Algorithms
Chapter 22
Elementary Graph Algorithms

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

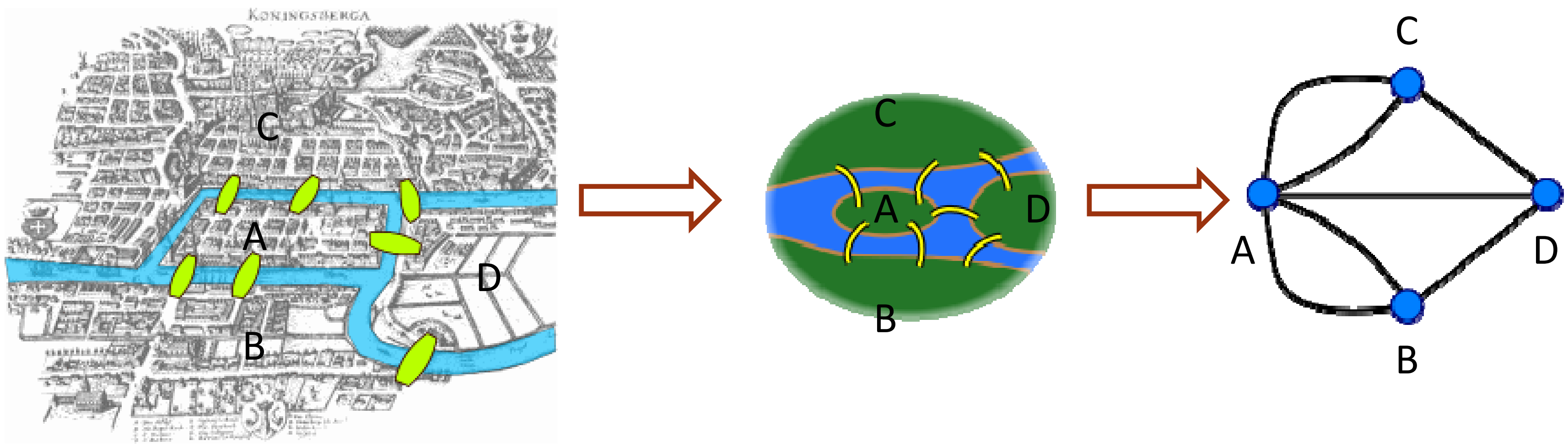
Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Representations of graphs**
- Breadth-first search
- Depth-first search
- Topological sort
- Strongly connected components

Konigsberg Bridge Problem

- Can we walk across all the bridges **exactly once** in returning back to the starting land area ?



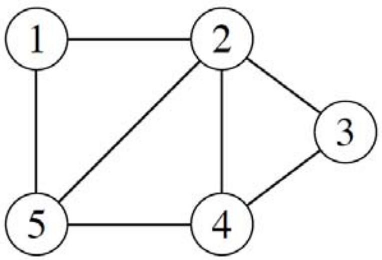
- Transferring to Graph model
  - Land → vertex
  - Bridge → edge

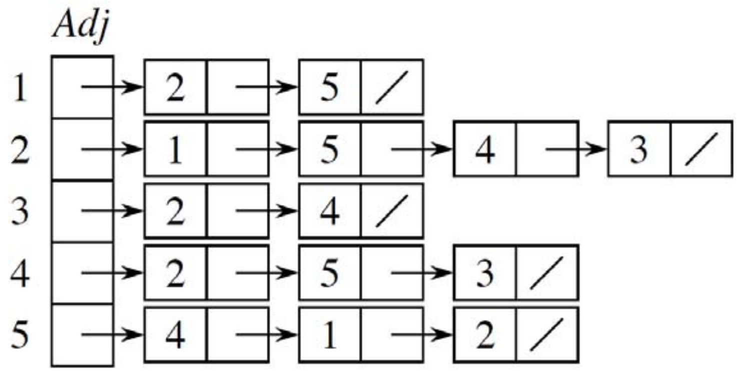
Graph representation

- Given a graph $G = (V, E)$.
  - May be either directed or undirected.
  - Two standard ways to represent a graph:
    - Adjacency lists, when the graph is **sparse**.
    - Adjacency matrix, when the graph is **dense**.
- When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$, where $|V| = n$ and $|E| = m$.
  - Example: $O(n+m)$.

