

CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #14

► Depth First Search & Applications

Prof. Pietro S. Oliveto

Department of Computer Science and Engineering

Southern University of Science and Technology (SUSTech)

`olivetop@sustech.edu.cn`

<https://faculty.sustech.edu.cn/olivetop>

Reading: Chapter 20 and

I. Wegener. A simplified correctness proof for a well-known algorithm computing strongly connected components. Information Processing Letters 83(1), pages 17–19 – On Blackboard

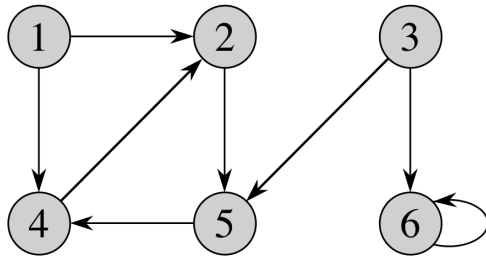
► Aims for this lecture

- Recap **depth-first search (DFS)** and depth-first trees.
- To show how DFS can **classify edges** for additional information about the graph.
- To show how to use DFS to
 - Check whether a graph contains cycles
 - Put tasks in the right order (topological sorting)
 - Compute strongly connected components in graphs
- To show the **correctness** of some remarkable algorithms.

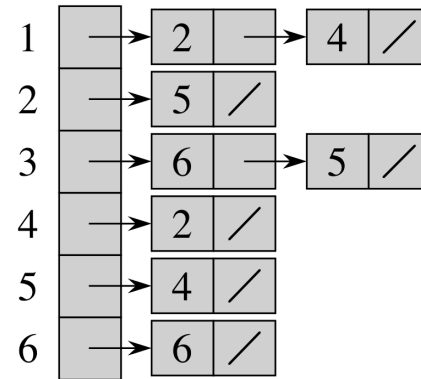
► Representations of graphs

- Using terminology for graphs $G = (V, E)$ from Appendix B
- **Adjacency-list representation:**
 - Array Adj of $|V|$ lists, one for each vertex.
 - The list Adj[u] contains all vertices v adjacent to u in G , i.e. there is an edge $(u, v) \in E$.
 - The sum of all adjacency list lengths equals $|E|$.
- **Adjacency-matrix representation:**
 - Assume that vertices are numbered $1, 2, \dots, n$.
 - Adjacency matrix is a $|V| \times |V|$ matrix with entries $a_{ij} = 1$ if $(i, j) \in E$ and $a_{ij} = 0$ otherwise.

► Example for a directed graph



(a)

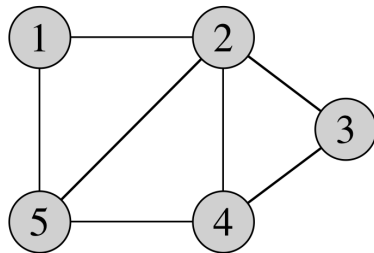


(b)

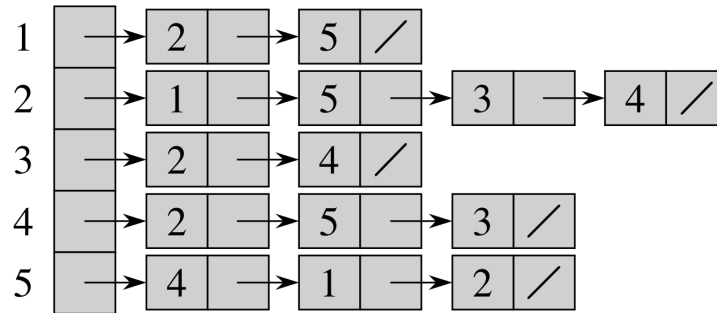
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

► Example for an undirected graph



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

- For every undirected edge $\{u, v\}$, v is in u 's adjacency list and u is in v 's adjacency list.
- Note the symmetry in the adjacency matrix along the main diagonal. It's sufficient to store the entries on and above the diagonal.

► Depth-first search (DFS)

- Works for undirected and directed graphs.
- Ideas:
 - Go into depth by exploring edges out of the most recently discovered vertex and backtrack when stuck.
 - Continue until all vertices reachable from the start vertex are discovered.
 - If any undiscovered vertices remain, continue with one of them as new source.
- As for BFS, define predecessors $v.\pi$ that represent several **depth-first trees**.
- These trees form a **depth-first forest**.

► DFS: Colours and timestamps

- DFS uses colours white, gray, black as for BFS:
 - **White**: vertex has not been discovered yet
 - **Gray**: vertex has been discovered, but is not finished yet.
 - **Black**: vertex has been finished (finished scan of adjacency list).
- Also uses **timestamps**:
 - **v.d** is the time v is first **discovered** (and grayed)
 - **v.f** is the time v is **finished** (and blackened)
 - Global variable time is incremented with each event
 - Hence for all vertices $v.d < v.f$

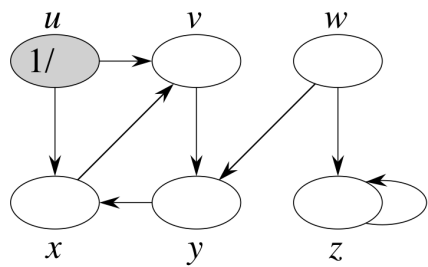
► DFS: Pseudocode

DFS(G)

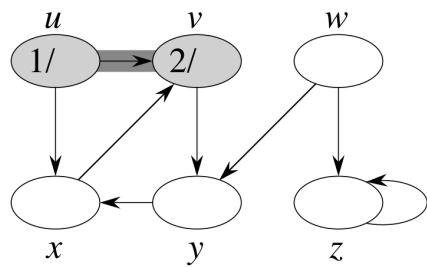
```
1: for each vertex  $u \in V$  do
2:    $u.colour = \text{white}$ 
3:    $u.\pi = \text{NIL}$ 
4:  $time = 0$ 
5: for each vertex  $u \in V$  do
6:   if  $u.colour == \text{white}$  then
7:     DFS-VISIT( $G, u$ )
```

DFS-VISIT(G, u)

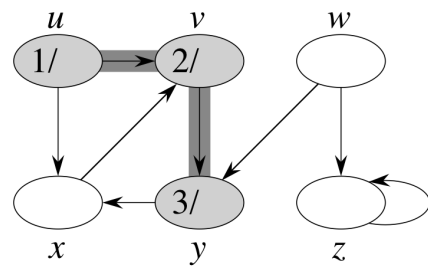
```
1:  $time = time + 1$ 
2:  $u.d = time$ 
3:  $u.colour = \text{gray}$ 
4: for each  $v \in \text{Adj}[u]$  do
5:   if  $v.colour == \text{white}$  then
6:      $v.\pi = u$ 
7:     DFS-VISIT( $G, v$ )
8:  $u.colour = \text{black}$ 
9:  $time = time + 1$ 
10:  $u.f = time$ 
```



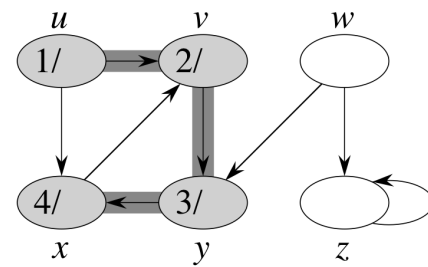
(a)



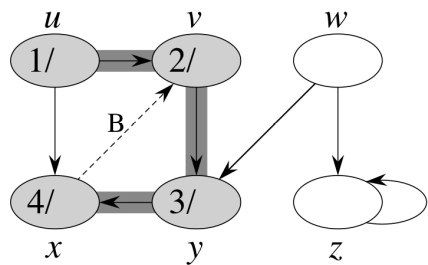
(b)



