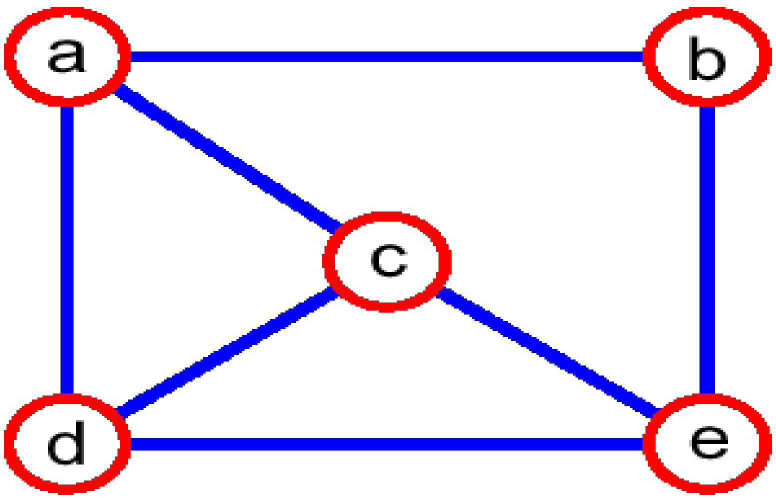
Next: Graph Algorithms

- Graphs Ch 22
- Graph representations
  - adjacency list
  - adjacency matrix
- Minimum Spanning Trees Ch 23
- Traversing graphs
  - Breadth-First Search
  - Depth-First Search

Graphs – Definition

- A graph G = (V,E) is composed of:
  - V: set of **vertices**
  - $E \subset V \times V$: set of **edges** connecting the **vertices**
- An edge e = (u,v) is a pair of vertices
- (u,v) is ordered, if G is a directed graph

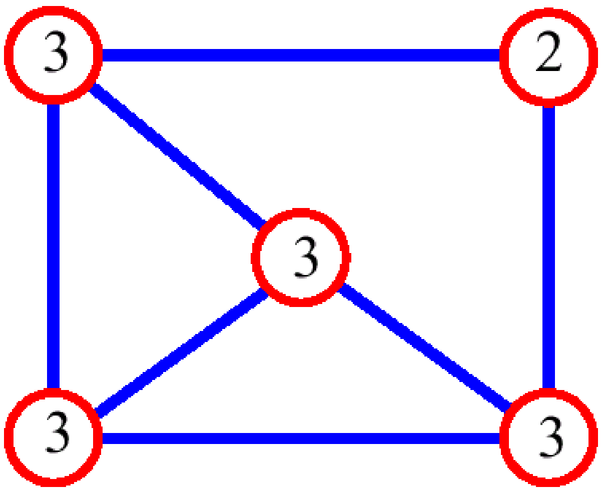


V= {a,b,c,d,e}

E=
{(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e),
(d,e)}

Graph Terminology

- adjacent vertices: connected by an edge
- degree (of a **vertex**): # of adjacent vertices



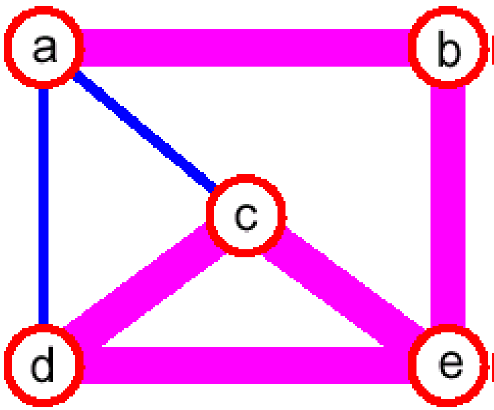
$$\sum_{v \in V} \deg(v) = 2(\#\ \text{of edges})$$

Since adjacent vertices each count the adjoining edge, it will be counted twice

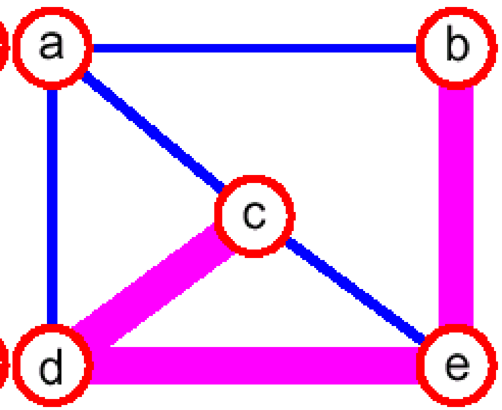
- path: sequence of vertices $v_1, v_2, \ldots, v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent

Graph Terminology (2)