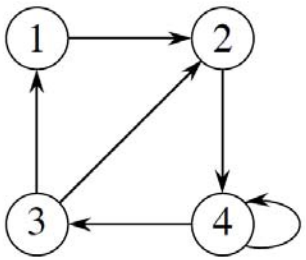
Adjacency lists$_{1/2}$

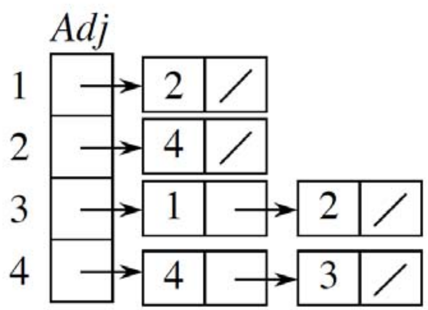
- Example: For an undirected graph:





- Example: For a directed graph:



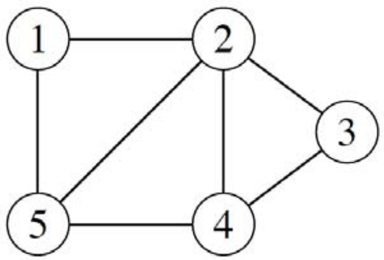


Adjacency lists$_{2/2}$

- Array *Adj* of ***n*** lists, one per vertex.
- Vertex *u*'s list has all vertices *v* such that $(u, v) \in E$.
- If edges have **weights**, can put the weights in the lists.
  - Weight: $w : E \rightarrow R$.
- Space: $\Theta(n + m)$.
- Time:
  - list all vertices adjacent to *u*: $\Theta(\deg(u))$.
  - determine if $(u, v) \in E$: $\Theta(\deg(u))$.

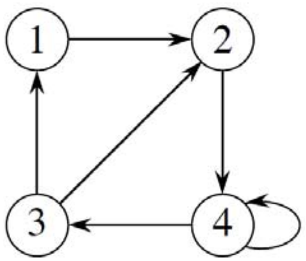
Adjacency matrix$_{1/2}$

- Example: For an undirected graph:



	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

- Example: For a directed graph:



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

Adjacency matrix$_{2/2}$

- A is an ***n*** x ***n*** matrix such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

