CSE 2202

Design and Analysis of Algorithms – I

Lecture 4

Topological Sort
Strongly Connected Components

DFS Review

DFS Review

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1          // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]    // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK          // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

DFS Review

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5       if v.color == WHITE
 6            v.π = u
 7            DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

```
DFS(G)
1  for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6       if u.color == WHITE
7            DFS-VISIT(G, u)
```

Every time DFS-VISIT(*G*, u) is called, u becomes the root of a new tree in the depth-first forest

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1   time = time + 1          // white vertex u has just been discovered
2   u.d = time
3   u.color = GRAY
4   for each v ∈ G.Adj[u]    // explore edge (u, v)
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   u.color = BLACK          // blacken u; it is finished
9   time = time + 1
10  u.f = time
```

- **The first "loop" over all vertices happens because we want to ensure we cover all vertices, including those not reachable from our starting vertex (i.e., if the graph is not connected). This part contributes O(V) to the overall time complexity.**

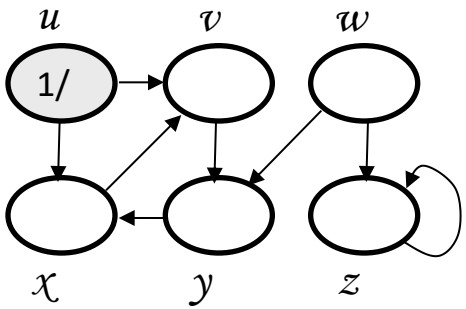
- Now, for each vertex, we don't exactly iterate over *all* vertices. Instead, we iterate over the *neighbours* of the given vertex. In other words, we iterate over its adjacency list, which basically corresponds to the edges connected to that vertex.

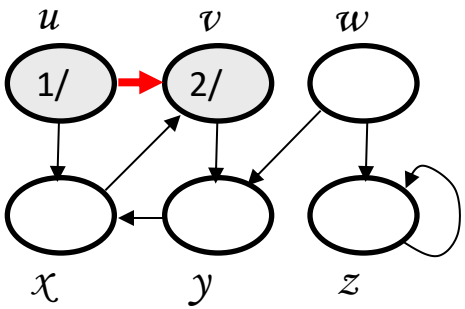
- **Each edge will be examined exactly twice in an undirected graph: once for each of its endpoints. Therefore, the second "loop" over the adjacency lists for each vertex contributes O(2E) = O(E) to the time complexity.**

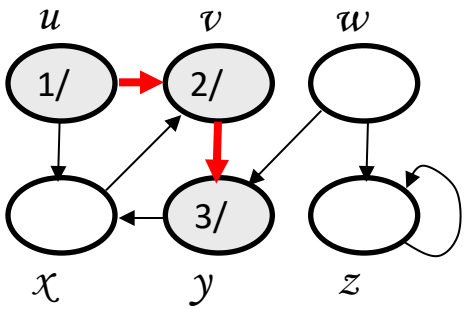
  So, when we sum the two parts, we get O(V + E).

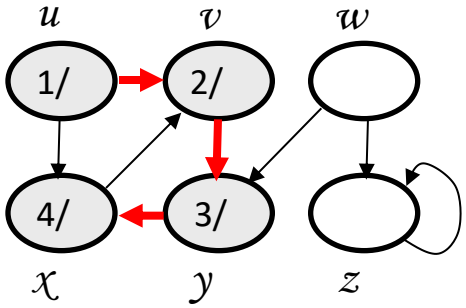
