CS 457, Fall 2019

Drexel University, Department of Computer Science

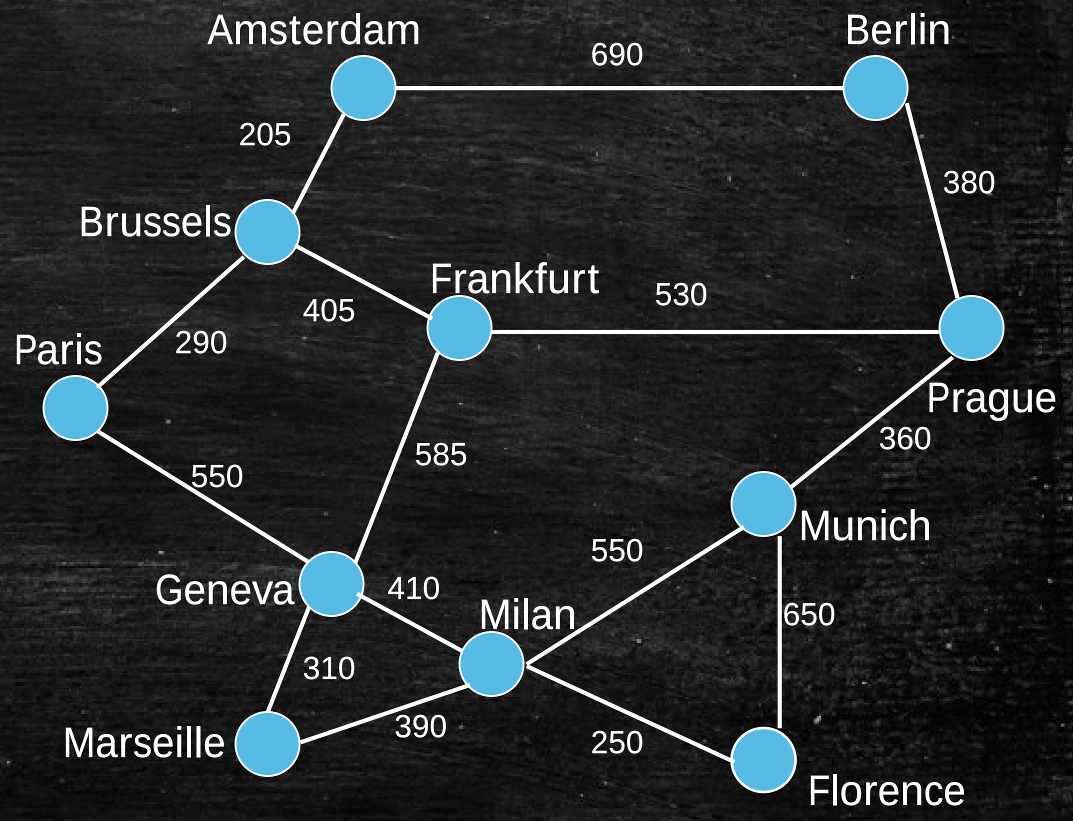
Lecture 16

Today's Lecture

- Graph algorithms
  - Minimum Spanning Tree
  - Breadth First Search
  - Depth First Search
  - Topological Sorting
  - Strongly Connected Components
- Greedy algorithms

Graphs

- Things to know:
  - **Path**
  - **Cycle**
  - **Sub-graph**
  - **Degree of a vertex**
  - **Maximum and minimum degree**
  - **Maximum number of edges**
  - **Connected components**
  - **Shortest path (weighted & unweighted)**
  - **Distance of two vertices**
  - **Tree (rooted tree)**
  - **Spanning tree of a graph**
  - **Acyclic graph**
  - **Bipartite graph**