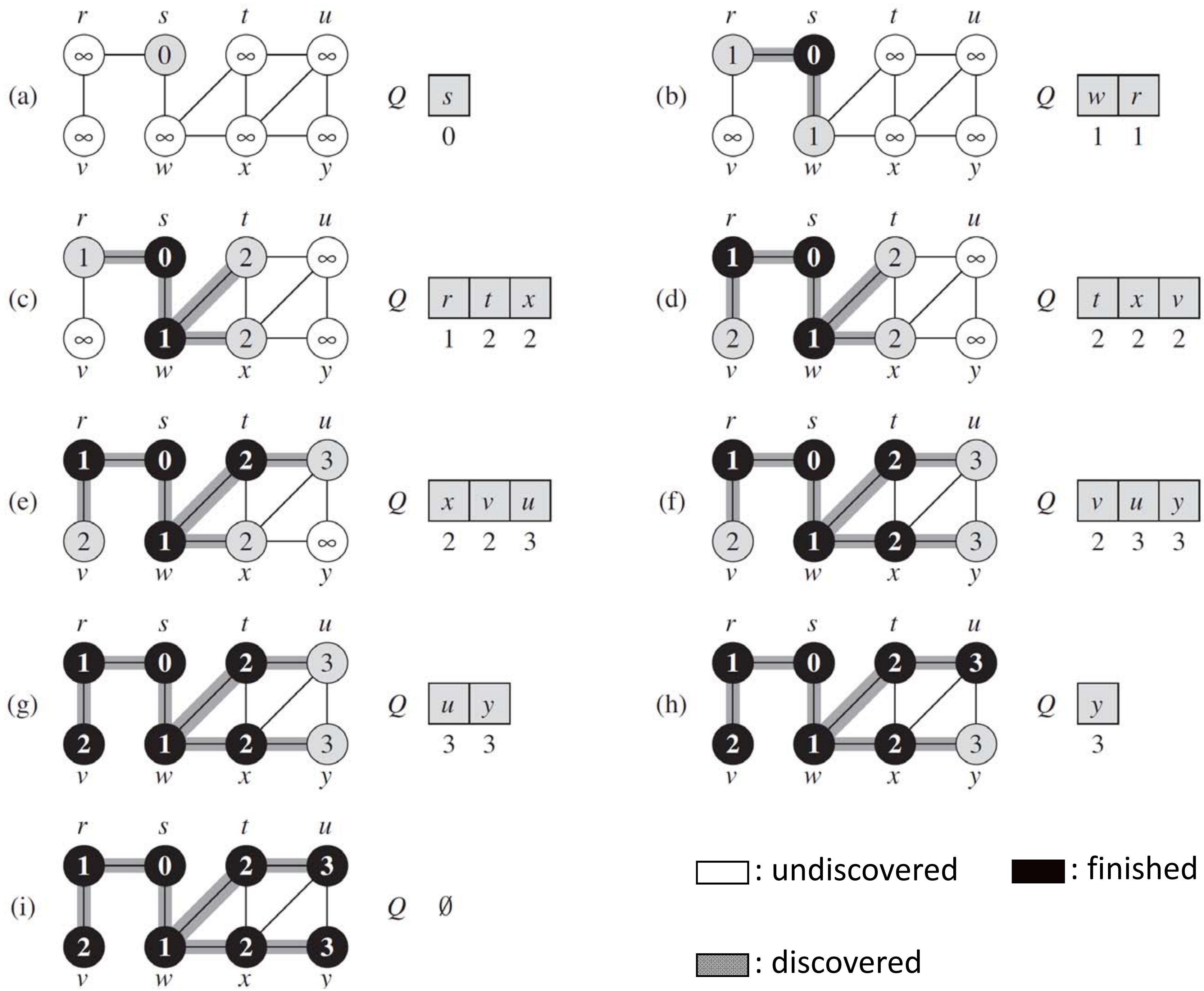
- Can store weights instead of bits for weighted graph.
- Space: $\Theta(n^2)$.
- Time:
  - list all vertices adjacent to u: $\Theta(n)$.
  - determine if $(u, v) \in E$: $\Theta(1)$.

Outline

- Representations of graphs
- **Breadth-first search**
- Depth-first search
- Topological sort
- Strongly connected components

Breadth-first search

- **Input:** A graph $G = (V, E)$ and a distinguished ***source*** vertex s.
  - G can either be directed or undirected.
- **Output:** The distance (smallest number of edges) from s to each reachable vertex.
  - As a by-product, it computes a "**breadth-first tree**" with root s that contains all reachable vertices.
- **Idea:** Discover all vertices at distance k from s before discovering any vertices at distance $k + 1$.
  - First hits all vertices 1 edge from s.
  - From there, hits all vertices 2 edges from s.
  - And so on.


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
Q
: undiscovered
: finished
: discovered

Pseudocode

```
BFS(G, s)
    for each vertex u ∈ V [G] – {s}
        color[u] ← WHITE
        d[u] ← ∞
        π[u] ← NIL
    color[s] ← GRAY
    d[s] ← 0
    π[s] ← NIL
    Q ← Ø
    ENQUEUE(Q, s)
    while Q ≠ Ø
        u ← DEQUEUE(Q)
        for each v ∈ Adj[u]
            if color[v] = WHITE
                color[v] ← GRAY
                d[v] ← d[u] + 1
                π[v] ← u
                ENQUEUE (Q, v)
        color[u] ← BLACK
```