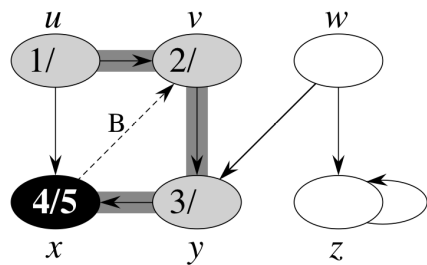
(c)



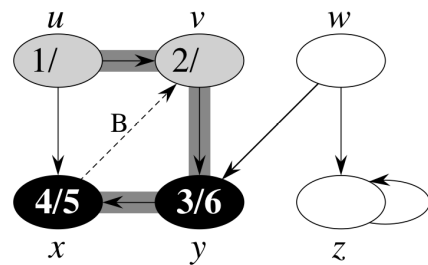
(d)



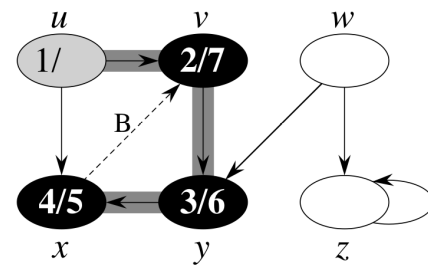
(e)



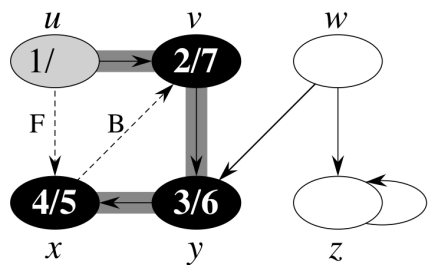
(f)



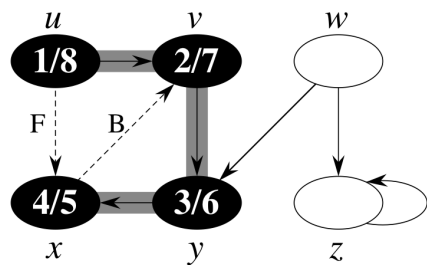
(g)



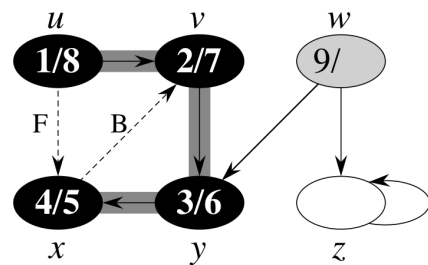
(h)



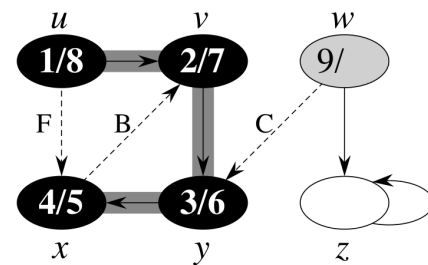
(i)



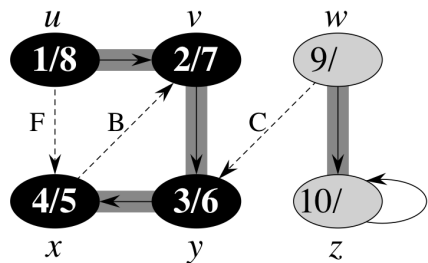
(j)



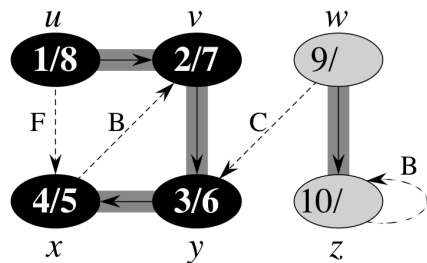
(k)



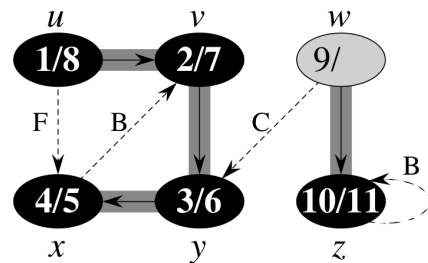
(l)



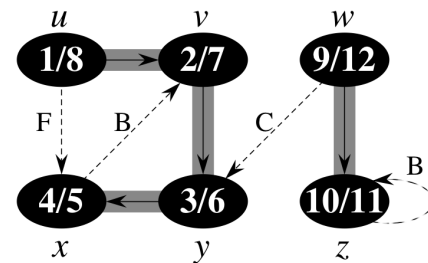
(m)



(n)



(o)



(p)

► DFS: Pseudocode and runtime

DFS(G)

```
1: for each vertex  $u \in V$  do
2:    $u.colour = \text{white}$ 
3:    $u.\pi = \text{NIL}$ 
4:  $time = 0$ 
5: for each vertex  $u \in V$  do
6:   if  $u.colour == \text{white}$  then
7:     DFS-VISIT( $G, u$ )
```

DFS-VISIT(G, u)

```
1:  $time = time + 1$ 
2:  $u.d = time$ 
3:  $u.colour = \text{gray}$ 
4: for each  $v \in \text{Adj}[u]$  do
5:   if  $v.colour == \text{white}$  then
6:      $v.\pi = u$ 
7:     DFS-VISIT( $G, v$ )
8:  $u.colour = \text{black}$ 
9:  $time = time + 1$ 
10:  $u.f = time$ 
```

Runtime?

- Runtime is $\Theta(|V| + |E|)$:
 - DFS runs in time $\Theta(|V|)$ exclusive of the time for DFS-Visit.
 - DFS-Visit is only called once for each vertex v as v must be white and is grayed immediately. The loop executes $|\text{Adj}[u]|$ times.
 - Since $\sum_{v \in V} |\text{Adj}[v]| = \Theta(|E|)$, the total cost for loop is $\Theta(|E|)$.

► Properties of DFS

Parenthesis structure: In any DFS of a (directed or undirected) graph, for any two vertices $u \neq v$, either

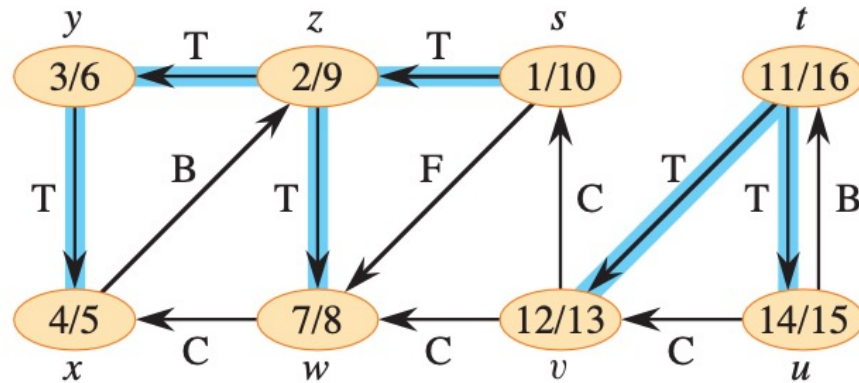
- DFS-Visit(v) is called during DFS-Visit(u), then **v is a descendant of u** and DFS-Visit(v) finishes earlier than u :

$$u.d < v.d < v.f < u.f$$

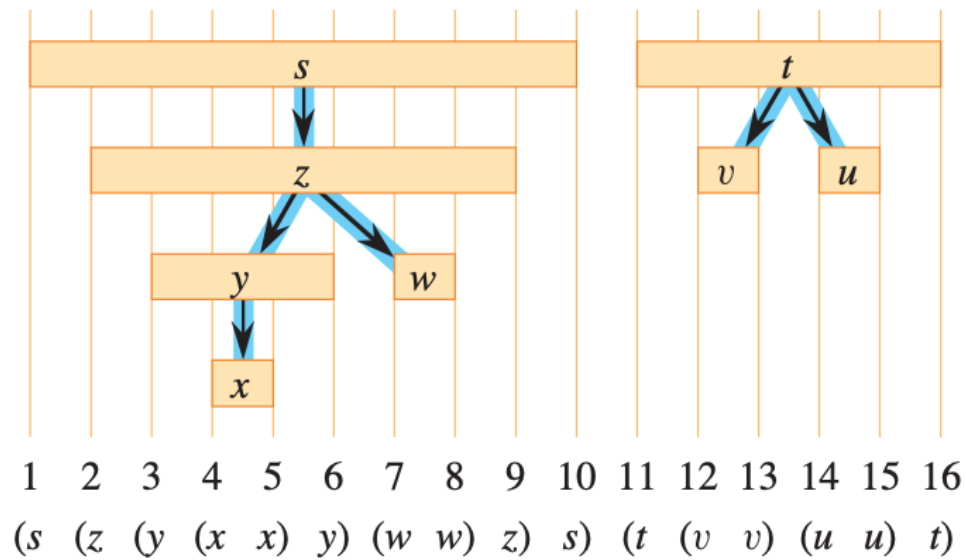
- the same happens with roles of v and u swapped, or
- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and **neither u nor v is a descendant of the other.**