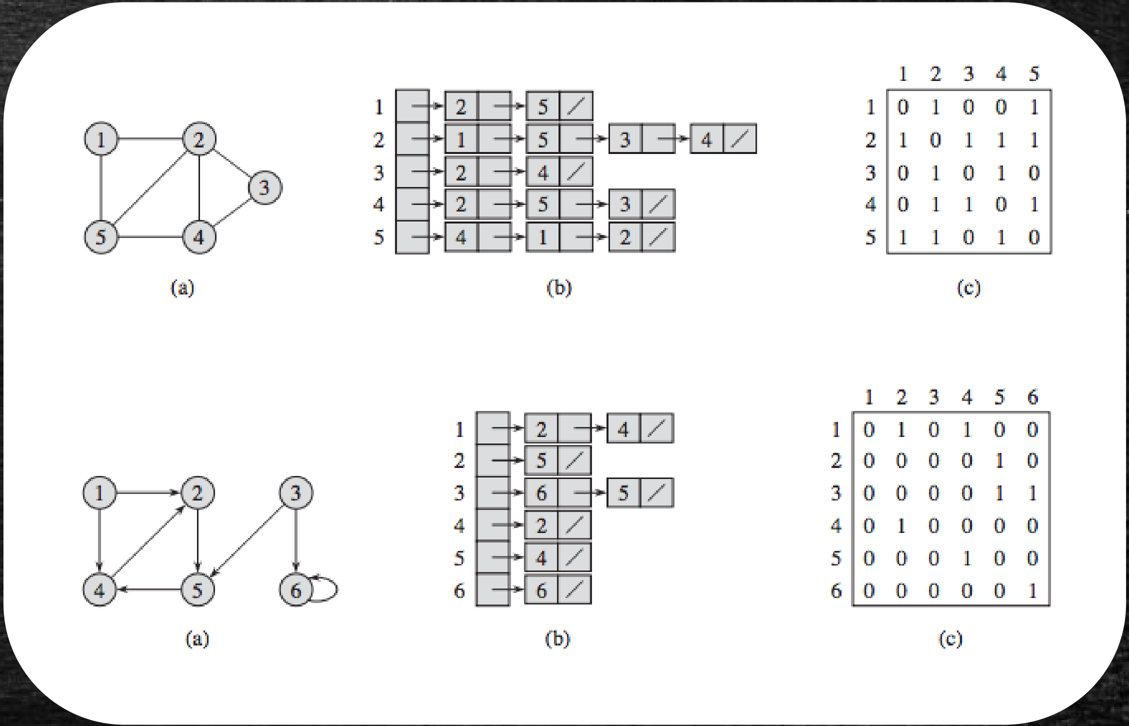


Definitions

- Given A graph $G=(V,E)$, where
  - V is its vertex set, $|V|=n$,
  - E is its edge set, with $|E|=m=O(n^2)$
- If G is connected then for every pair of vertices u,v in G, there is path connecting them
- In an undirected graph, an edge $(u, v)=(v, u)$.
- In a directed graph, (u, v) is different from (v, u).
- In a weighted graph there are weights associated with edges and/or vertices.
- Running time of graph algorithms are usually expressed in terms of n or m.

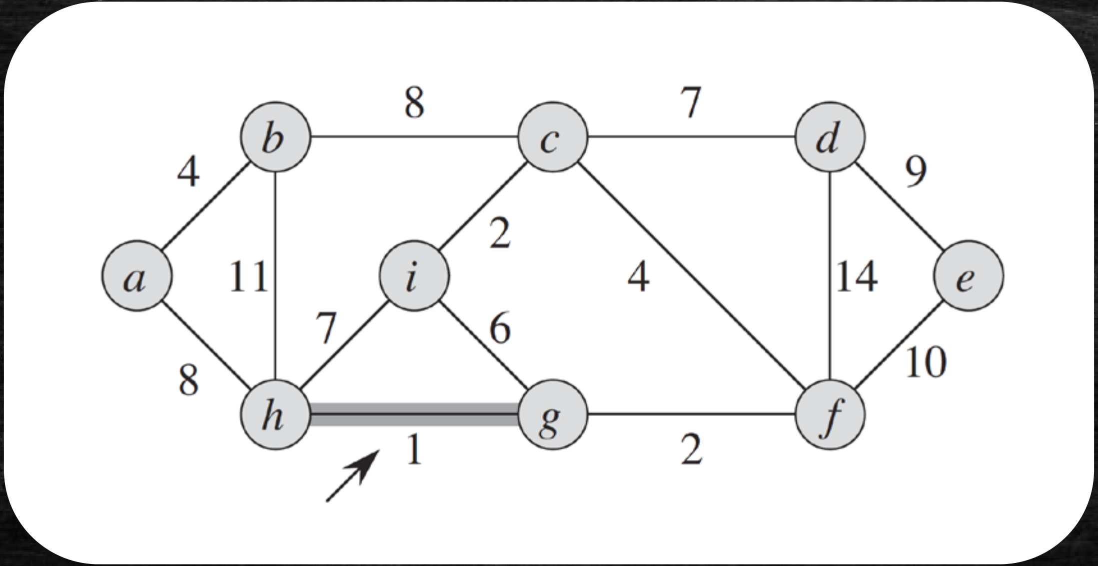
Graph Representations

- Adjacency List
  - Good for sparse graphs
- Adjacency Matrix
  - Quick edge existence query
  - Simple



Minimum Spanning Tree of a Weighted Graph

- Let G=(V,E) be a graph on n vertices, m edges, and a weight w on edges in E.
- Sub-graph T=(V,E') with $E' \subseteq E$ with no cycles is a spanning tree
- The weight of T is the sum of the weights of its edges: $w(T) = \sum_{(u,v)\in E'} w(u,v)$