Complexity

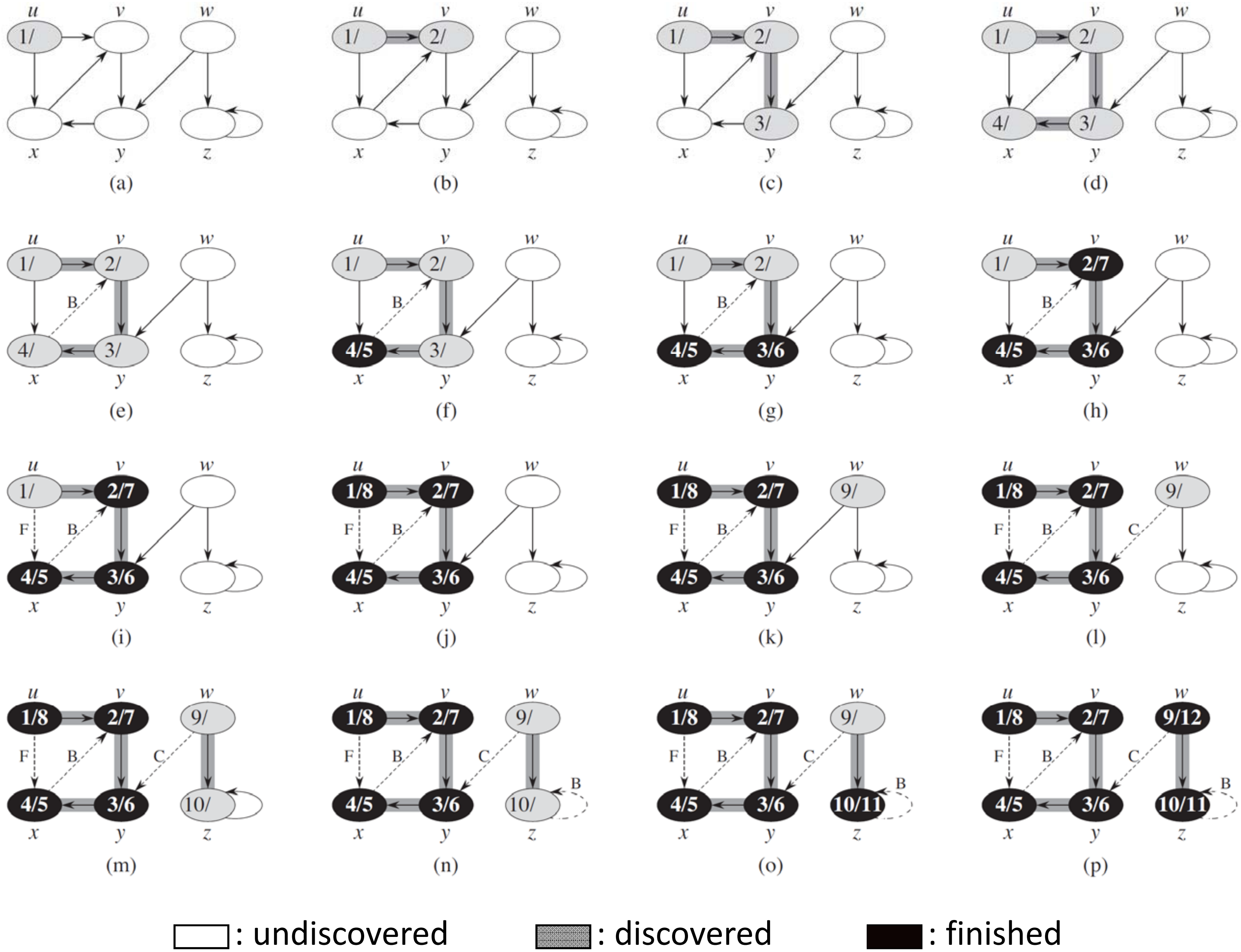
- The algorithm uses a first-in, first-out ***queue*** Q to manage the set of gray vertices.
- $\pi[v]$: the predecessor of v.
- Breadth-first tree : $G_\pi = (V_\pi, E_\pi)$
  - $V_\pi = \{v \in V: \pi[v] \neq \text{NIL}\} \cup \{s\}$
  - $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$
- The path in breadth-first tree from s to v is a shortest path (containing the fewest number of edges) from s to v.
- Time: $O(n+m)$.
  - $O(n)$: every vertex enqueued at most once.
  - $O(m)$: using adjacency list, each edge is scanned at most twice.