DFS Complexity

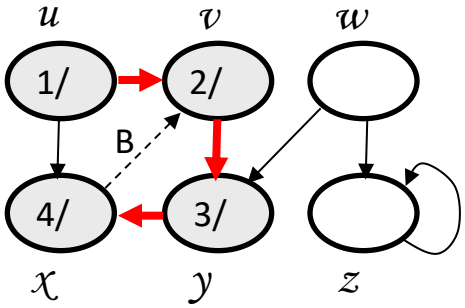
Example

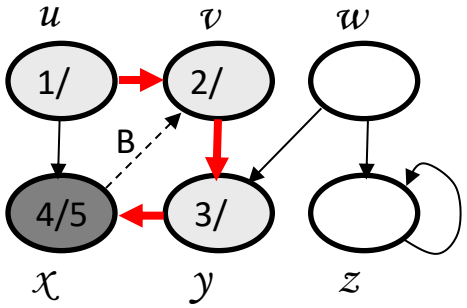

u
v
w
1/
x
y
z

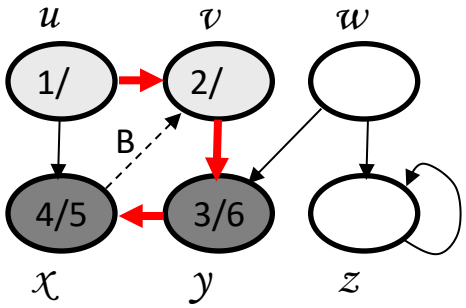

u
v
w
1/
2/
x
y
z

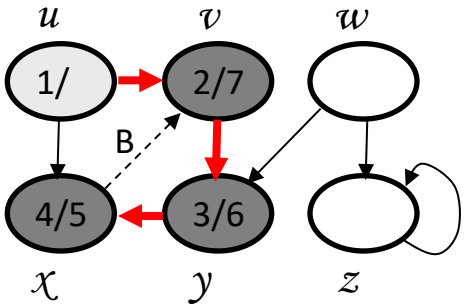

u
v
w
1/
2/
3/
x
y
z

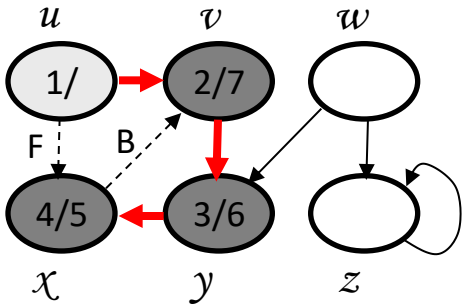

u
v
w
1/
2/
4/
3/
x
y
z

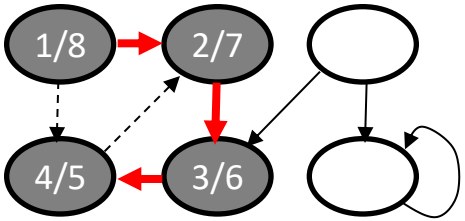

u
v
w
1/
2/
B
4/
3/
x
y
z

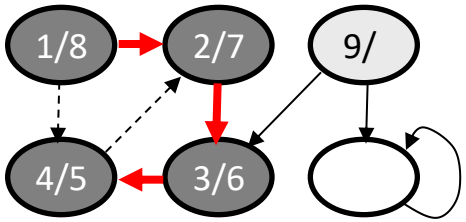

u
v
w
1/
2/
B
4/5
3/
x
y
z

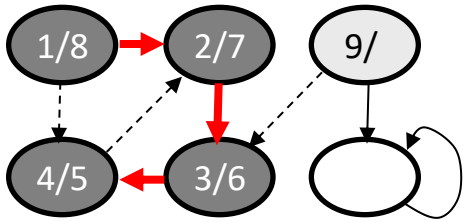

u
v
w
1/
2/
B
4/5
3/6
x
y
z

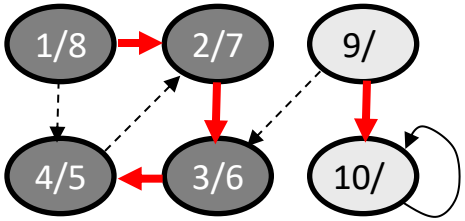

u
v
w
1/
2/7
B
4/5
3/6
x
y
z

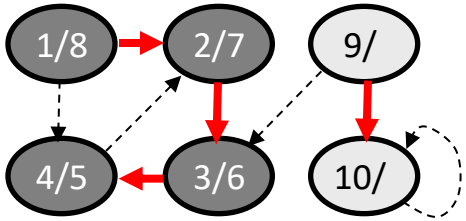

u
v
w
1/
2/7
F
B
4/5
3/6
x
y
z

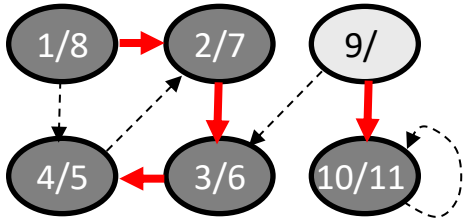


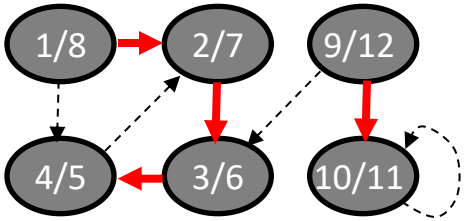










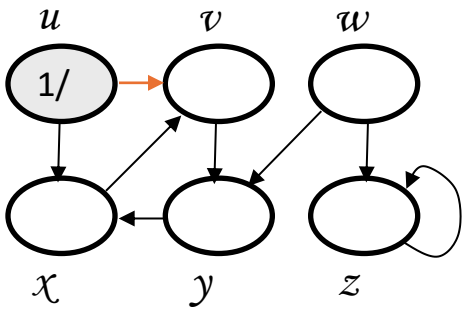


The results of DFS may depend on:

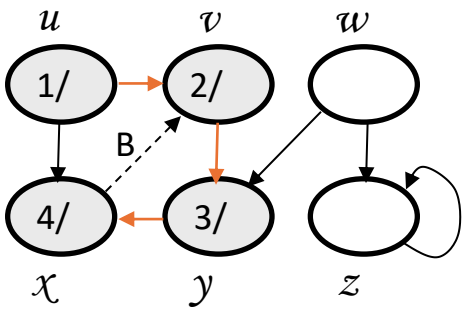
- The order in which nodes are explored in procedure DFS
- The order in which the neighbors of a vertex are visited in DFS-VISIT

Edge Classification

- **Tree edge** (reaches a WHITE vertex)**:**
  - (u, v) is a tree edge if v was first discovered by exploring edge (u, v)

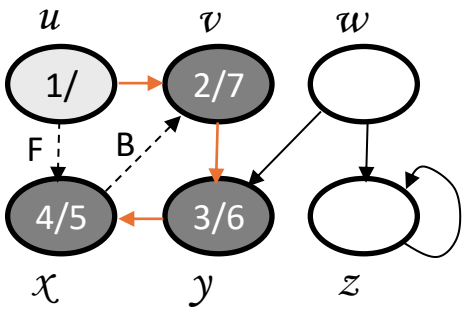


- **Back edge** (reaches a GRAY vertex)**:**
  - (u, v), connecting a vertex u to an ancestor v in a depth first tree
  - Self loops (in directed graphs) are also back edges

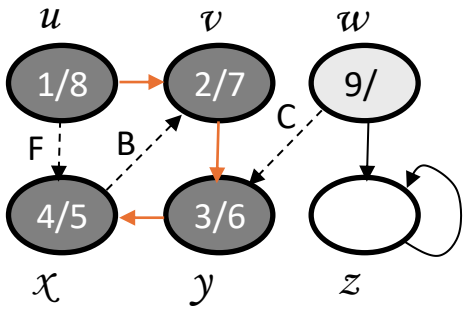


Edge Classification

- **Forward edge** (reaches a BLACK vertex & d[u] < d[v]): **(here d[u] is the same as *u.d*)**
  - Non-tree edges (u, v) that connect a vertex u to a descendant v in a depth first tree



- **Cross edge** (reaches a BLACK vertex & d[u] > d[v]):
  - Can go between vertices in same depth-first tree (as long as there is no ancestor / descendant relation) or between different depth-first trees

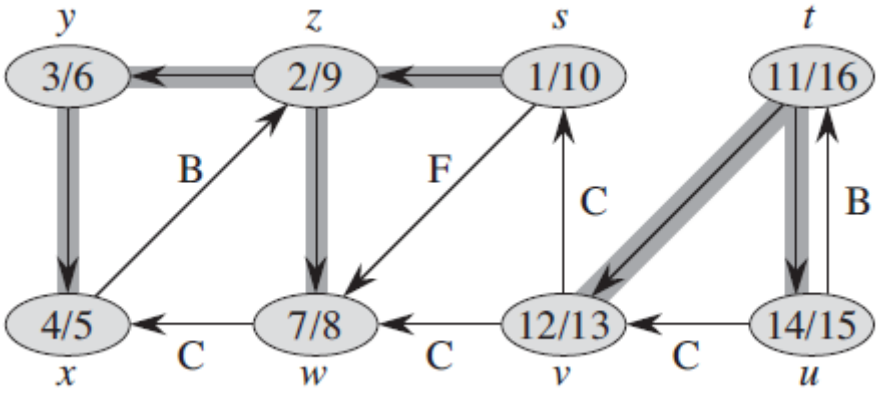


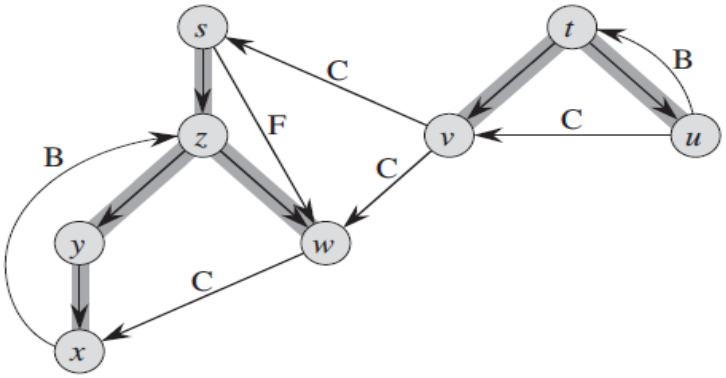
In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

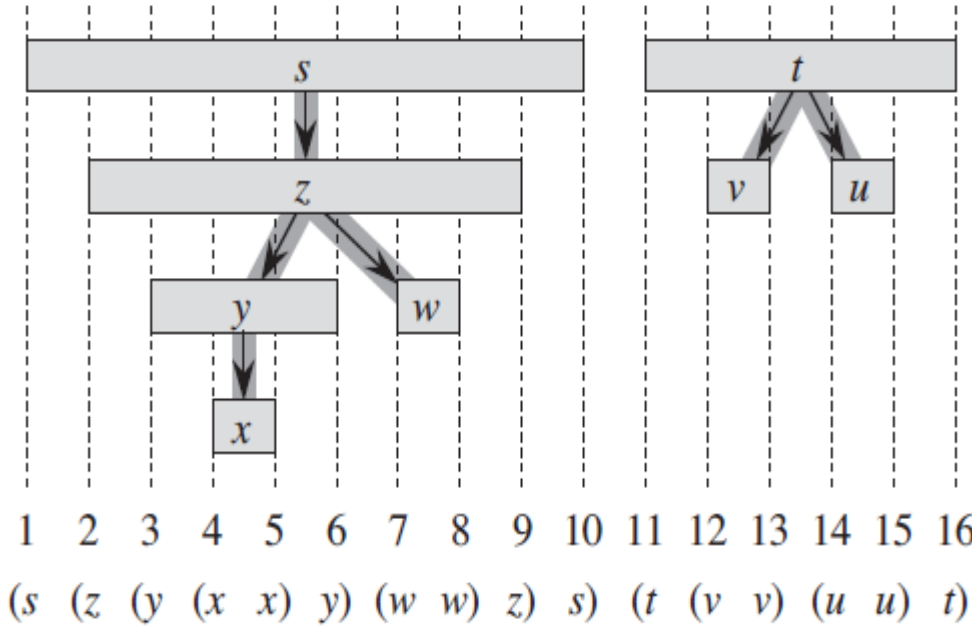
Theorem 22.7 (Parenthesis theorem)

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.