NB: Recursive calls mean that DFS implicitly uses a **stack** to store vertices while exploring the graph (cf. BFS using a queue).

► Paranthesis structure: example



(a)



(b)

► White-path theorem

Theorem 22.9: In a depth-first forest of a (directed or undirected) graph, vertex v is a descendant of a vertex u **if and only if** at the time $u.d$ that the search discovers u , there is a path from u to v consisting entirely of white vertices.

- This means: **if and only if** there is a white path from u to v , DFS will create a DFS tree with edges from u to v .
- “If and only if” indicates a statement like “ $A \Leftrightarrow B$ ”
- We split this into two steps:
 1. Prove that $A \Rightarrow B$
 2. Prove that $A \Leftarrow B$
- It is often easier to focus on proving one implication.

► White-path theorem (2)

Theorem 22.9: In a depth-first forest of a (directed or undirected) graph, vertex v is a descendant of a vertex u **if and only if** at the time $u.d$ that the search discovers u , there is a path from u to v consisting entirely of white vertices.

Proof of “ \Rightarrow ” (being descendant implies white path):

- If $u = v$ then u is still white when $u.d$ is set, thus a white path from u to v exists (just one vertex $u = v$).
- If v is a proper descendant of u , then $u.d < v.d$ and therefore v is white at time $u.d$. This holds for all descendants of u , hence a white path from u to v exists at time $u.d$.

► White-path theorem (3)

Theorem 22.9: In a depth-first forest of a (directed or undirected) graph, vertex v is a descendant of a vertex u **if and only if** at the time $u.d$ that the search discovers u , there is a path from u to v consisting entirely of white vertices.

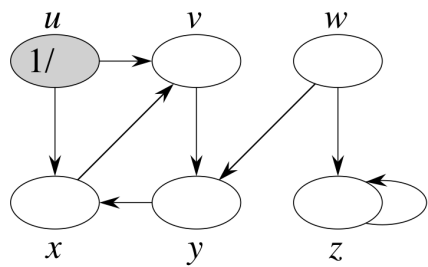
Proof of “ \Leftarrow ” (white path implies descendancy) **by contradiction:**

- Suppose there is a white path from u to v at time $u.d$.
- Assume v is the **first vertex on the path** which is **not a descendant of u** (otherwise we consider this first vertex instead).
- Let w be the predecessor of v on the path (could be $w = u$). Hence w must be a descendant of u (by above assumption). Thus $w.f \leq u.f$.
- v is discovered after u but before w is finished (as there is an edge from w to v), so we get: $u.d < v.d < w.f \leq u.f$.
- How large is $v.f$? Parenthesis structure tells us that $u.d < v.d < v.f < u.f$ is the only feasible case for $v.f$ and so v must be a descendant of u .

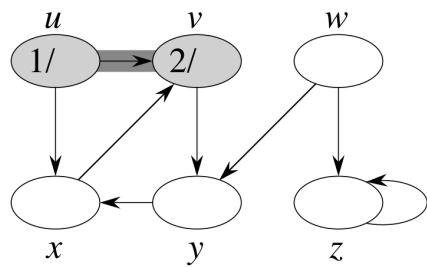
► Classification of edges in directed graphs

DFS can be used to classify edges of the input graph.

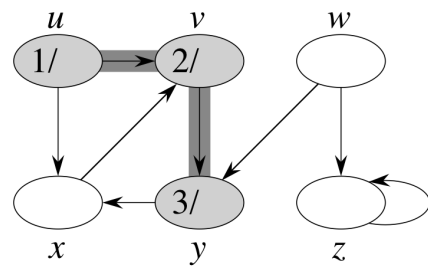
1. **Tree edges** are edges in the depth-first forest. Edge (u, v) is a tree edge if, v was first discovered by exploring edge (u, v) .
An edge (u, v) is a tree edge if at the time of exploration v is white.
2. **Back edges** are edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree (or self-loops in directed graphs).
An edge (u, v) is a back edge if at the time of exploration v is gray.
3. **Forward edges** are nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree (pointing forward in the tree).
 *(u, v) is a forward edge if v is black and was discovered later:
 $u.d < v.d$.*
4. **Cross edges** are all other edges: either leading to a subtree constructed earlier or leading to a different (earlier) depth-first tree.
 (u, v) is a cross edge if v is black and was discovered earlier: $u.d > v.d$.



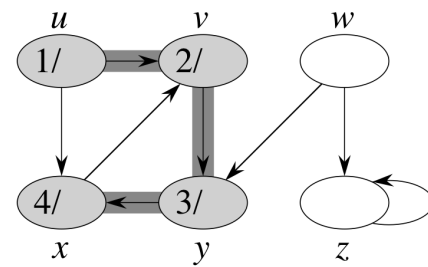
(a)



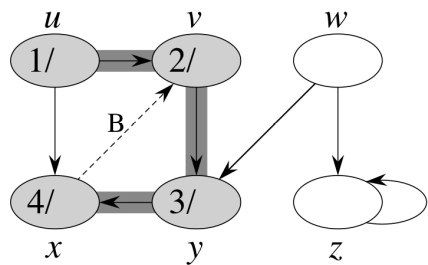
(b)



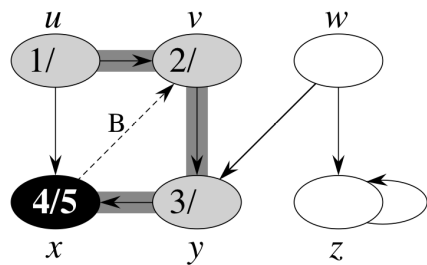
(c)



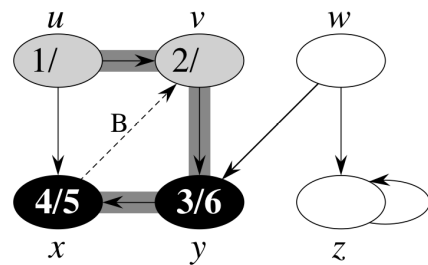
(d)



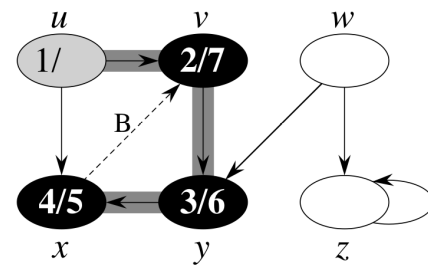
(e)



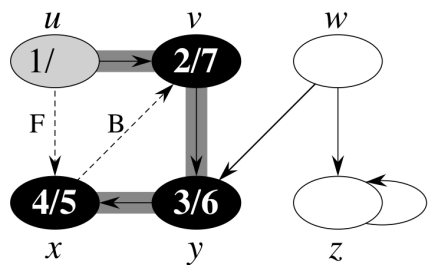
(f)