Outline

- Representations of graphs
- Breadth-first search
- **Depth-first search**
- Topological sort
- Strongly connected components

Depth-first search

- **Input:** A graph $G = (V, E)$. No source vertex is given!
  - G can either be directed or undirected.
- **Output:** Two timestamps: $d[v]$ = discovery time and
  $f[v]$ = finishing time.
  - It also computes a **depth-first forest** $G_\pi = (V, E_\pi)$, where $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$.
- Will methodically explore **every** edge.
  - Start over from different vertices as necessary.
- As soon as we discover a vertex, explore from it.
  - Unlike BFS, which puts a vertex on a queue so that we explore from it later.


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)
: undiscovered
: discovered
: finished

DEPTH-FIRST SEARCH pseudocode$_{1/2}$

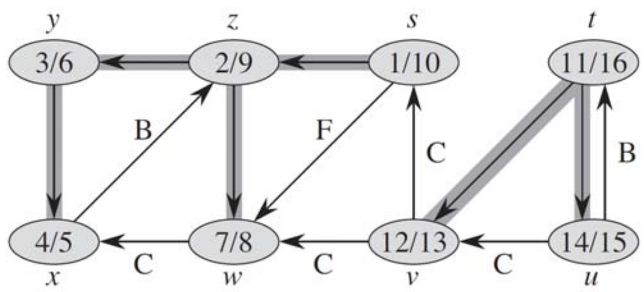
```
DFS(G)
    for each vertex u ∈ V [G]
        color[u] ← WHITE
        π[v] ← NIL
    time ← 0
    for each vertex u ∈ V [G]
        if color[u] = WHITE
            DFS-VISIT(u)

DFS-VISIT(u)
    color[u] ← GRAY             % White vertex u has just been discovered.
    time ← time + 1
    d[u] ← time
    for each v ∈ Adj[u]         % Explore edge(u, v).
        if color[v] = WHITE
            π[v] ← u
            DFS-VISIT(v)
    color[u] ← BLACK            % Blacken u; it is finished.
    f [u] ← time ← time +1
```

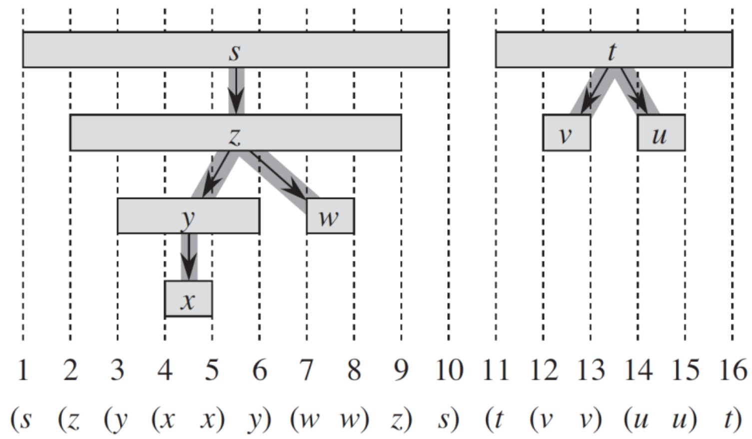
DEPTH-FIRST SEARCH pseudocode$_{2/2}$

- $\pi[v]$: the predecessor of v.
- Discovery and finish times:
  - Unique integers from 1 to $2n$.
  - For all v, $d[v] < f[v]$.
  - In other words, $1 \le d[v] < f[v] \le 2n$.
- Time: $\Theta(n+m)$.
  - $\Theta(n)$: The procedure DFS-VISIT is called exactly once for each vertex $v \in V[G]$.
  - $\Theta(m)$: Using adjacency list, each edge is scanned at most twice.