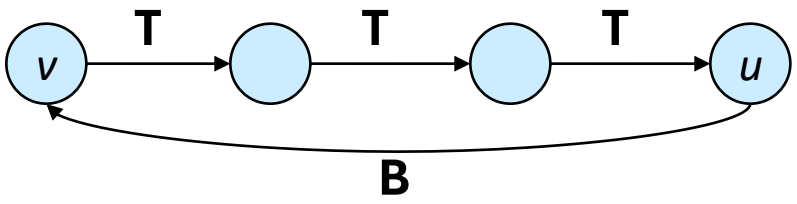




Well-formed expression: parenthesis are properly nested

Characterizing a DAG

Lemma 22.11
A directed graph *G* is acyclic iff a DFS of G yields no back edges.



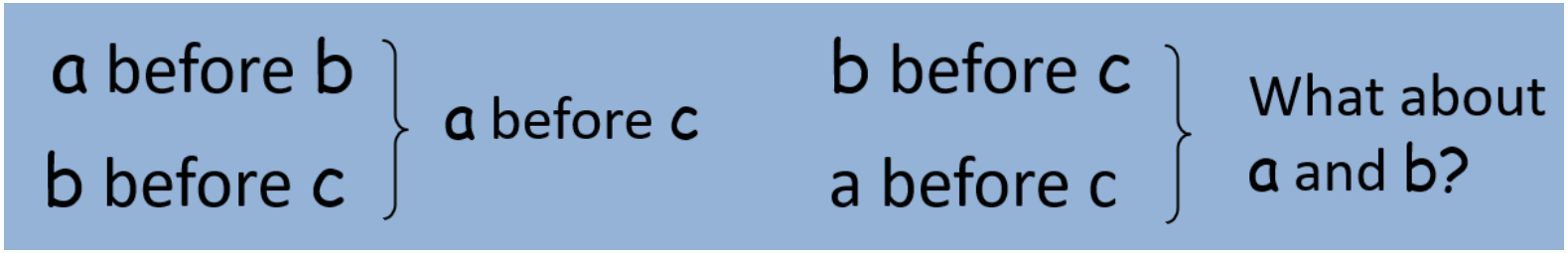
Topological sort

Topological Sort

Topological sort of a directed acyclic graph

G = (V, E): a linear order of vertices such that if there exists an edge (u, v), then u appears before v in the ordering.

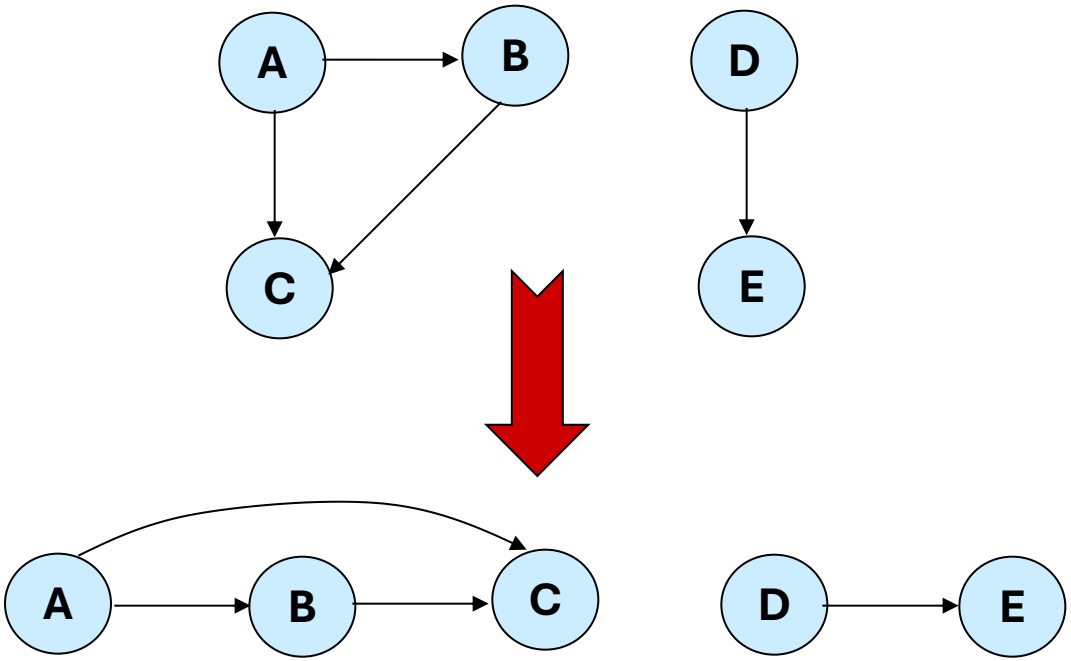
- Used to represent precedence of events or processes that have a **partial order**



Topological sort helps us establish a **total order**

Topological Sort

Want to “sort” a directed acyclic graph (DAG).