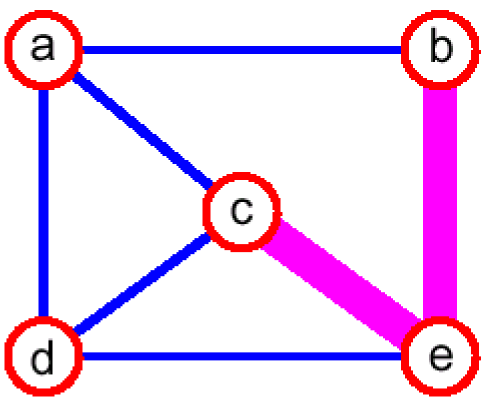
- simple path: no repeated vertices



a b e d c



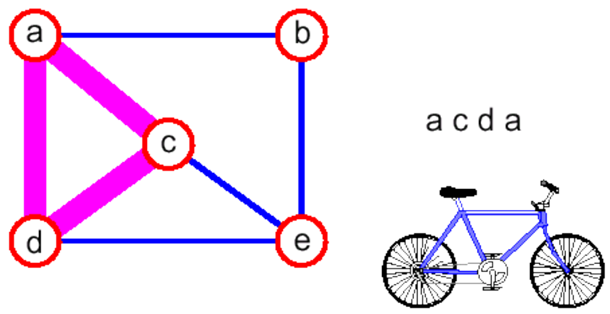
b e d c



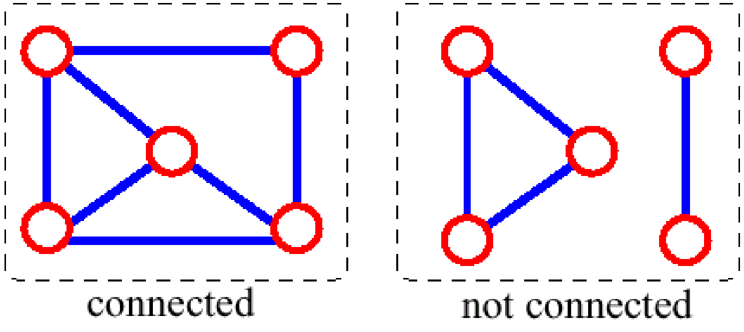
b e c

Graph Terminology (3)

- cycle: simple path, except that the last vertex is the same as the first vertex

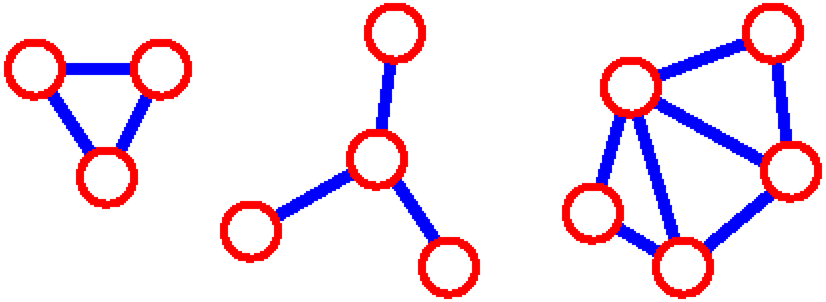


- connected graph: any two vertices are connected by some path



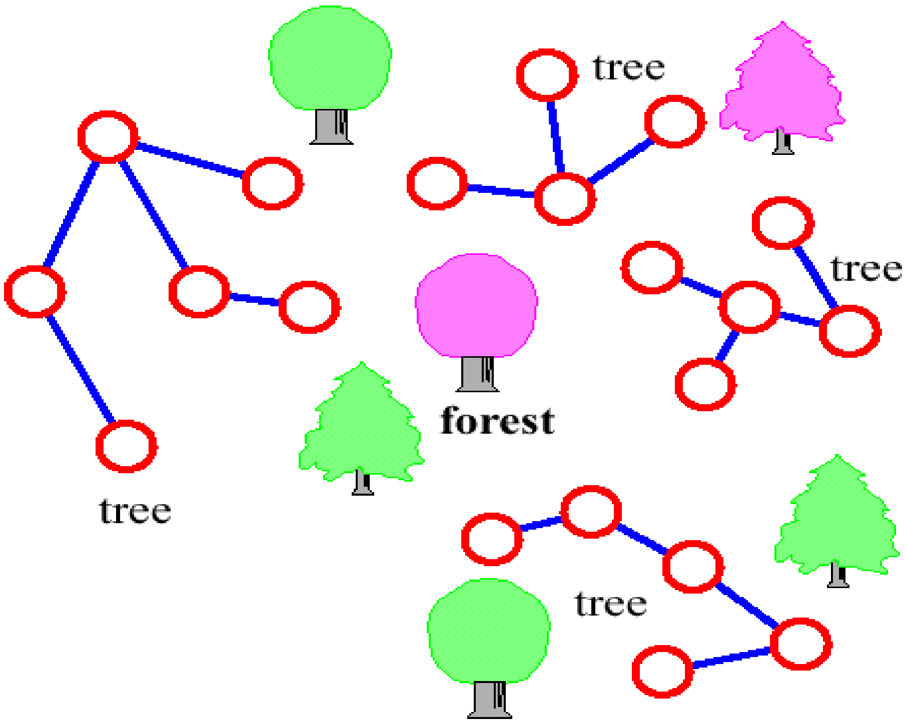
Graph Terminology (4)

- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components



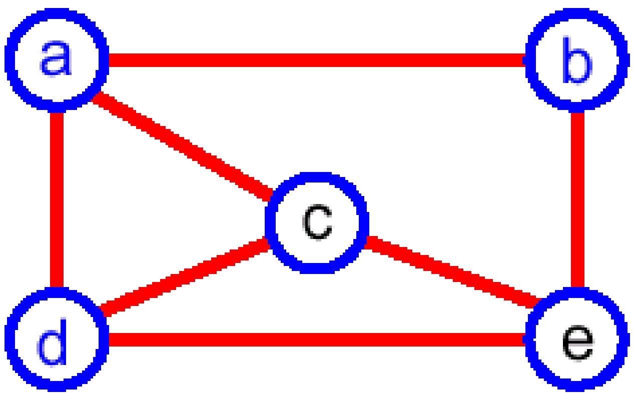
Graph Terminology (5)

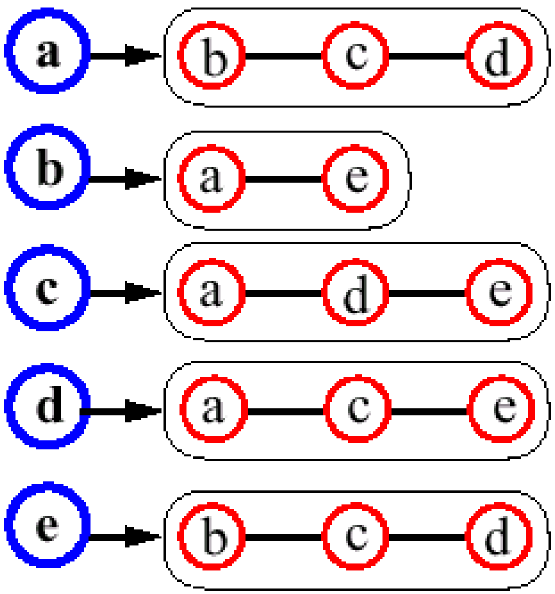
- (free) tree - connected graph without cycles
- forest - collection of trees



Data Structures for Graphs

- The **Adjacency list** of a vertex v: a sequence of vertices adjacent to v
- Represent the graph by the adjacency lists of all its vertices

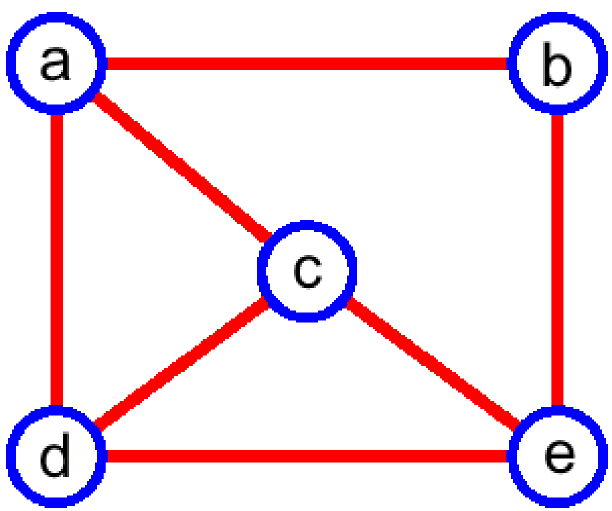




$$\text{Space } = \Theta(n + \sum \deg(v)) = \Theta(n + m)$$

Data Structures for Graphs

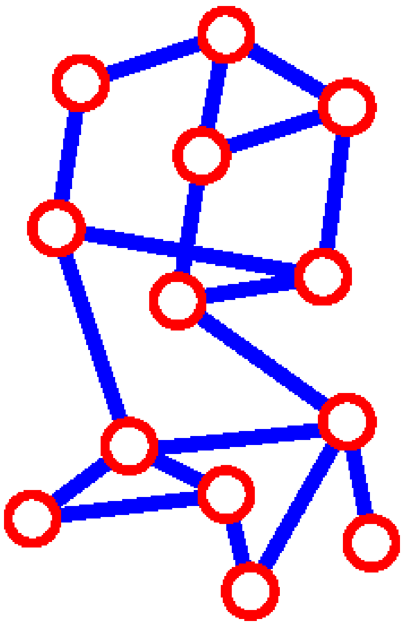
- Adjacency matrix
- Matrix M with entries for all pairs of vertices
- M[i,j] = true – there is an edge (i,j) in the graph
- M[i,j] = false – there is no edge (i,j) in the graph
- Space = $O(\boldsymbol{n}^2)$