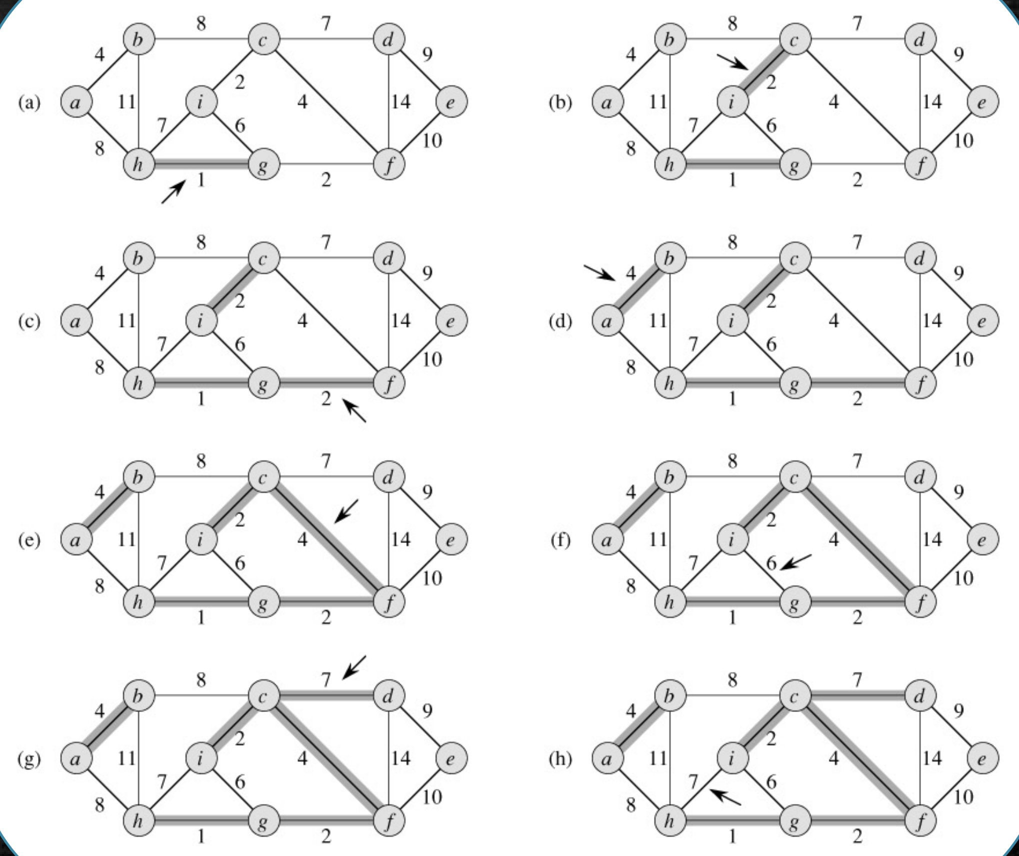


Set Operations

- We will use the following set operations:
  - Make-Set(v): creates a set containing element v, i.e., $\{v\}$
  - Find-Set(u): returns the set to which v belongs to
  - Union(u, v): creates a set which is the union of two sets, the one containing v and the one containing u