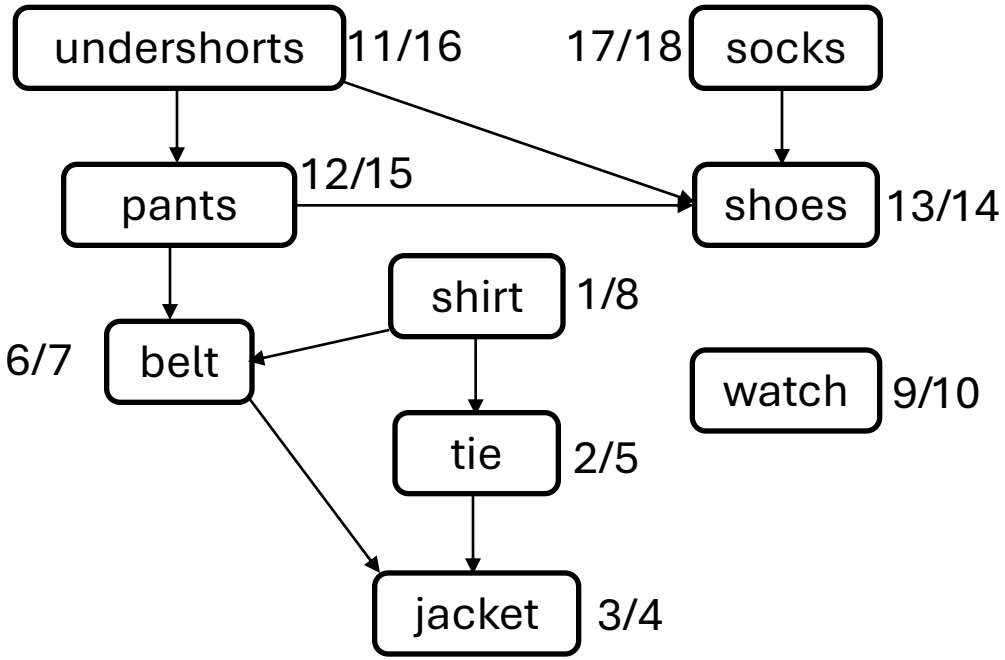
Think of original DAG as a **partial order**.

Want a **total order** that extends this partial order.

Topological Sort - Application

- Application 1
  - in scheduling a sequence of jobs.
  - The jobs are represented by vertices,
  - there is an edge from *x* to *y* if job *x* must be completed before job *y* can be done
    - (for example, washing machine must finish before we put the clothes to dry). Then, a topological sort gives an order in which to perform the jobs
- Application 2
  - In open credit system, how to take courses (in order) such that, pre-requisite of courses will not create any problem

Topological Sort (Fig – Cormen)



TOPOLOGICAL-SORT(V, E)

1. Call DFS(V, E) to compute finishing times f[v] for each vertex v
2. When each vertex is finished, insert it onto the front of a linked list
3. Return the linked list of vertices

socks | undershorts | pants | shoes | watch | shirt | belt | tie | jacket

Running time: $\Theta(V + E)$

Topological Sort on detection of a cyclic graph?

- It's not possible to generate a **topological sort for a graph** that contains a cycle.
- Topological sorting is for D**irected Acyclic Graphs (DAGs)**
- If you attempt a topological sort on a graph with a cycle, your algorithm should detect the cycle and report that a topological sort is not possible.
- This can be done, for instance, by maintaining **three states for each vertex during a DFS**: unvisited, visited but still in recursion stack (i.e., being explored), and completely processed.

If a vertex that's still in the recursion stack is encountered again, that indicates a cycle.

Strongly Connected Component

- A classic application of DFS: **decomposing** a directed graph into its **strongly connected components**.
- We will see how to do so using **two depth-first searches**.

- Many algorithms that work with directed graphs begin with such a decomposition.
- **After decomposing the graph into strongly connected components**
  - such algorithms run separately on each one and
  - then combine the solutions according to the structure of connections among components.

Connectivity

- Connected Graph
  - In an **undirected graph** G, two vertices u and v are called connected if G contains a path from u to v. Otherwise, they are called disconnected.
  - A **directed graph** is called connected if every pair of distinct vertices in the graph is connected.
- **Connected Components**
  - A connected component is a **maximal connected subgraph** of G. Each vertex belongs to exactly one connected component, as does each edge.

Connectivity (cont.)

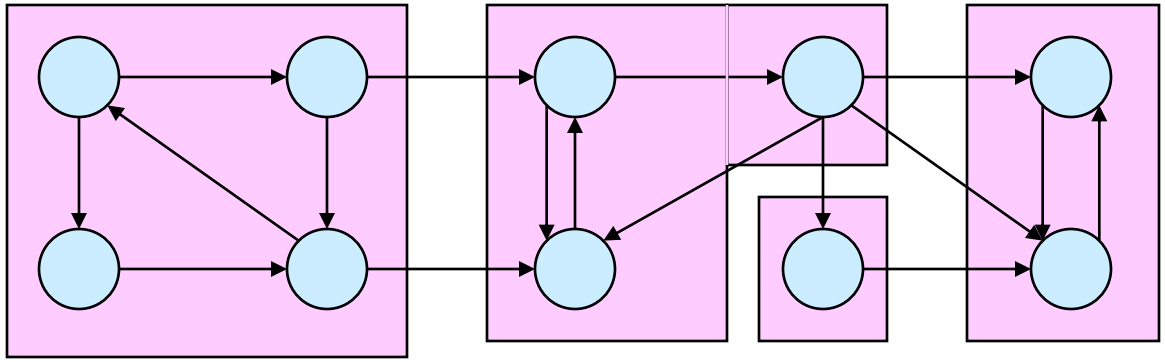
- Weakly Connected Graph
  - A directed graph is called **weakly connected** if replacing all of its directed edges with undirected edges produces a connected (undirected) graph.
- Strongly Connected Graph
  - It is strongly connected or strong if it contains a directed path from u to v for every pair of vertices u, v. The strong components are the maximal strongly connected subgraphs

Connected Components

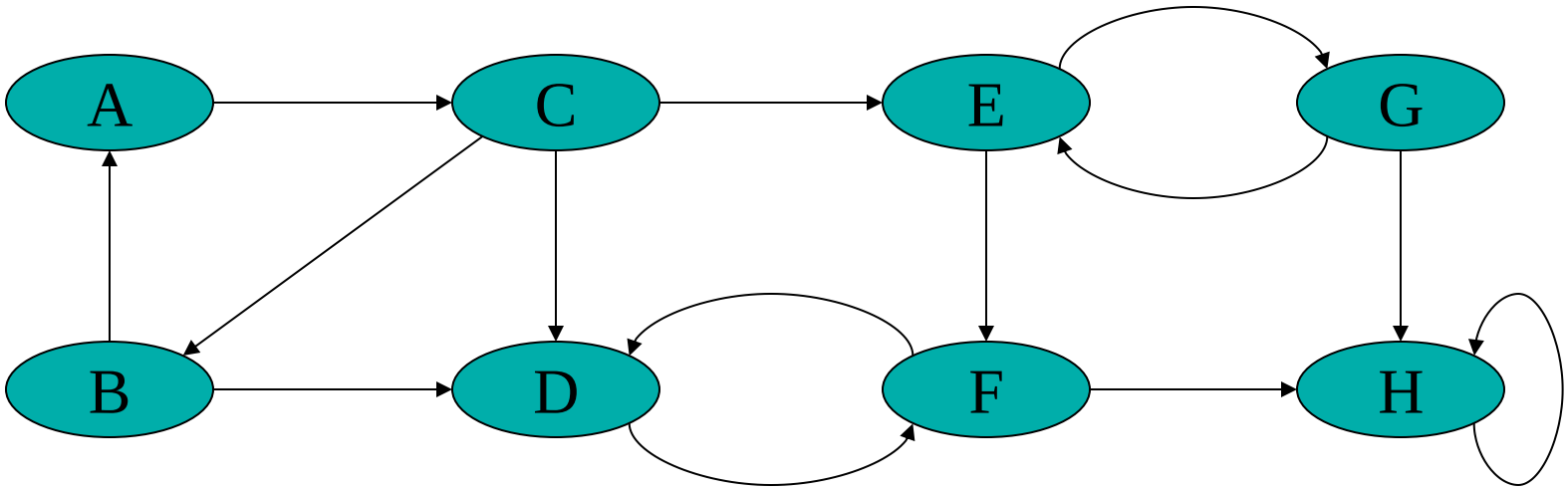
- Strongly connected graph
  - A directed graph is called *strongly connected* if for every pair of vertices *u* and *v* there is a path from *u* to *v* and a path from *v* to *u*.