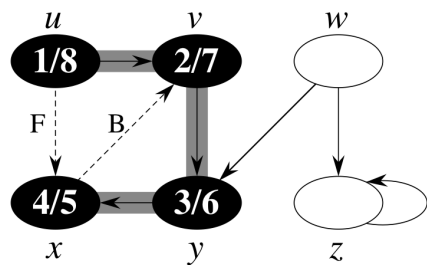
(g)



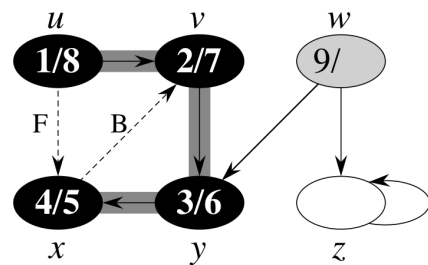
(h)



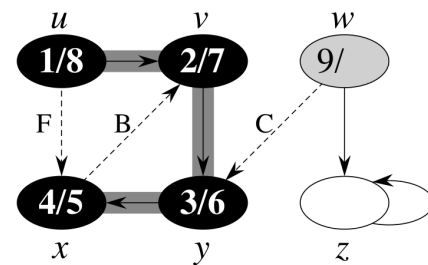
(i)



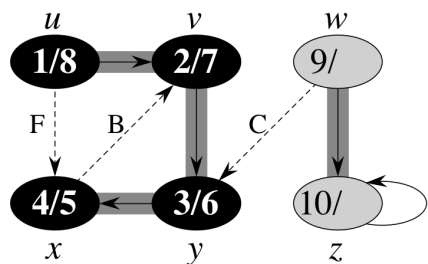
(j)



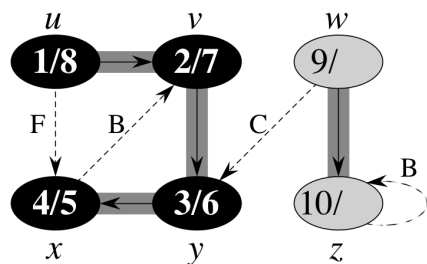
(k)



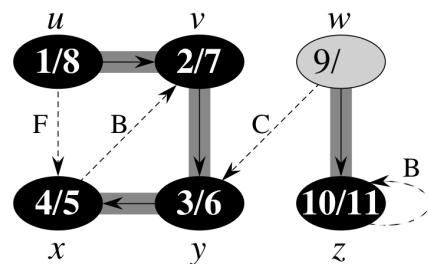
(l)



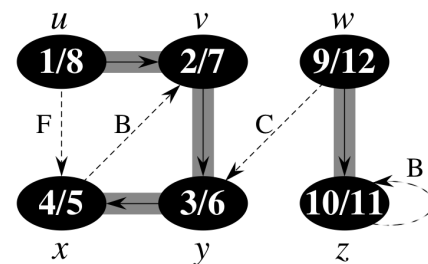
(m)



(n)

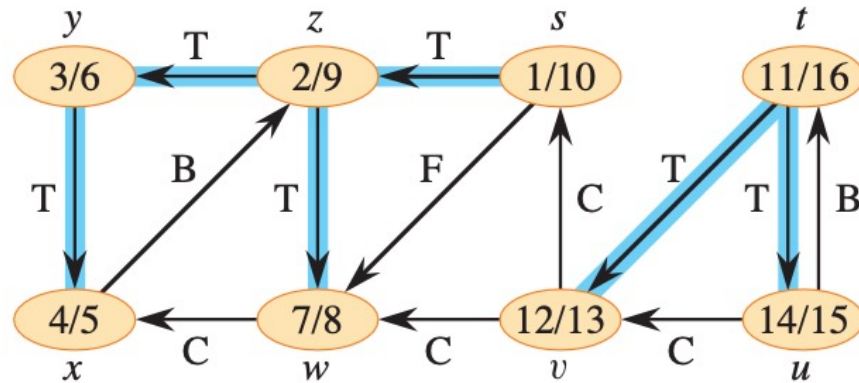


(o)

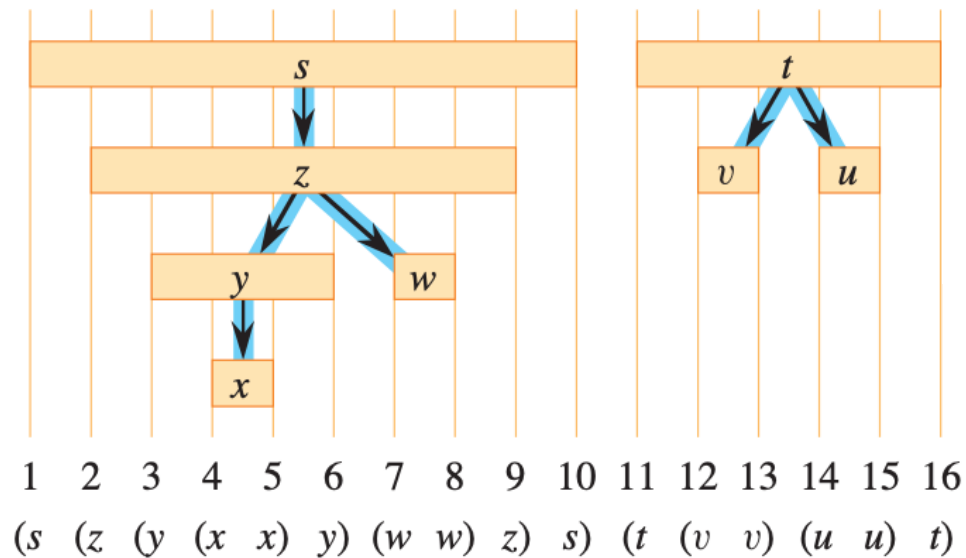


(p)

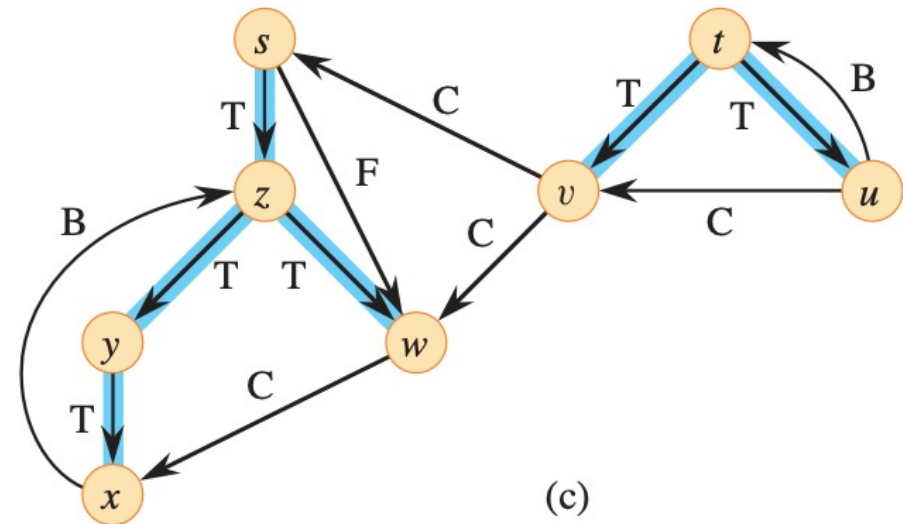
► Edge classification: example



(a)



(b)



(c)

► Edge classification in undirected graphs

Theorem 22.10: In a depth-first search of an **undirected** graph, every edge is either a tree edge or a back edge.

→ There are **no forward/cross edges** in undirected graphs.

Proof:

- Let $\{u, v\}$ be an arbitrary edge, and assume without loss of generality that $u.d < v.d$.
- Since v is on u 's adjacency list, search must discover and finish v before it finishes u .
- If the first time the edge is explored, it is in the direction from u to v , then v is undiscovered and it becomes a **tree edge**.
- If the edge is first explored from v to u , then it becomes a **back edge**, since u is still gray.