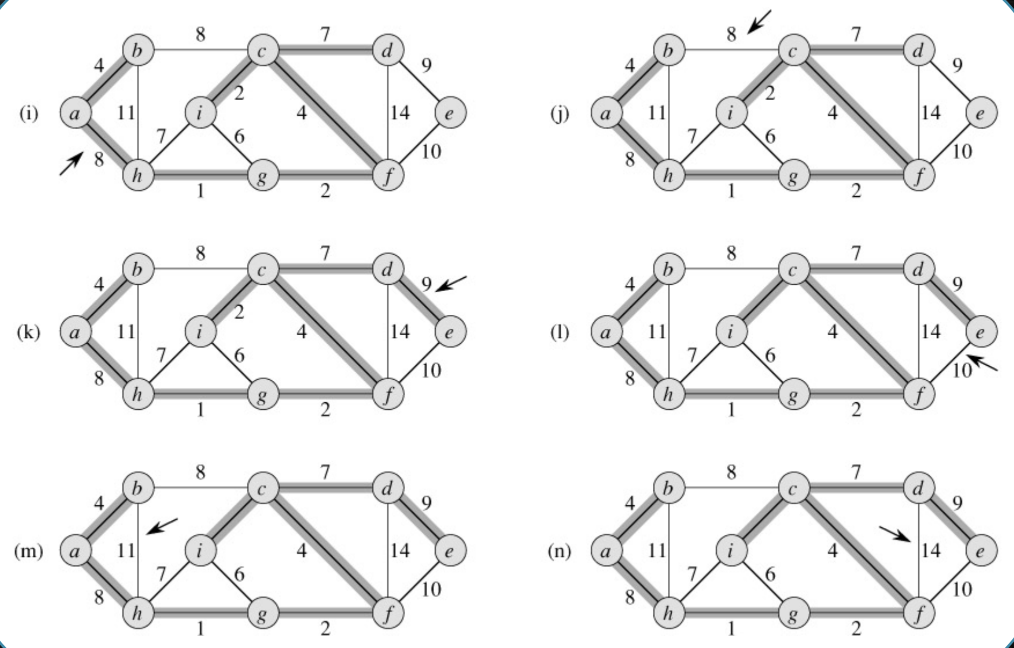
Kruskal’s Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
      MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
      if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
return A
```



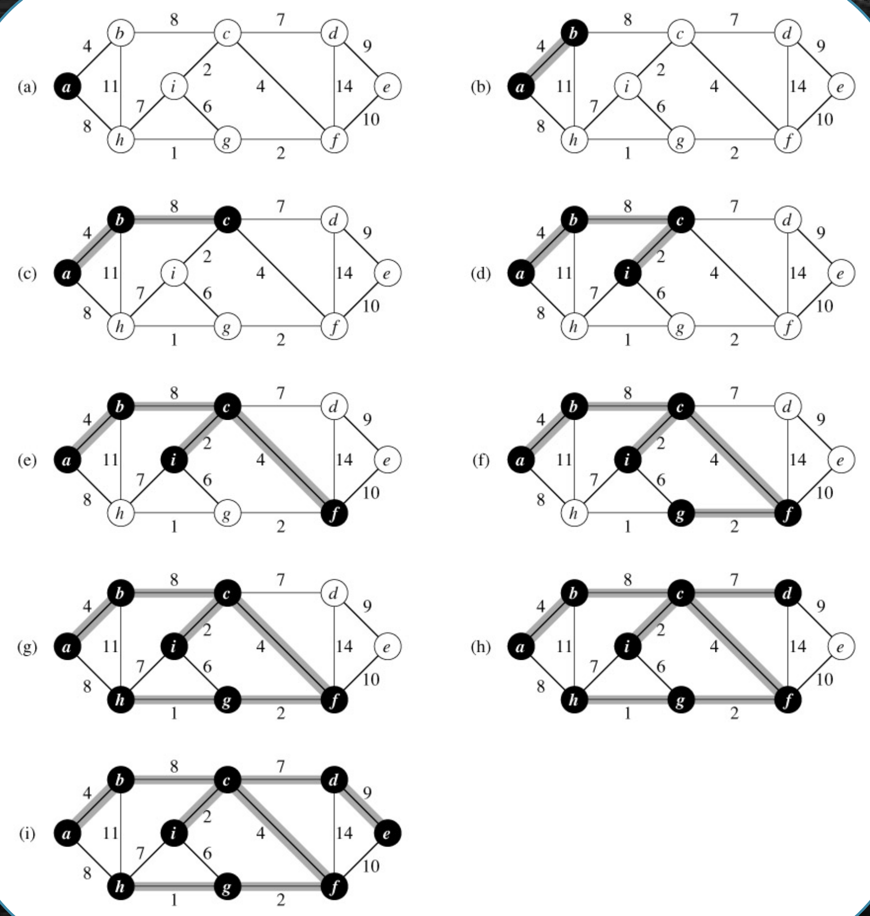
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```



Prim’s Algorithm

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



Practice Problem

- Is the path between two vertices in an MST necessarily a shortest path between the two vertices in the full graph? Give a proof or a counterexample
  - Answer: No it is not. For example, consider a graph that forms a single n-vertex cycle. The minimum spanning tree will remove just one edge (u, v), significantly increasing the distance between u and v

Breadth First Search (BFS)

- Given a graph $G = (V,E)$, BFS starts at some source vertex s and discovers which vertices are reachable from s
- The distance between a vertex v and s is the minimum number of edges on a path from s to v (the shortest path length)
- BFS discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths from s
- At any given time there is a frontier of vertices that have been discovered, but not yet processed.
- BFS first visits all vertices across the breadth of this frontier (hence the name)

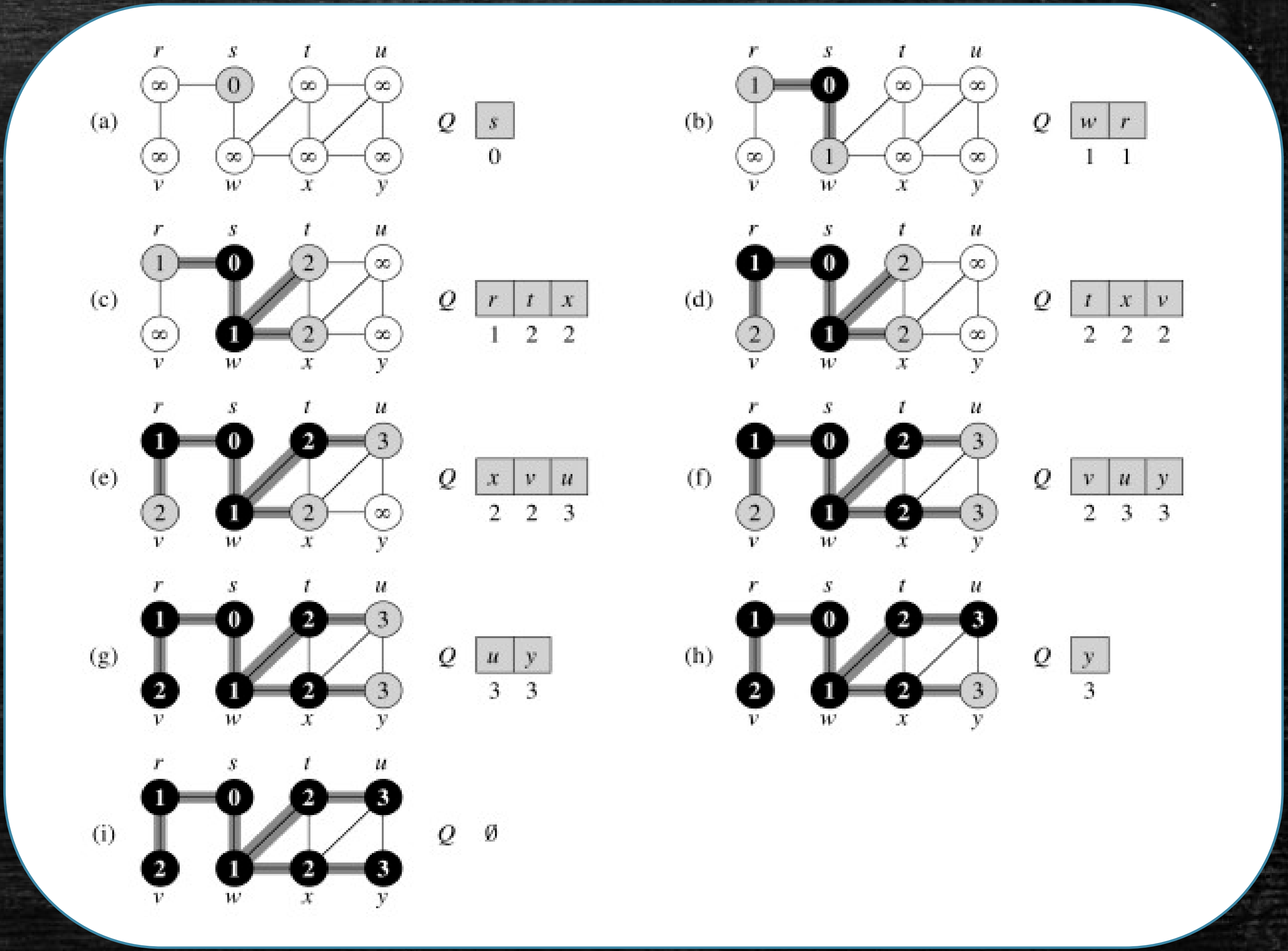
BFS: coloring

- We will use the following coloring procedure to show the status of BFS at each instance of time:
  - Initially all vertices (except the source) are colored white, meaning that they are undiscovered
  - When a vertex has first been discovered, it is colored gray (and is part of the frontier)
  - When a gray vertex is processed, it becomes black

BFS Algorithm

BFS (G, s)

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



BFS predecessor subgraph of G

For a graph $G = (V, E)$ with source s the predecessor subgraph of G is:

- $G_\pi = (V_\pi, E_\pi)$, where
- $V_\pi = \{v \in V : v.\pi \neq \mathrm{NIL}\} \cup \{s\}$
- $E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$

DFS Algorithm

DFS (G)