- Strongly Connected Components (SCC)
  - The **strongly connected components** (**SCC**) of a directed graph are its maximal strongly connected subgraphs.
- Here, we work with
  - Directed unweighted graph

Strongly Connected Components

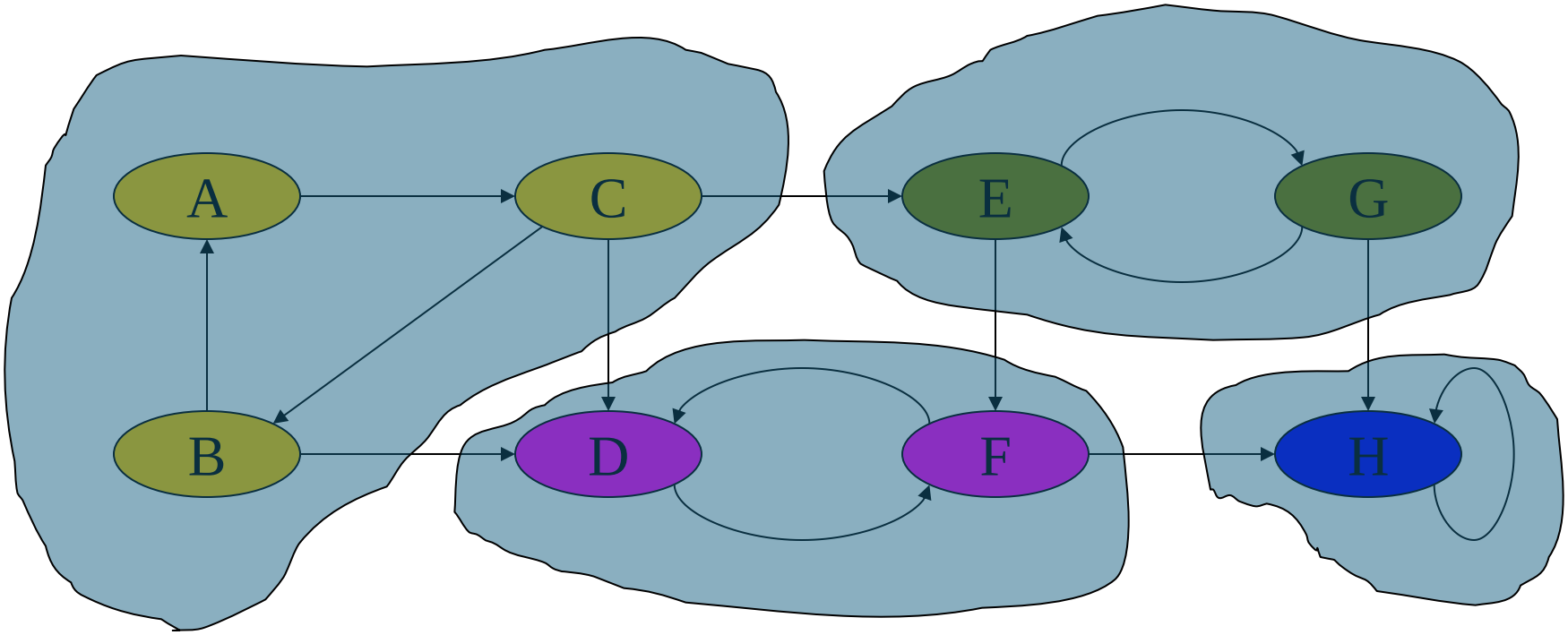
- G is strongly connected if every pair (u, v) of vertices in G is reachable from one another.
- A **strongly connected component** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$



DFS - Strongly Connected Components

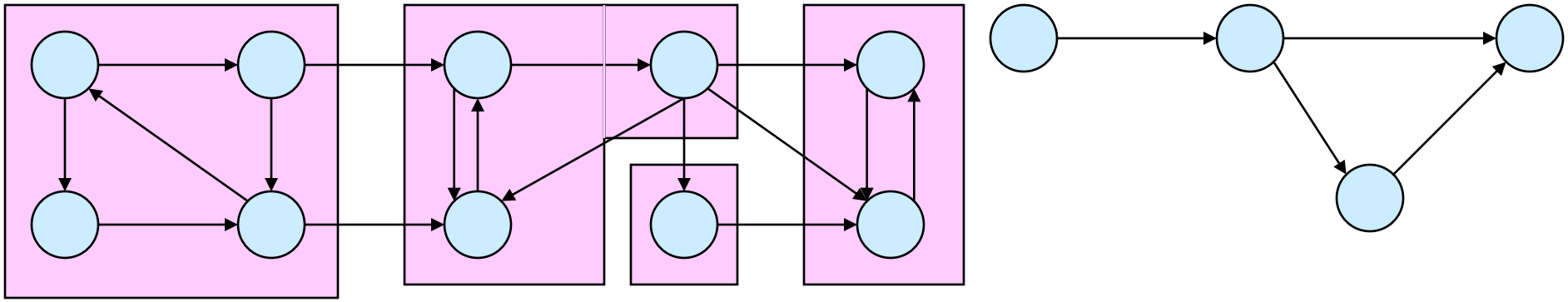


DFS - Strongly Connected Components



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's an edge between the corresponding SCC's in G.
- G^{SCC} for the example considered:



Transpose of a Directed Graph

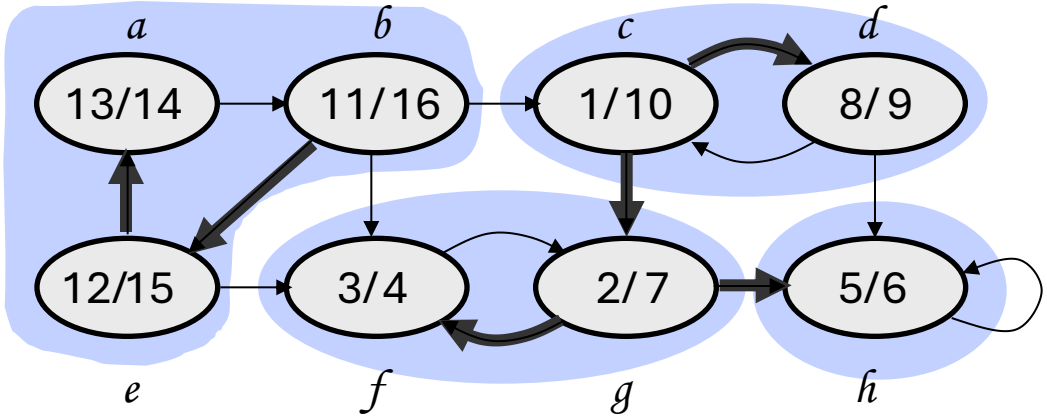
- G^T = **transpose** of directed G.
  - $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$.
  - G^T is G with all edges reversed.
- G and G^T have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^T.)