► Precedence graphs

- Graphs have many applications. One of them is modelling precedences:
 - Vertices represent tasks
 - A edge (u, v) means that task u has to be executed before task v .
- Coming up: how to order tasks such that all precedence constraints are respected.
- But this is only feasible if the precedence graph does not contain any cycles!
- Such a graph is called **acyclic**.

► Application of DFS: testing for cycles

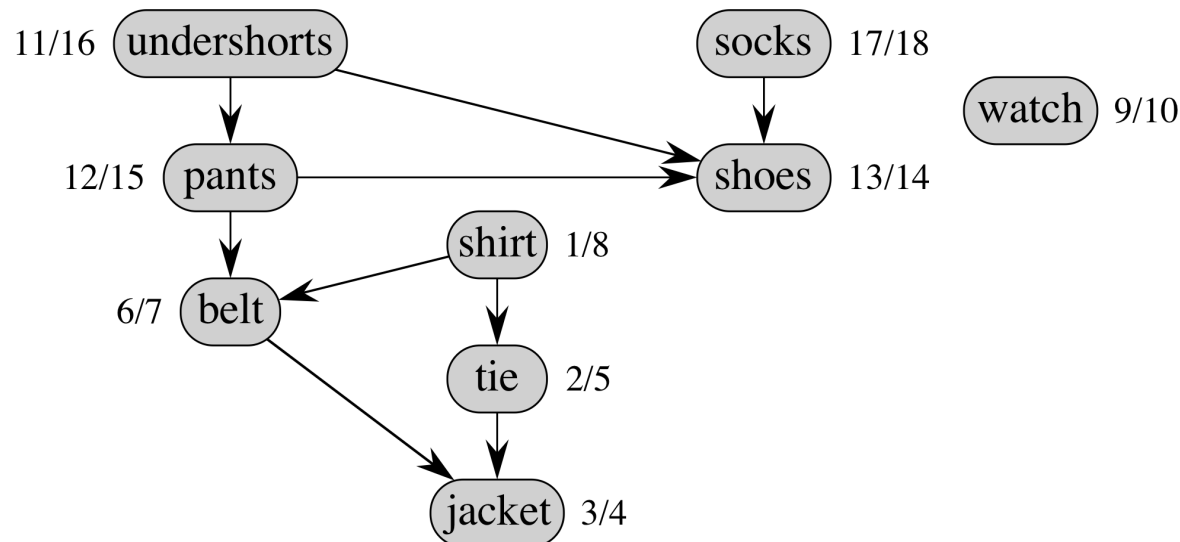
Theorem (adapted from Lemma 22.11): A graph G contains a cycle if and only if DFS finds at least one back edge.

Proof (for directed graphs):

- “ \Leftarrow ”: Suppose DFS produces a back edge (u, v) . Then v is an ancestor of u in the depth-first tree. Thus, G contains a path (of tree edges) from v to u , and the back edge completes a cycle.
- “ \Rightarrow ”: Suppose that G contains a cycle C . We show that DFS yields a back edge. Let v be the first vertex to be discovered in C , and let (u, v) be the edge on C going into v . At time $v.d$, the vertices of C form a path of white vertices from v to u . By the white-path theorem, u becomes a descendant of v . Therefore, (u, v) is a back edge.

► Topological sorting

- Consider a directed acyclic graph (“dag”).
- A topological sort of a dag is a linear ordering of all its vertices such that for each edge (u, v) , u appears before v .
- If vertices are arranged on a horizontal line, all edges go from left to right.
- Example: Professor Bumstead getting dressed.