```
for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
time = 0
for each vertex u ∈ G.V
        if u.color = WHITE
                DFS-Visit(G,u)
```

DFS-Visit(G,u)

```
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
        if v.color == WHITE
            v.π = u
            DFS-Visit(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

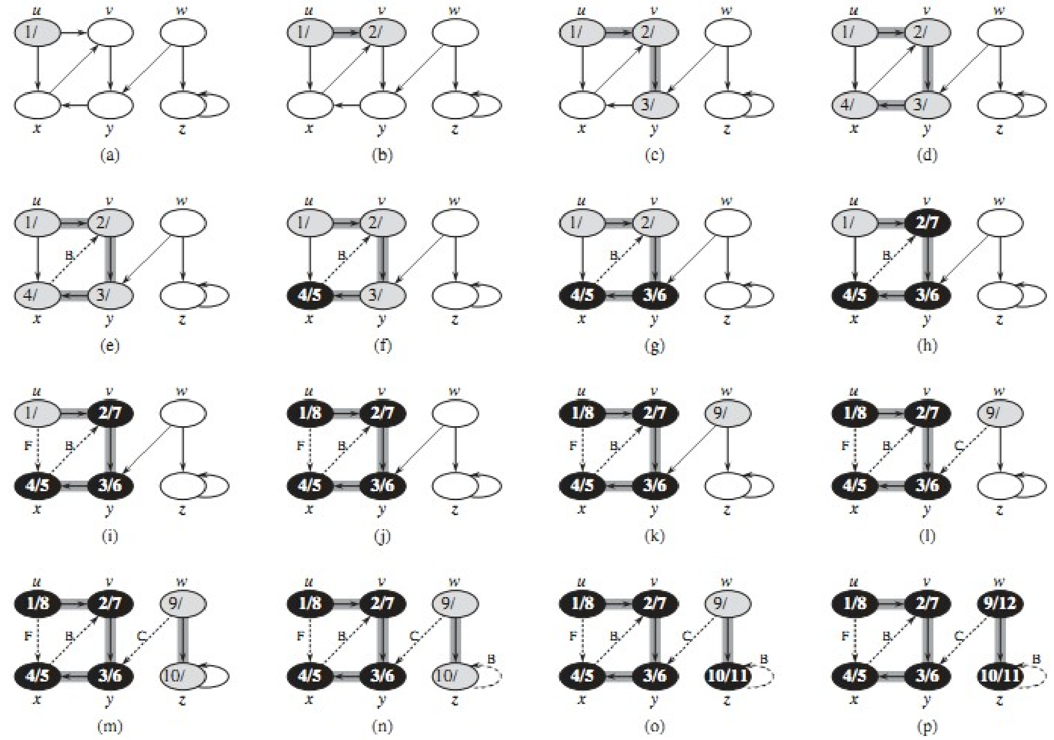
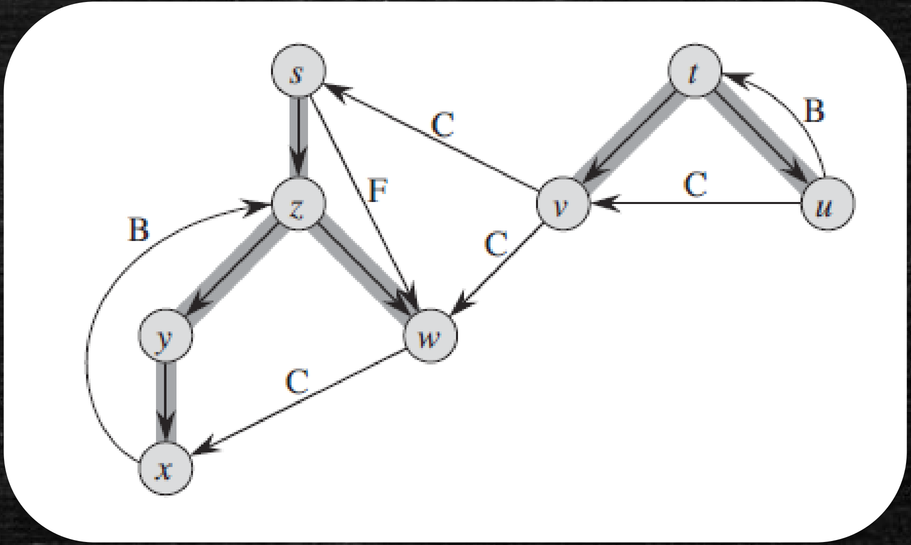