Algorithm to determine SCCs

STRONGLY-CONNECTED-COMPONENTS(G)

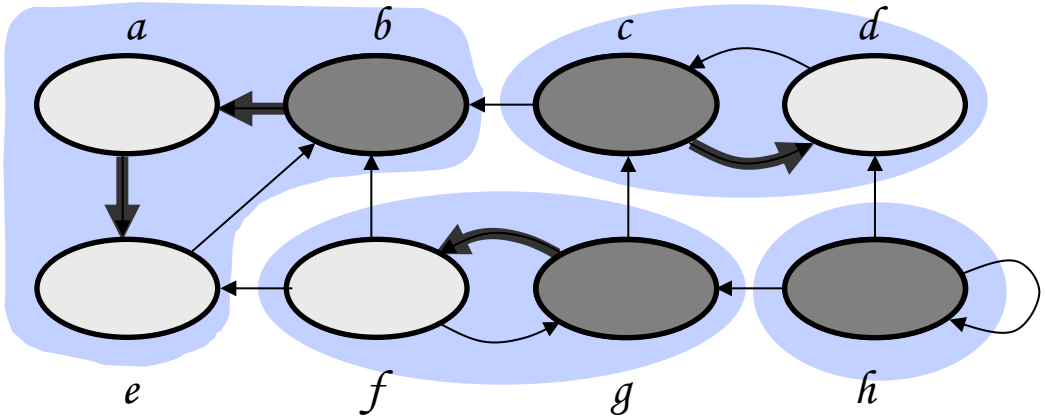
1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices
in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
separate strongly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

Example



DFS on the initial graph G

b	e	a	c	d	g	h	f
16	15	14	10	9	7	6	4

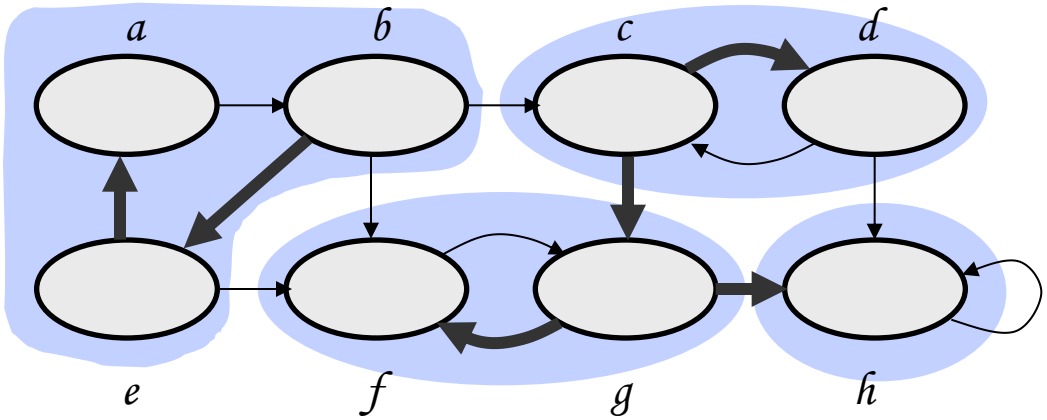


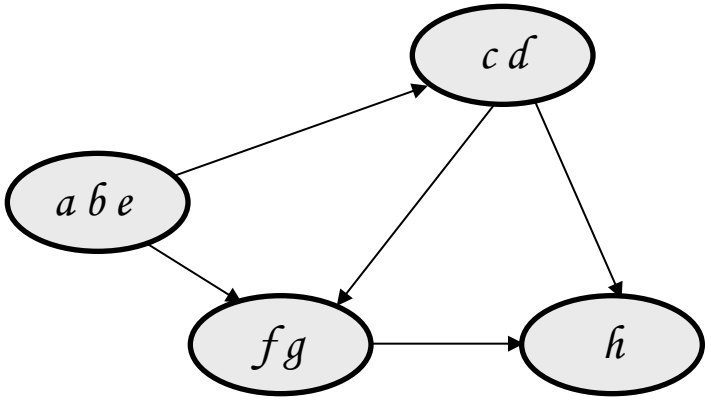
DFS on G^T:

- start at b: visit a, e
- start at c: visit d
- start at g: visit f
- start at h

Strongly connected components: $C_1 = \{a, b, e\}$, $C_2 = \{c, d\}$, $C_3 = \{f, g\}$, $C_4 = \{h\}$

Component Graph

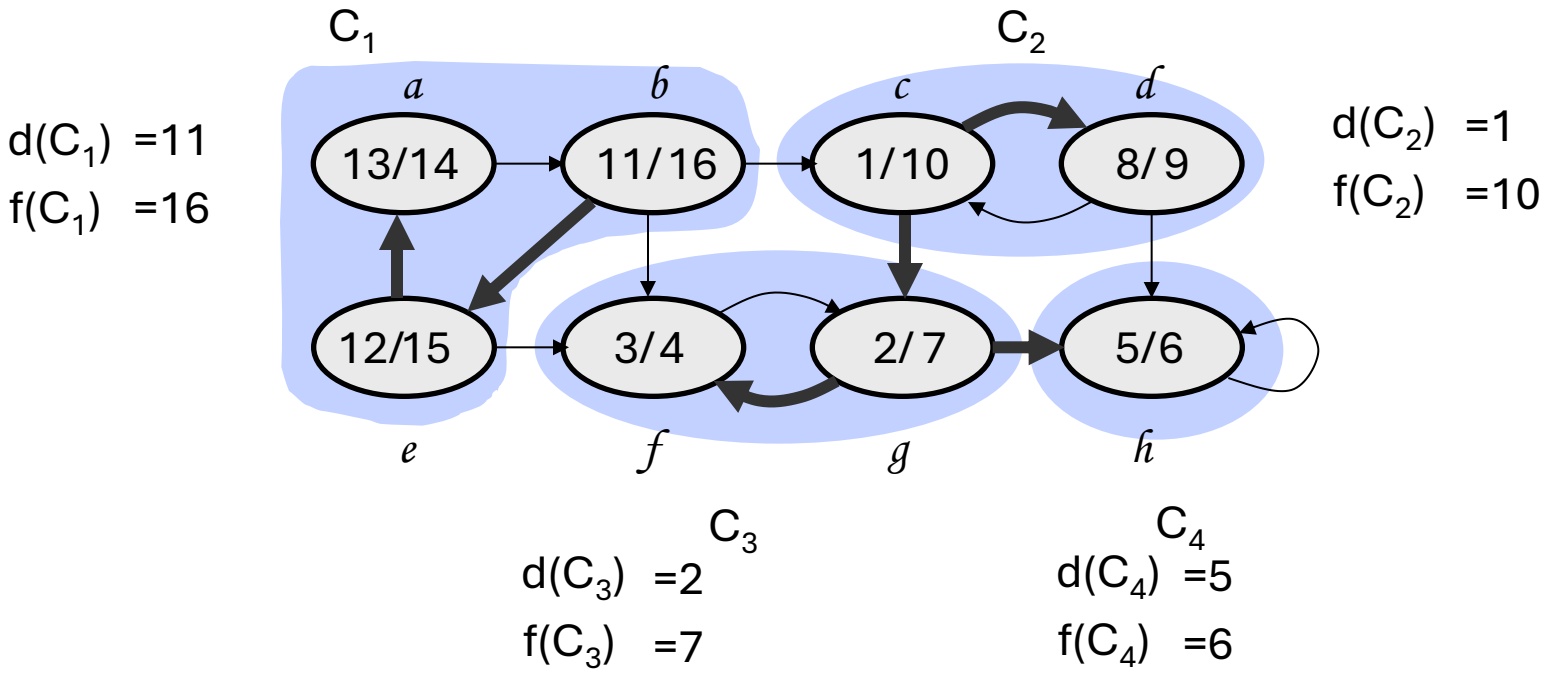




- The **component graph** $G^{SCC} = (V^{SCC}, E^{SCC})$:
  - $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$, where v_i corresponds to each strongly connected component C_i