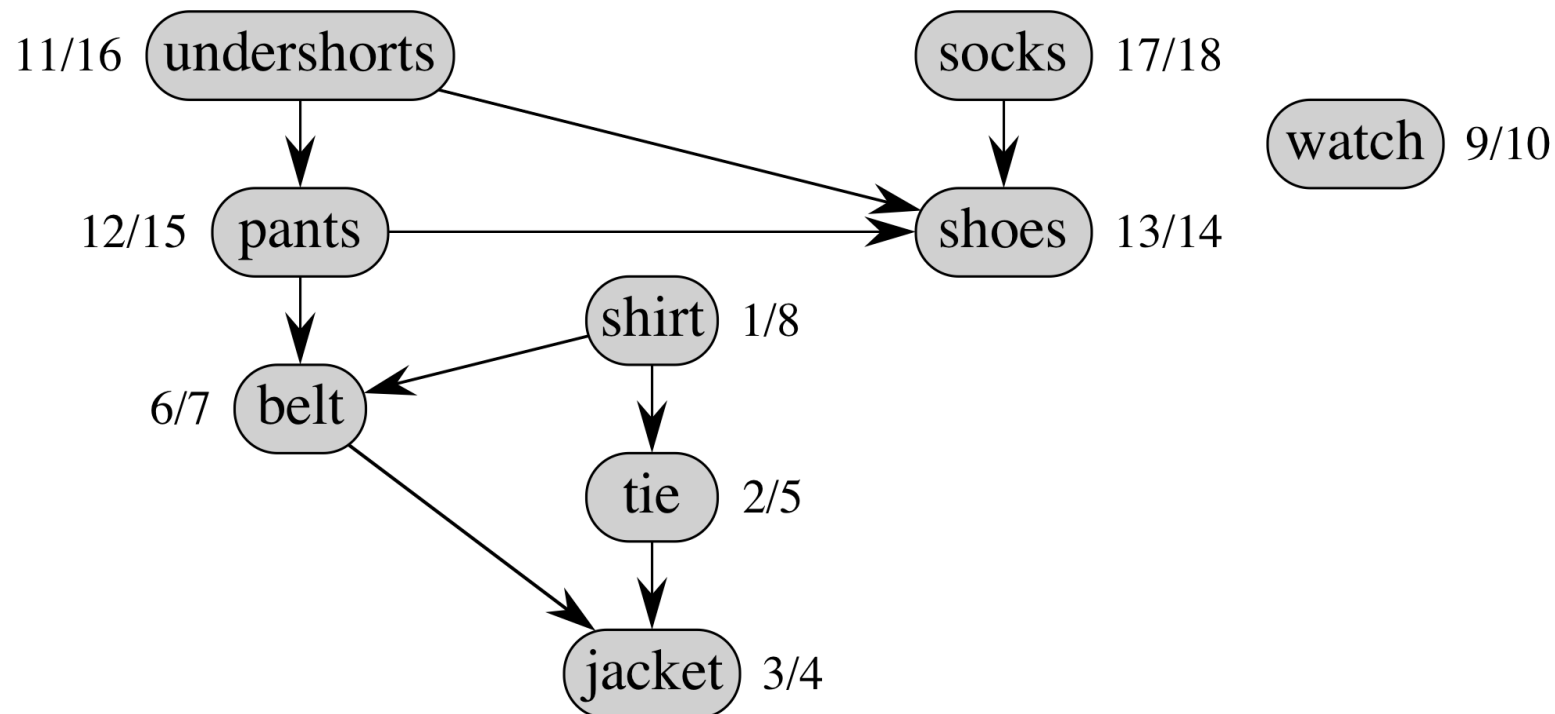


► Computing a topological sort

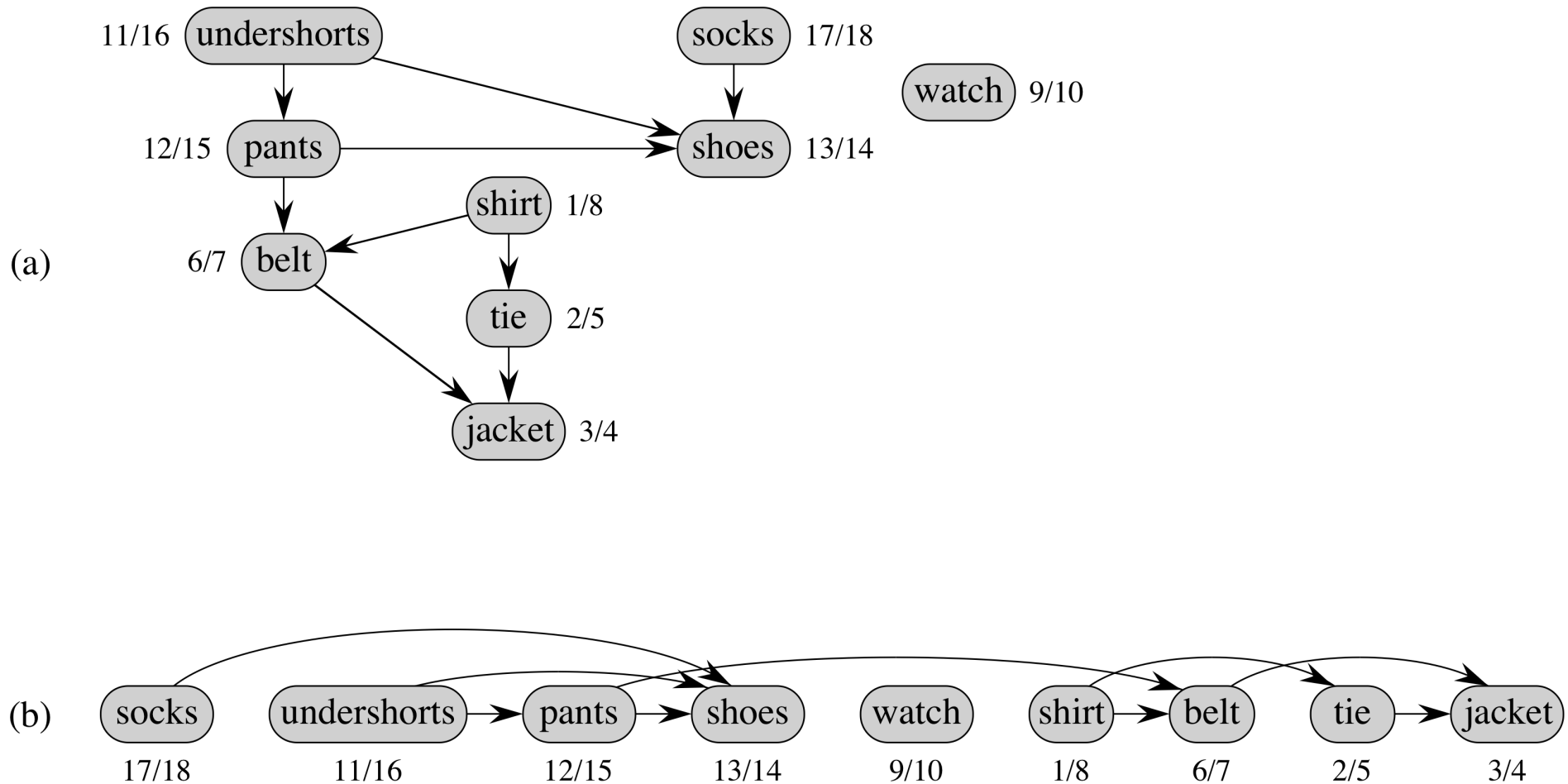
- Here's how to use DFS to compute a topological sort:

TOPOLOGICAL-SORT(G)

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



► Professor Bumstead getting dressed



► Topological sort: Runtime

TOPOLOGICAL-SORT(G)

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

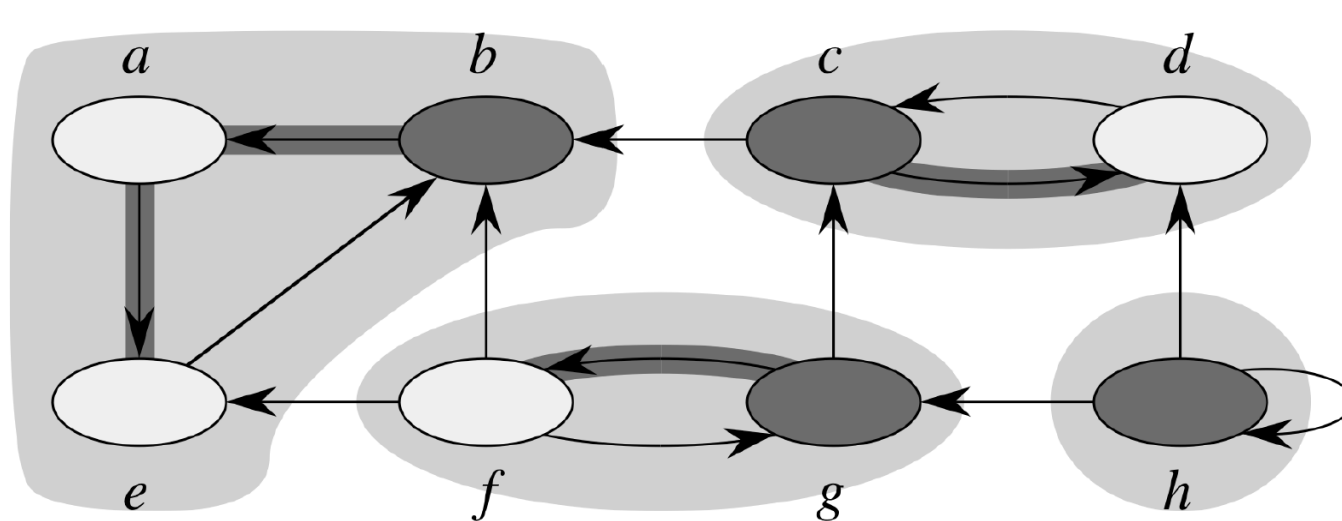
- Runtime:
 - $\Theta(|V| + |E|)$ time for DFS
 - $+O(1)$ for each vertex inserted in to the linked list $\rightarrow +O(|V|)$
 - Total time $\Theta(|V| + |E|)$
- Why on earth does this work?!

► Topological sort: correctness proof

- Suffices to show that if G contains an edge (u, v) , then $v.f < u.f$. Then v is inserted to the list earlier and will come to rest after u .
- Consider any edge (u, v) explored by DFS. When this edge is explored, v cannot be gray, since then v would be an ancestor of u and (u, v) would be a back edge, contradicting the fact that G is acyclic.
- Therefore, v must be either white or black.
 - If v is white, it becomes a descendant of u , and so $v.f < u.f$ by parenthesis structure.
 - If v is black, it has been finished and $v.f$ has been set. Because we are still exploring from u , a timestamp $u.f$ will be assigned later and once we do, it will be larger: $v.f < u.f$.