Properties of depth-first search$_{1/3}$

- Another important property of depth-first search is that discovery and finishing times have **parenthesis structure**.
  - When vertex *u* is discovered ➔ represent *u* with "(*u*".
  - When vertex *u* is finished ➔ represent *u* with "*u*)".





Properties of depth-first search$_{2/3}$

- **Theorem 22.7** (Parenthesis theorem)
  For any two vertices *u* and *v*, exactly one of the following three conditions holds:
  - the intervals [*d*[*u*], *f*[*u*]] and [*d*[*v*], *f*[*v*]] are entirely disjoint, and neither *u* nor *v* is a descendant of the other in the depth-first forest,
  - the interval [*d*[*u*], *f*[*u*]] is contained entirely within the interval [*d*[*v*], *f*[*v*]], and *u* is a descendant of *v* in a depth-first tree, or
  - the interval [*d*[*v*], *f*[*v*]] is contained entirely within the interval [*d*[*u*], *f*[*u*]], and *v* is a descendant of *u* in a depth-first tree.

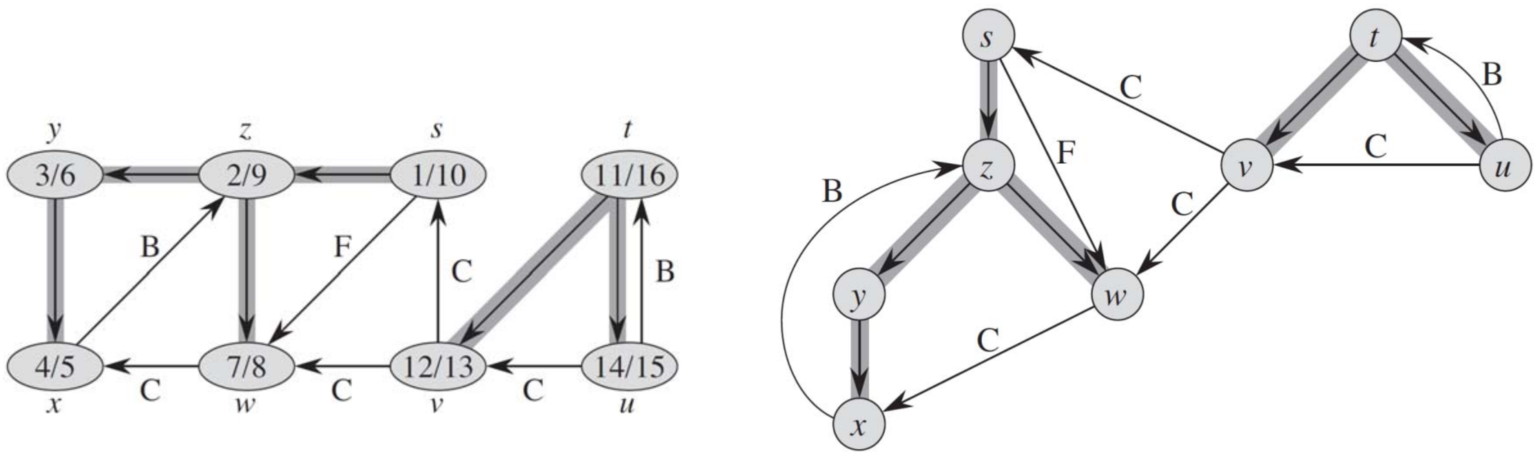
Properties of depth-first search$_{3/3}$

- **Corollary 22.8** (Nesting of Descendants' Intervals)
  Vertex *v* is a proper descendant of vertex *u* in the depth-first forest for a graph *G* if and only if $d[u] < d[v] < f[v] < f[u]$.

- **Theorem 22.9** (White-path theorem)
  Vertex *v* is a descendant of vertex *u* if and only if at the time $d[u]$, there is a *u*-*v* path consisting of only white vertices.

Classification of edges

- Four edge types:
  - **Tree edge:** in the depth-first forest G_{π}.
  - **Back edge:** non-tree edge (*u, v*) such that *u* is a descendant of *v*.
    (including self-loop)
  - **Forward edge:** non-tree edge (*u, v*) such that *u* is an ancestor of *v*.
  - **Cross edge:** non-tree edge (*u, v*) such that *u* is neither a descendant nor an ancestor of *v*.