  - There is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and $y \in C_j$
- The component graph is a DAG

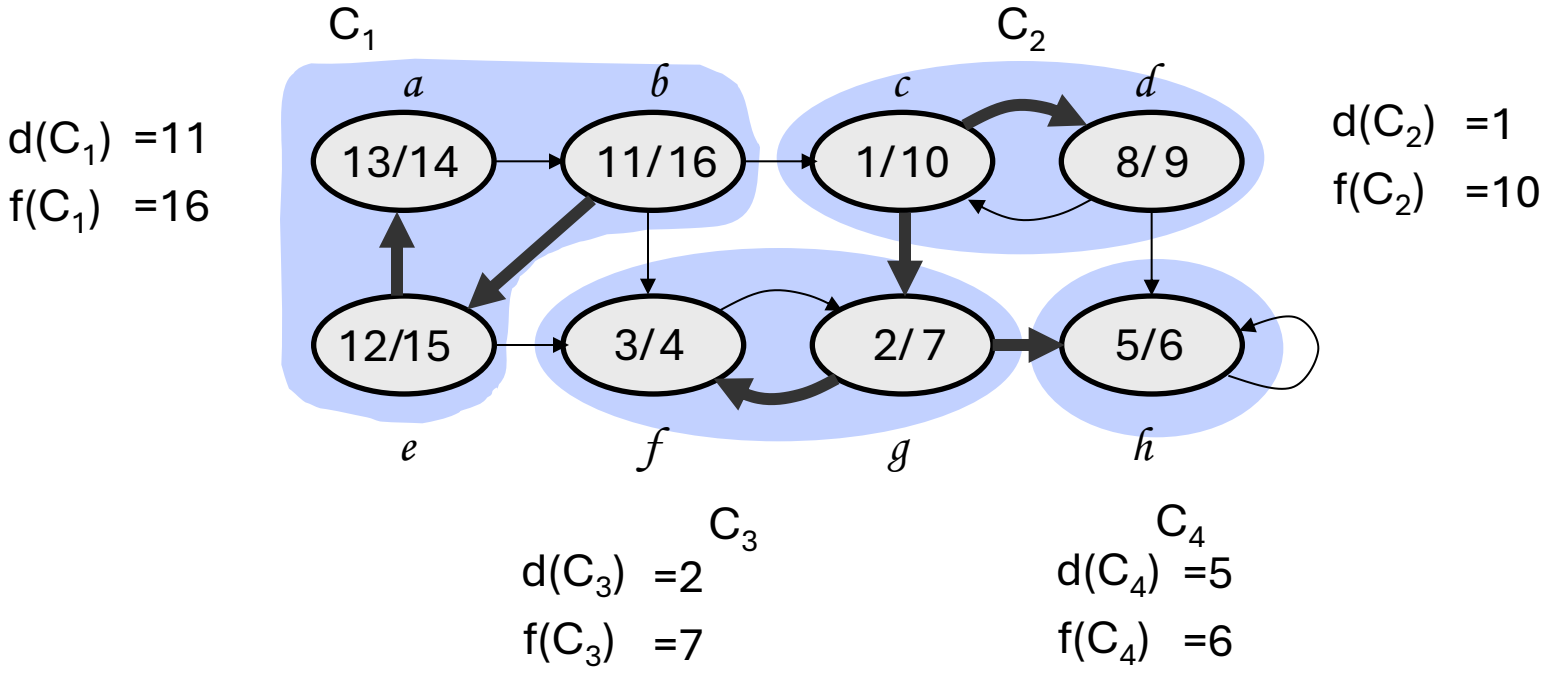
Notations

- Extend notation for d (starting time) and f (finishing time) 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)



Lemma 2

- Let C and C' be distinct SCCs in a directed graph G = (V, E). If there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$ then f(C) > f(C').
- Consider C_1 and C_2, connected by edge (b, c)



$d(C_1) = 11$
$f(C_1) = 16$

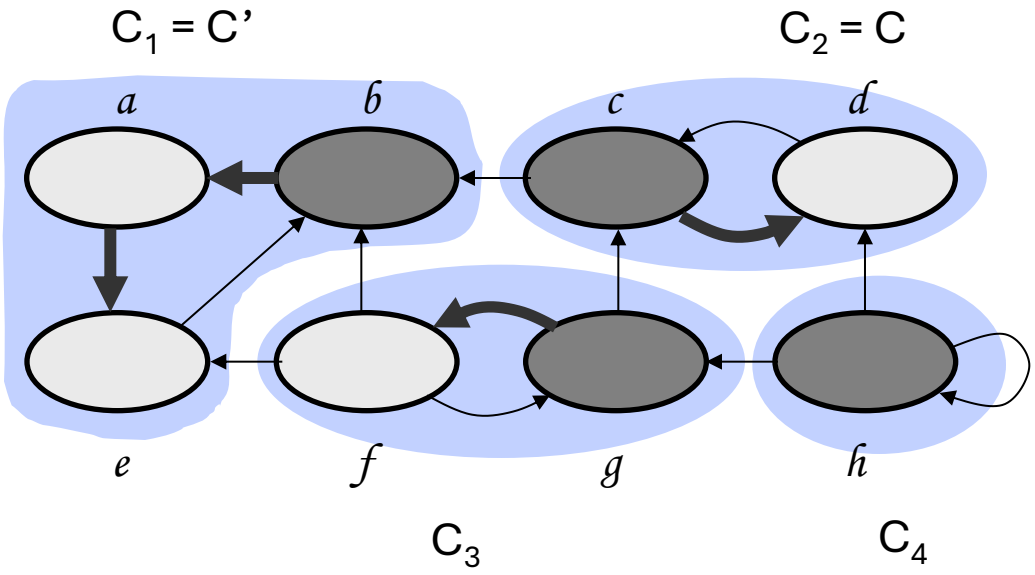
$d(C_2) = 1$
$f(C_2) = 10$

$d(C_3) = 2$
$f(C_3) = 7$

$d(C_4) = 5$
$f(C_4) = 6$

Corollary 1

- Let C and C' be distinct SCCs in a directed graph G = (V, E). If there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$ then $f(C) < f(C')$.