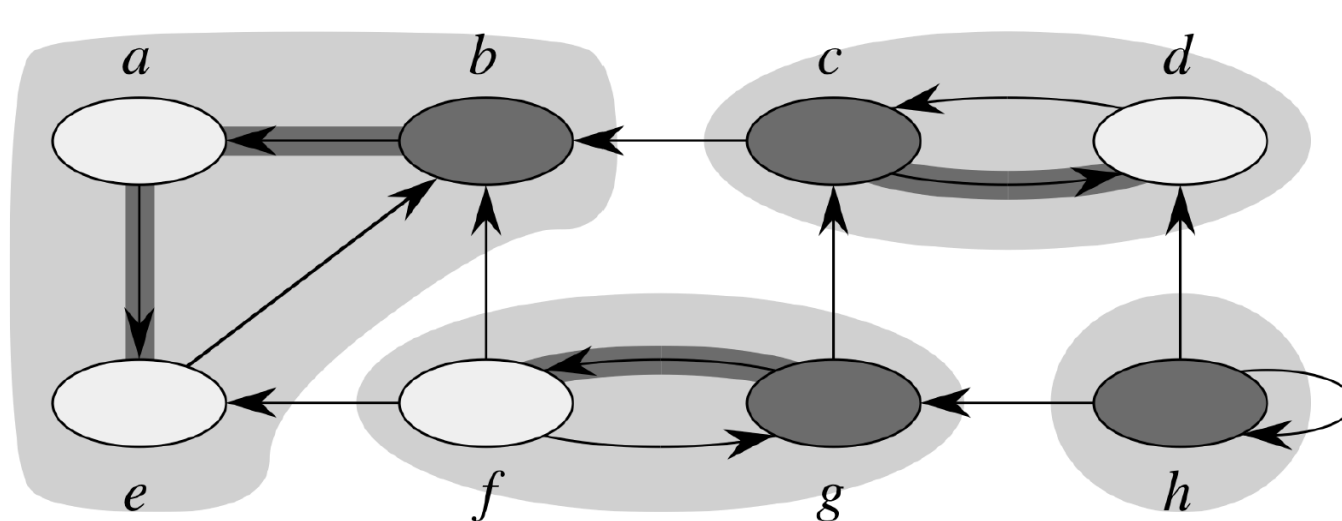
► Strongly connected components

- A directed graph is called **strongly connected** if every two vertices are reachable from each other.
- The **strongly connected components (SCCs)** of a directed graph are the equivalence classes under the “mutually reachable” relation. In other words, they are maximal sets of vertices where all vertices in every set are mutually reachable.



► Strongly connected components

- Applications:
 - Finding groups of friends in social network graphs.
 - Many algorithms working on directed graphs decompose the graph into its SCCs, run separately on all of them, and then combine solutions for all SCCs to one overall solution.



► Computing SCCs with DFS

- Let G^T be the transpose of G , i. e. the graph where all edges have their direction reversed.
- Note that G and G^T have the same SCC as u and v are reachable in G^T if and only if they are reachable in G .
- G^T can be computed in time $O(|V| + |E|)$.

STRONGLY-CONNECTED-COMPONENTS(G)

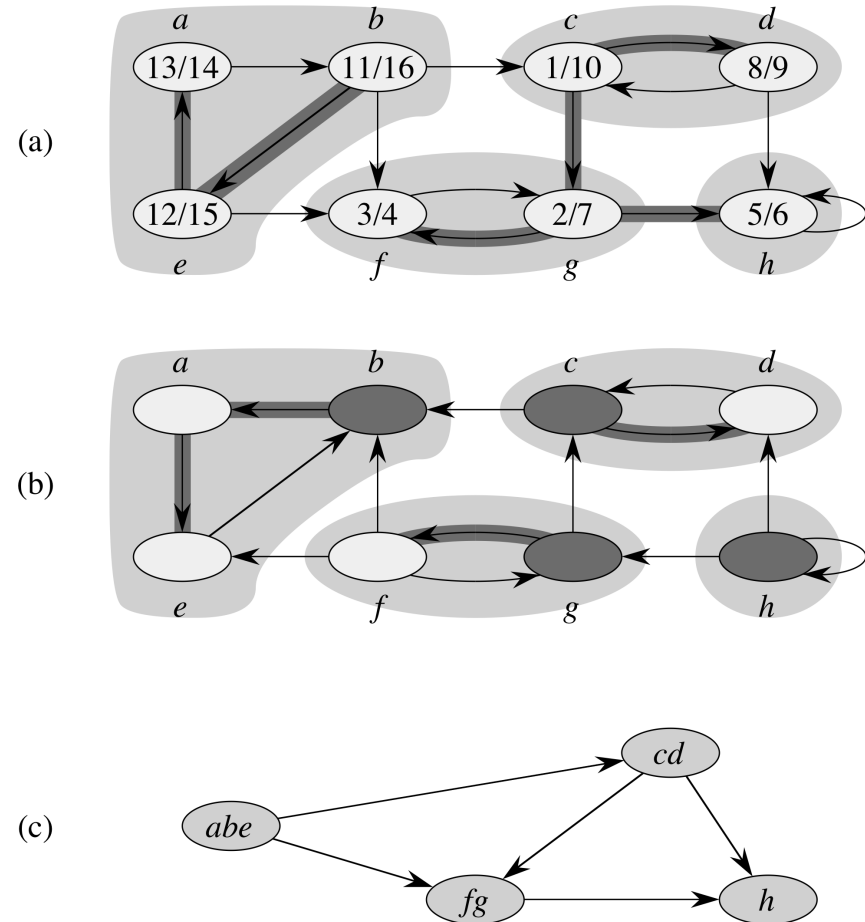
- 1: call DFS(G) to compute finishing times $v.f$ for each vertex v
 - 2: compute G^T
 - 3: call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
 - 4: output the vertices of the tree in the depth-first forest formed in line 3 as a separate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

► Strongly connected components: example

STRONGLY-CONNECTED-COMPONENTS(G)

- 1: call DFS(G) to compute finishing times $v.f$ for each vertex v
 - 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 the tree in the depth-first forest formed in line 3 as a separate SCC
-

Runtime?

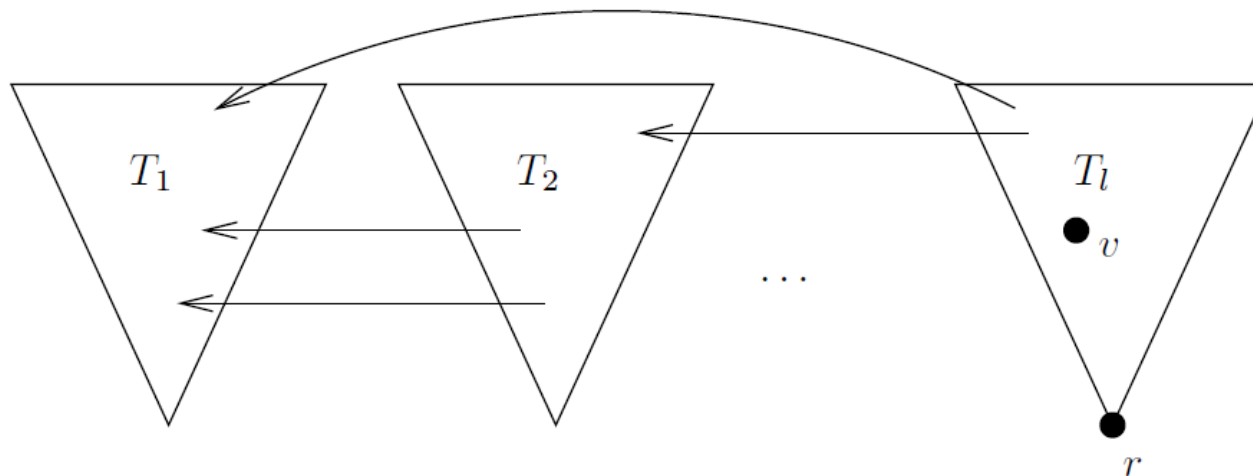


► Correctness of the SCC algorithm

- Why on earth does this work? It's a miracle!
- Proof in the book is 3 pages of lemmas and not very intuitive.
- Let's use a simpler and more intuitive proof by Ingo Wegener:
- *A simplified correctness proof for a well-known algorithm computing strongly connected components*, Information Processing Letters 83(1), pages 17–19 (on Blackboard)

► Correctness (2)