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

Depth-First Forest Edges

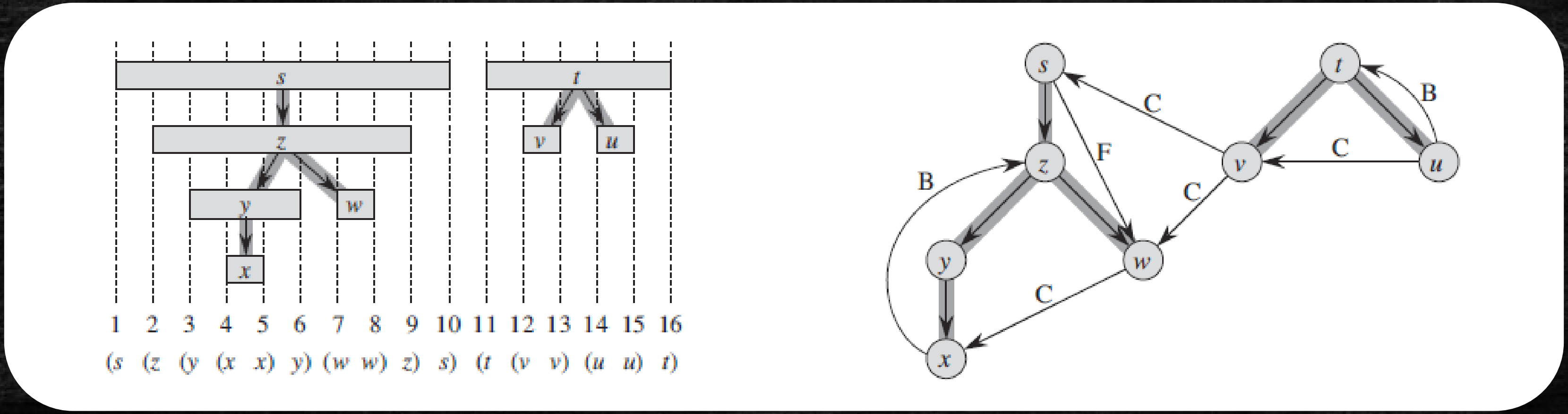
Classification of edges based on depth-first forest:

- Tree Edges: Edges in the depth-first forest G_π
- Back Edges: connecting a vertex to an ancestor in the depth-first tree
- Forward Edges: connecting a vertex to a descendant in the depth-first tree
- Cross edges: all other edges



Cycles in a Graph

- Time stamps of DFS help determine if a graph $G = (V, E)$ contains any cycles
- Consider any DFS forest of G, and consider any edge (u, v) in E:
  - If this edge is a tree, forward, or cross edge, then $u.f > v.f$
  - If the edge is a back edge then $u.f < v.f$
- G has a cycle if and only the DFS forest has a back edge
- Checking if G is acyclic reduces to checking if it has a back edge

