$.



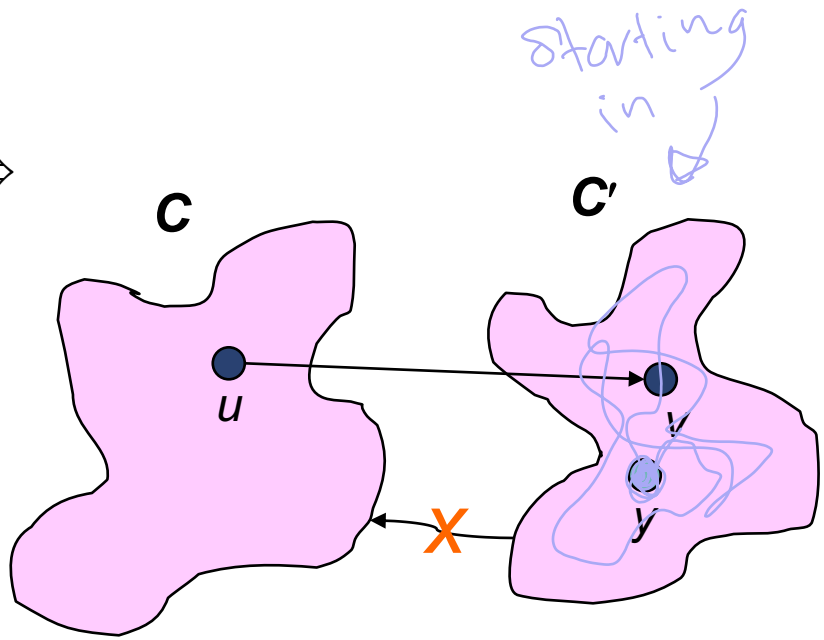
SCCs and DFS finishing times

Lemma 3

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Proof:

- Case 2: $d(C) > d(C')$
  - Let y be the first vertex discovered in C'.
  - At $d[y]$, all vertices in C' are white and there is a white path from y to each vertex in $C' \Rightarrow$ all vertices in C' become descendants of y. Again, $f[y] = f(C')$.
  - At $d[y]$, all vertices in C are also white.
  - By lemma 2, since there is an edge (u, v), we cannot have a path from C' to C.
  - So no vertex in C is reachable from y.
  - Therefore, at time $f[y]$, all vertices in C are still white.
  - Therefore, for all $w \in C$, $f[w] > f[y]$, which implies that $f(C) > f(C')$.



SCCs and DFS finishing times

Corollary 1
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof:

- $(u, v) \in E^T \Rightarrow (v, u) \in E$.
- Since SCC's of G and G^T are the same, $f(C') > f(C)$, by Lemma 3.

- Comment: each edge in G^T that goes between different strongly connected components goes from a component with an earlier finishing time (in the first depth-first search) to a component with a later finishing time.

SCCs - Correctness

1) At beginning, DFS visit only vertices in the first SCC

- When we do the second DFS, on G^T, start with SCC C such that $f(C)$ is maximum.
- The second DFS starts from some $x \in C$, and it visits all vertices in C.
- Corollary 1 says that since $f(C) > f(C')$ for all $C \neq C'$, there are no edges from C to C' in G^T.
- Therefore, **DFS will visit *only* vertices in *C***.
- Which means that the depth-first tree rooted at x contains *exactly* the vertices of C.

SCCs - Correctness

2) DFS doesn't visit more than one new SCC at the time

- The next root in the second DFS is in SCC *C′* such that *f* (*C′*) is maximum over all SCC's other than C.
  - DFS visits all vertices in *C′*, but **the only edges out of *C′* go to *C*, *which we have already visited***.
  - Therefore, the only tree edges will be to vertices in *C′*.
- Iterate the process.
- Each time we choose a root, it can reach only:
  - vertices in its SCC—get tree edges to these,
  - vertices in SCC's *already visited* in second DFS—get *no* tree edges to these.

Outline

- Graphs.
  - Introduction.
  - Strong Connected Components / Topological Sort.
  - Network Flow 1.
  - Network Flow 2.
  - Shortest Path.
  - Minimum Spanning Trees.
  - Bipartite Graphs.


time for questions