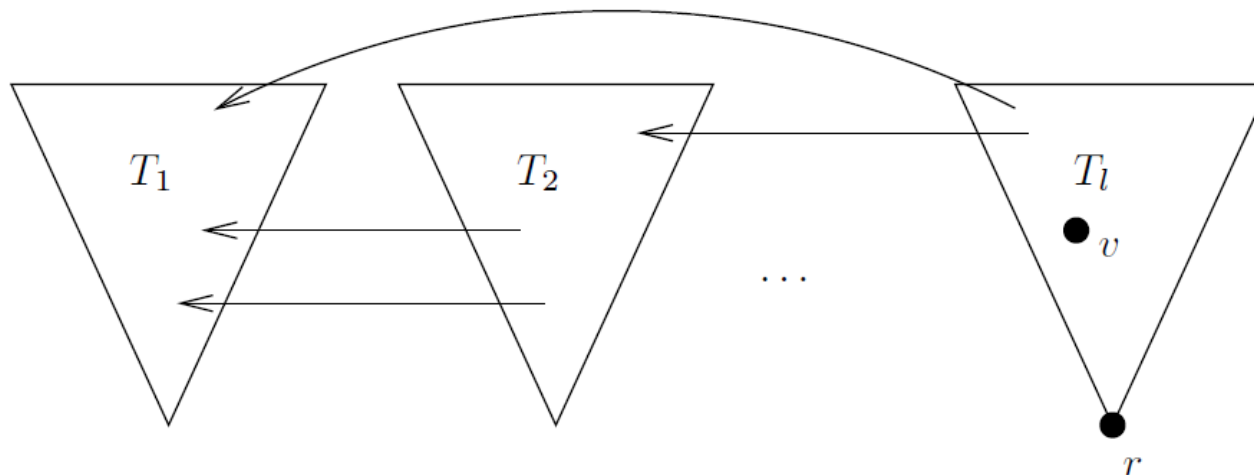
- Draw constructed depth-first trees from left to right and name them T_1, T_2, \dots, T_l .
- Then **edges between trees can only go right to left** (otherwise, e.g. if there is an edge from T_1 to T_2 , parts of T_2 would have been included in the depth-first tree T_1)



- Hence **each SCC must be contained in one of the trees.**

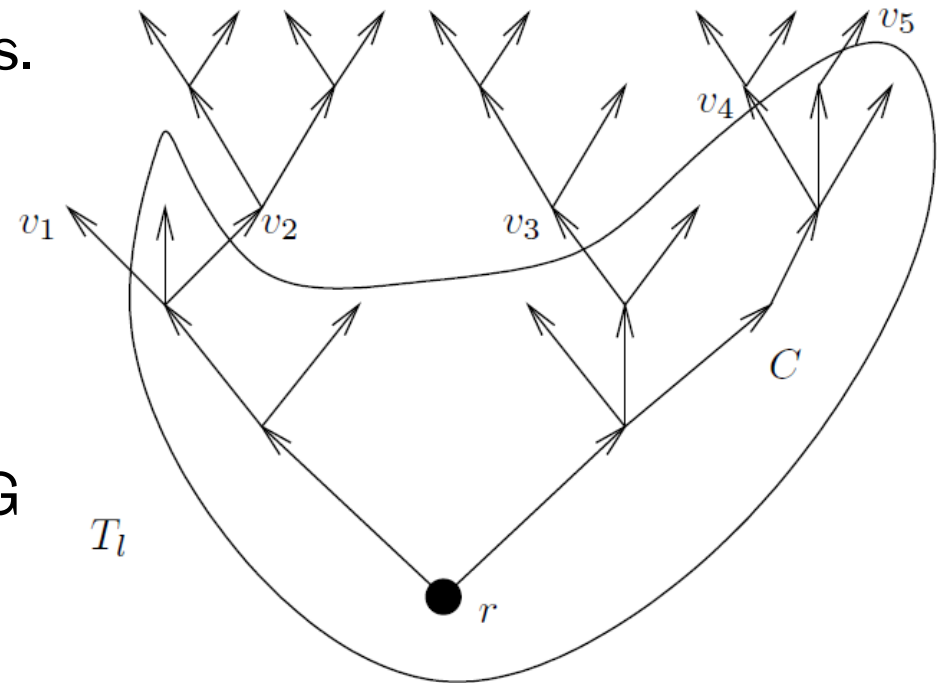
► Correctness (3) – finding a first SCC

- The algorithm starts the second DFS on G^T computing the SCC C containing the **root r of the last tree** (as r finished last).
- We know that there is a path from r to all $v \in T_l$ (tree edges). So C is the set of all vertices v for which there is a path v to r in G . This is the set of **all vertices v reachable from r in G^T** .
- After reversing all edges, DFS from r in G^T cannot leave T_l . Hence DFS in G^T from r outputs **exactly the SCC containing r** .



► Correctness (4) – extracting a first SCC

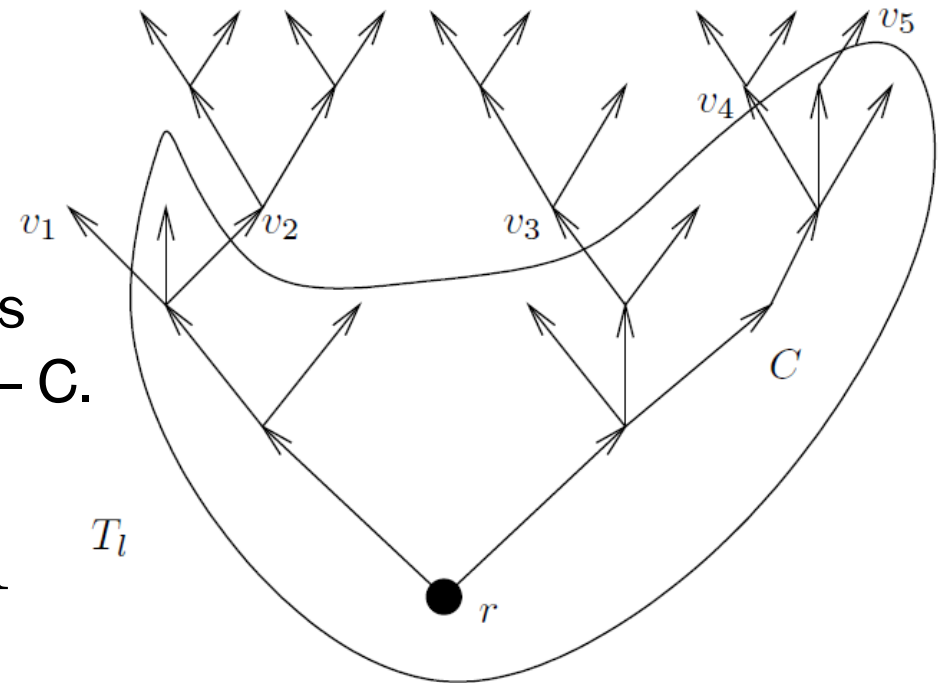
- How does the SCC C containing r look like?
- If v belongs to C , then all vertices on the path r to v must also belong to C (as there is a path from v back to r).
- Hence C is a connected part of T_l .
- T_l without C splits into subtrees.
- T_1, \dots, T_{l-1} along with these subtrees is a depth-first forest which is also the result of a DFS traversal of $G - C$.
- The time stamps from DFS on G also work as time stamps for DFS on $G - C$! (**main insight**)



► Correctness (5) – repeated extraction

Proving correctness by induction over the number of SCCs:

- **Base case:** If the graph is a single SCC, the algorithm outputs it.
- Assume the algorithm is correct for graphs with $k - 1$ SCCs.
- For a graph with k SCCs, the algorithm correctly outputs the SCC C containing the root r of the last DFS tree.
- Algorithm continues with vertices and depth-first (sub-)trees in $G - C$.
- By the induction hypothesis, it then outputs the remaining $k - 1$ SCCs of $G - C$ correctly as well.



► Summary for Depth-First Search

- Depth-first search explores the graph going into depth and using backtracking in time $\Theta(|V| + |E|)$.
- DFS classifies edges into **tree**, **back**, **forward**, and **cross edges**.
- DFS is used to test whether a graph is **acyclic** in time $\Theta(|V| + |E|)$. Can be improved to $O(|V|)$ for **undirected** graphs (exercise!).
- DFS is used for **topological sorting** in directed acyclic graphs in time $\Theta(|V| + |E|)$.
- DFS is used to determine **strongly connected components** in graphs in time $\Theta(|V| + |E|)$.
- Seen detailed **correctness proofs** to demystify algorithms that appear magical at first glance.