- Consider C_2 and C_1, connected by edge (c, b)



- Since $(c, b) \in E^T \Rightarrow (b, c) \in E$
- From previous lemma:

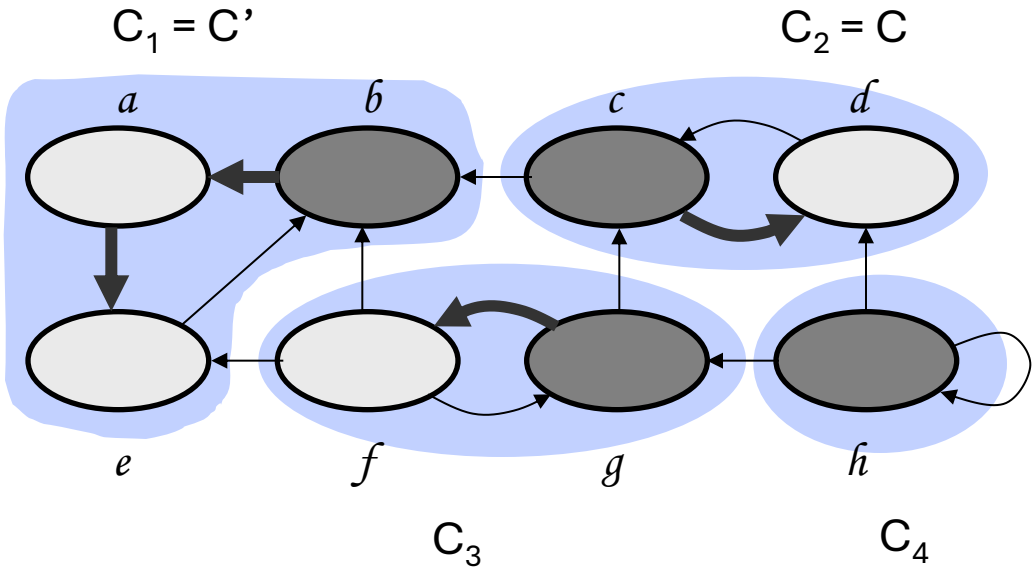
  $f(C_1) > f(C_2)$

  $f(C') > f(C)$

  $f(C) < f(C')$

Corollary 2

- Each edge in G^T that goes between different components goes from a component with an earlier finish time (in the DFS) to one with a later finish time



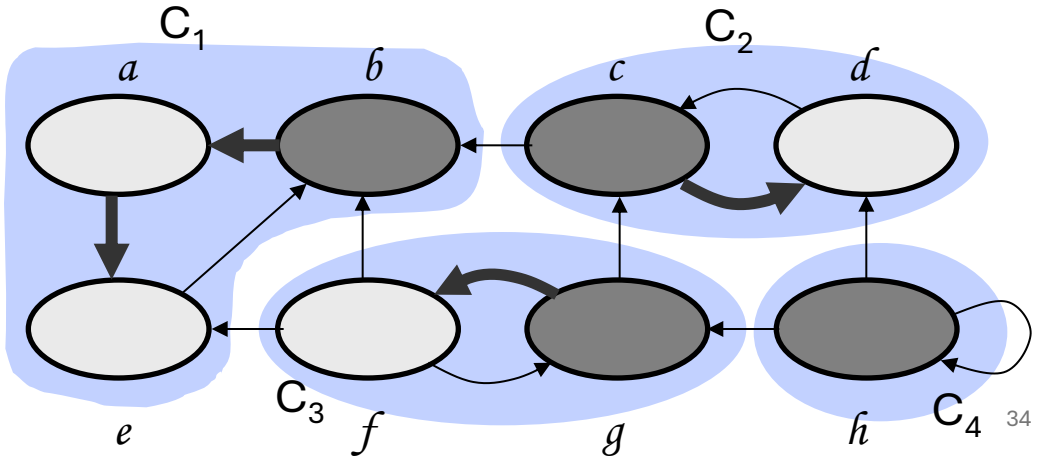
Why does SCC Work?

- When we do the second DFS, on G^T, we start with a component C such that f(C) is maximum (b, in our case)
- We start from b and visit all vertices in C_1
- From corollary: f(C) > f(C') in G for all $C \neq C' \Rightarrow$ there are no edges from C to any other SCCs in G^T

$\Rightarrow$ DFS will visit only vertices in C_1

$\Rightarrow$ The depth-first tree rooted at b contains exactly the vertices of C_1

b	e	a	c	d	g	h	f
16	15	14	10	9	7	6	4



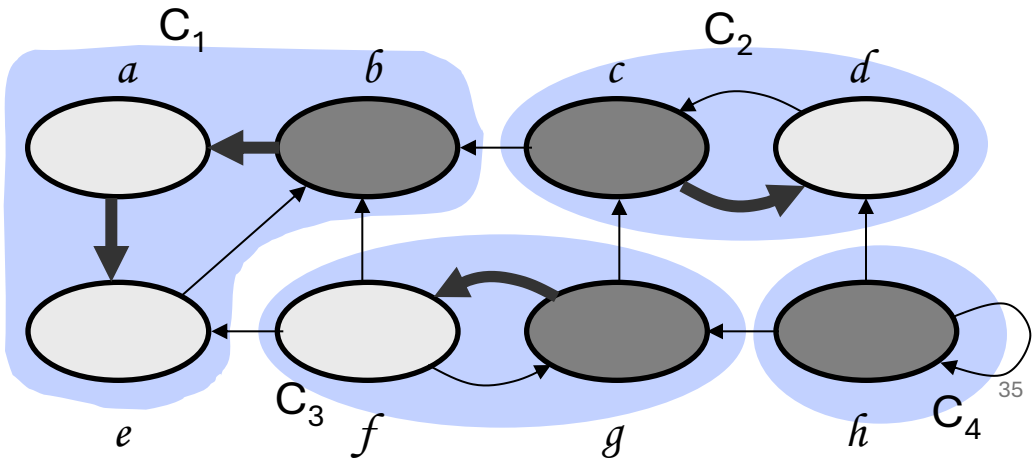
Why does SCC Work? (cont.)

- The next root chosen in the second DFS is in SCC C_2 such that f(C) is maximum over all SCC's other than C_1
- DFS visits all vertices in C_2
  - the only edges out of C_2 go to C_1, which we've already visited

$\Rightarrow$ The only tree edges will be to vertices in C_2

- Each time we choose a new root it can reach only:
  - vertices in its own component
  - vertices in components *already visited*

b	e	a	c	d	g	h	f
16	15	14	10	9	7	6	4



How can the number of strongly connected components of a graph change if a new edge is added?

How can the number of strongly connected components of a graph change if a new edge is added?

- When a new edge is added to a directed graph, the number of strongly connected components (SCCs) can either decrease or stay the same, but it cannot increase.

1. **Decrease**: The addition of an edge may connect two previously separate SCCs. For example, if you have a graph with two SCCs A and B, and you add an edge from a node in A to a node in B and an edge from a node in B to a node in A, then A and B will become part of the same SCC, reducing the total number of SCCs.
2. **Stay the same**: If the new edge connects two nodes within the same SCC or if it is a new edge within a single SCC, then it will not affect the number of SCCs. This is because all the nodes in the SCC are already reachable from each other, and the new edge doesn't change this.

adding a new edge can only potentially **increase the size of an existing SCC** or **merge separate S**, but it cannot create a new SCC out of existing ones. CCs

- **Simpler method:** the algorithm for strongly connected components would be simpler
  - if it used the original (instead of the transpose) graph in the second depth-first search
  - and scanned the vertices in order of increasing finishing times.

  Does this simpler algorithm always produce correct results?

Reference

- Book: Cormen – Chapter 22 – Section 22.5
- Exercise:
  - 22.5-1: Number of componets change?
  - 22.5-6: Minimize edge list
  - 22.5-7: Semiconnected graph