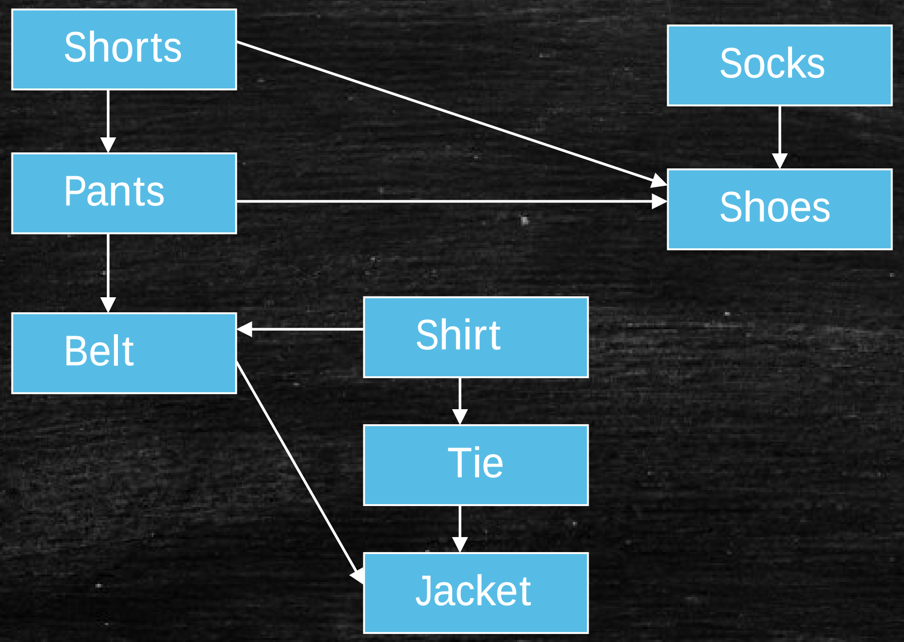


Directed Acyclic Graph

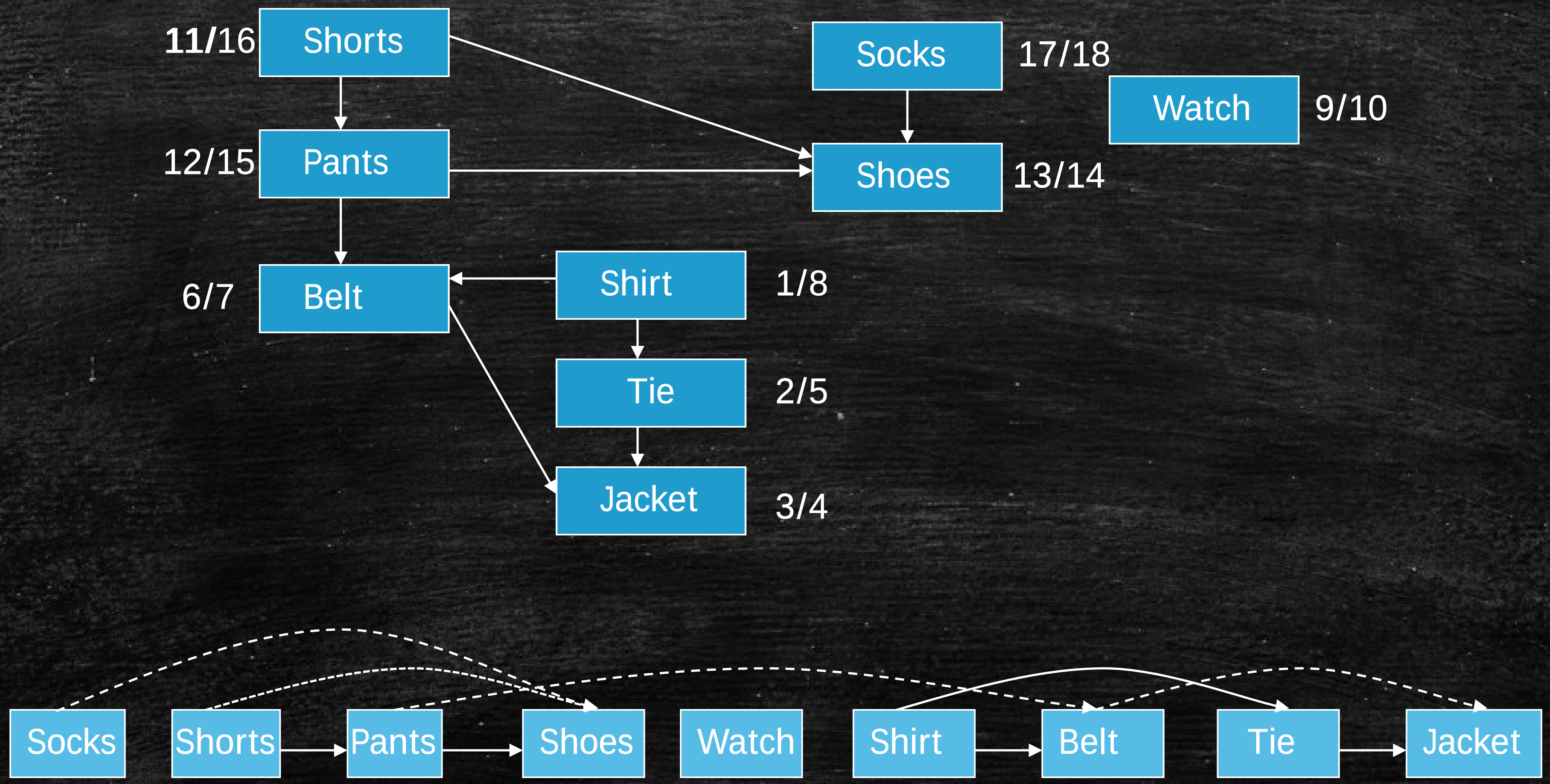
- A directed acyclic graph is often called a DAG for short.
- DAG's arise in many applications with precedence / ordering constraints
- In general a precedence constraint graph is a DAG in which
  - vertices are tasks and
  - the edge (u, v) means that task u must be completed before task v begins.

Topological Sort

- A topological sort of a DAG is a linear ordering of the vertices of the DAG such that for each edge (u, v), u appears before v in the ordering.