Modify DFS algorithm to classify edges

- **Idea :** Each edge (u, v) can be classified by the color of the vertex v that is reached when the edge is first explored.
  - **White:** tree edge.
  - **Gray:** back edge.
  - **Black:** forward edge if $d[u] < d[v]$ and cross edge if $d[u] > d[v]$.

- If G is an undirected graph, an edge is classified as the **first** type that applies.

Theorem 22.10

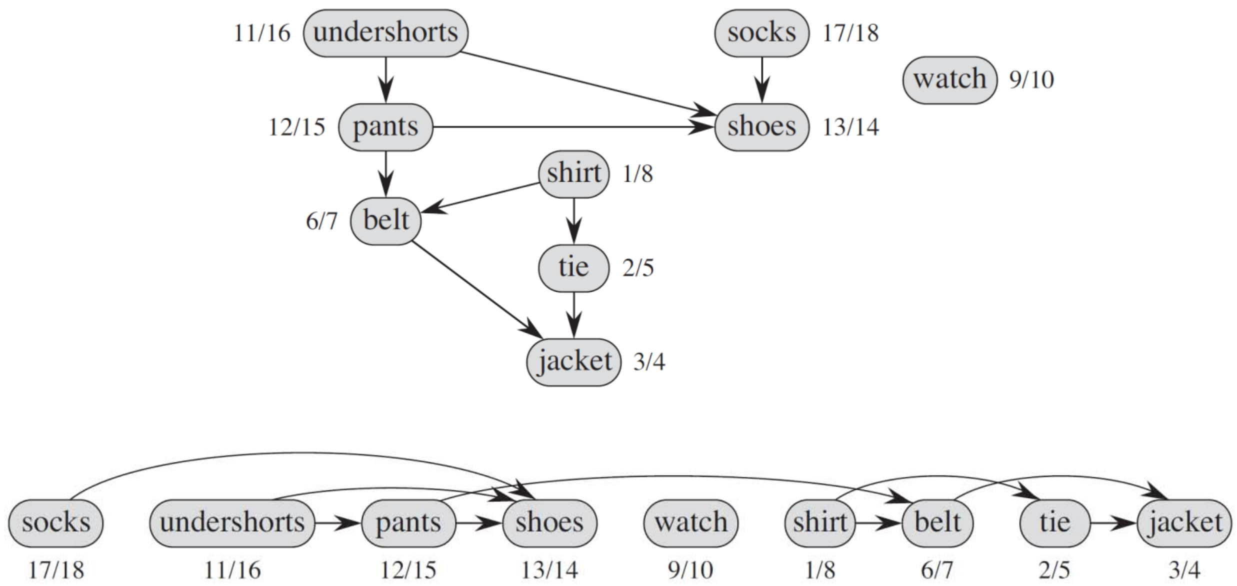
- **Theorem 22.10 :** In a depth-first search of an undirected graph *G*, every edge of *G* is either a tree edge or a back edge.
- **Proof:**
  - Suppose (u, v) is an edge in *G* with $d[u] < d[v]$.
  - Since *v* is on *u*'s adjacency list, *v* must be discovered and finished before we finish *u.*
  - If (u, v) is explored first from *u* to *v*, then (u, v) is a tree edge.
  - Otherwise, (u, v) is a back edge, since *u* is still gray at the time the edge is first explored.

Outline

- Representations of graphs
- Breadth-first search
- Depth-first search
- **Topological sort**
- Strongly connected components

Topological sort

- Use depth-first search to perform a topological sort of a directed acyclic graph (dag).
- A **topological sort** of a dag *G* is a linear ordering of all its vertices such that if *G* contains an edge (*u*, *v*), then *u* appears before *v* in the ordering.



Pseudocode

TOPOLOGICAL-SORT(G)

1. call DFS(G) to compute finishing times $f[v]$ 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

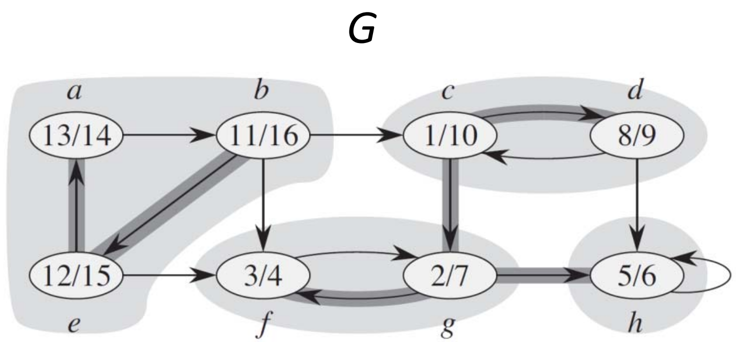
- Time: $\Theta(n+m)$.
  - Depth-first search takes $\Theta(n+m)$ time.
  - It takes $O(1)$ time to insert each of the n vertices.
- Correctness: Refer to textbook.

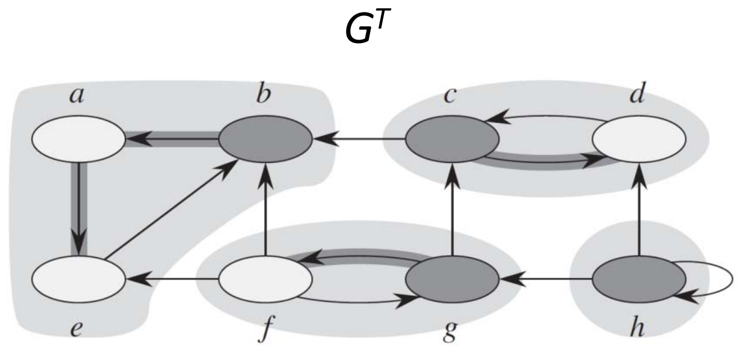
Outline

- Representations of graphs
- Breadth-first search
- Depth-first search
- Topological sort
- **Strongly connected components**

Strongly connected components

- A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \in V$ such that for every pair of vertices u and v in C, we have both u-v path and v-u path.
- The **transpose** of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(u,v) : (v,u) \in E\}$.
  - E^T consists of the edges of G with their directions reversed.
- Observe that G and G^T have exactly the same strongly connected components.



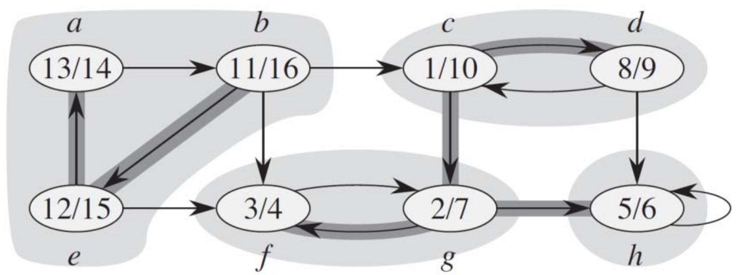


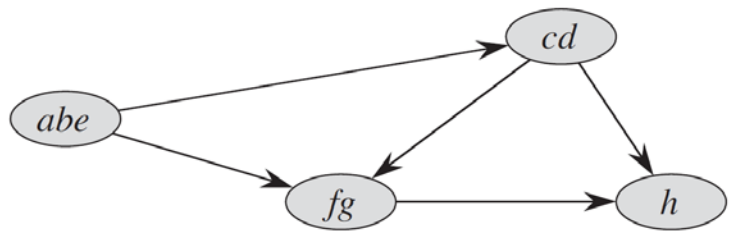
Pseudocode

STRONGLY-CONNECTED-COMPONENTS(G)

1. call DFS(G) to compute finishing times $f[u]$ 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 $f[u]$ (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

G





Complexity

- Time: $\Theta(n+m)$.
  - Two depth-first searches take $\Theta(n+m)$ time.
- Correctness: Refer to textbook.
- For an undirected graph *G*, performing DFS once can obtain all "connected components".
  - See data structures chapter 6 for more information.