Example



11/16
Shorts
12/15
Pants
6/7
Belt
Socks
17/18
Shoes
13/14
Watch
9/10
Shirt
1/8
Tie
2/5
Jacket
3/4
Socks
Shorts
Pants
Shoes
Watch
Shirt
Belt
Tie
Jacket

Topological Sort Algorithm

TopologicalSort(G)

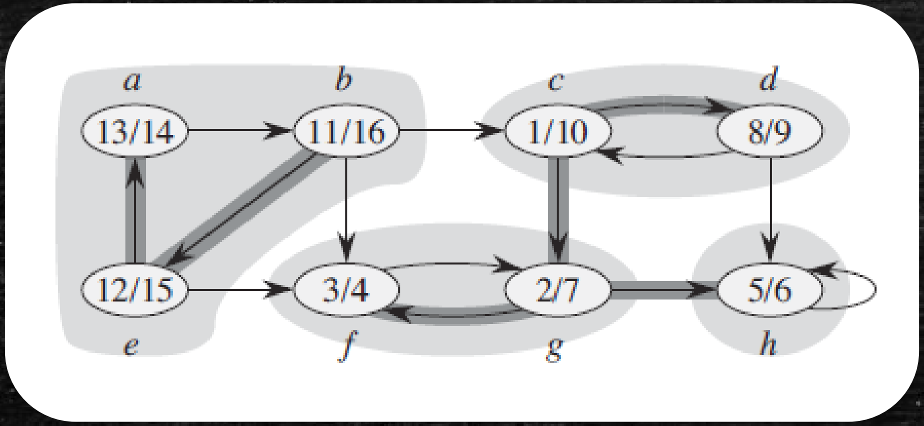
1. call DFS(G) to compute finishing times $v.f$ for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

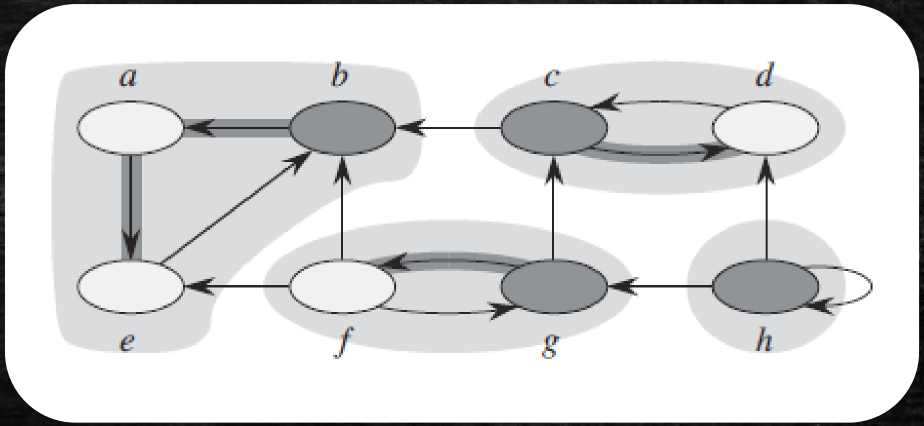
- Running time
- DFS: $\Theta(n+m)$
- Insertion to linked list: $\Theta(n)$

Strongly Connected Components

Strongly Connected Componenets(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, use decreasing order w.r.t. $v.f$
4. return the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

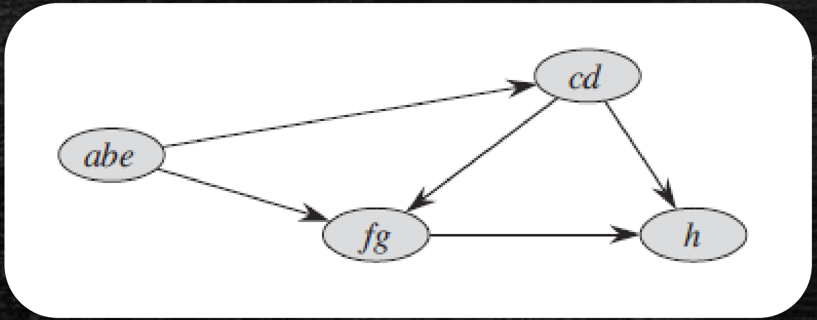


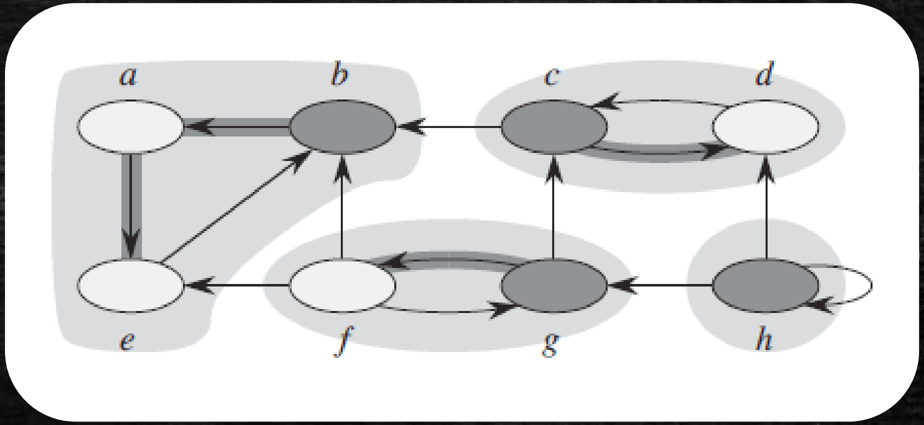


Strongly Connected Components

Strongly Connected Componenets(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, use decreasing order w.r.t. $v.f$
4. return the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component





Practice Problems

- A root of a DAG is a vertex r such that every other vertex of the DAG can be reached from r using a directed path. Give an algorithm that determines whether a given DAG has a root.

- Provide an algorithm that, given a directed acyclic graph $G = (V, E)$ and two vertices $s, t \in V$, returns the number of simple paths from s to t in G.