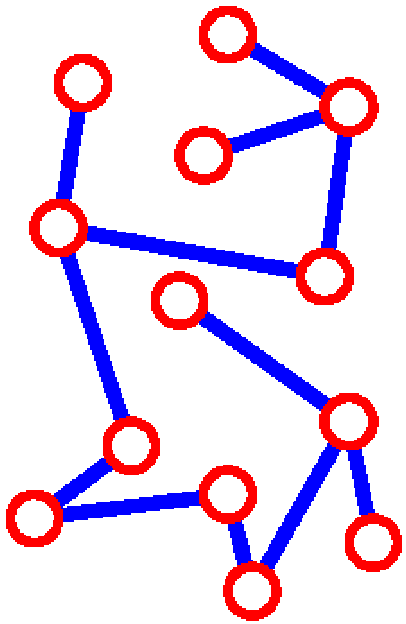
	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

Spanning Tree

- A **spanning tree** of **G** is a subgraph which
  - is a tree
  - contains all vertices of **G**



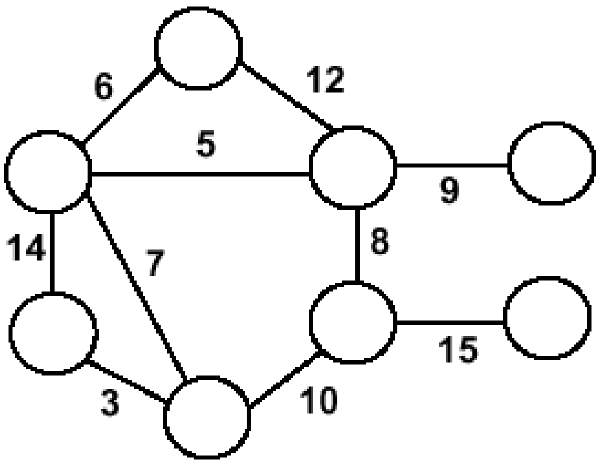
G



spanning tree of **G**

Minimum Spanning Trees

- Undirected, connected graph $G = (V,E)$
- Weight function $W: E \rightarrow R$ (assigning cost or length or other values to edges)

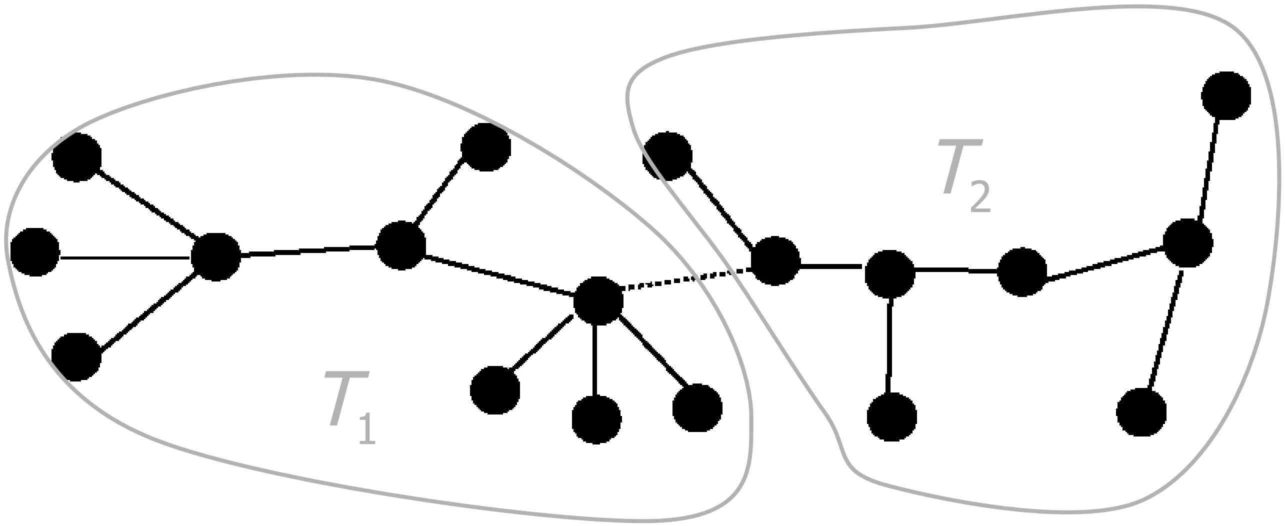


- Spanning tree: tree that connects all vertices
- Minimum spanning tree: tree that connects all the vertices and minimizes

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

Optimal Substructure

- MST T



- Removing the edge (u,v) partitions T into T_1 and T_2

$$w(T) = w(u,v) + w(T_1) + w(T_2)$$

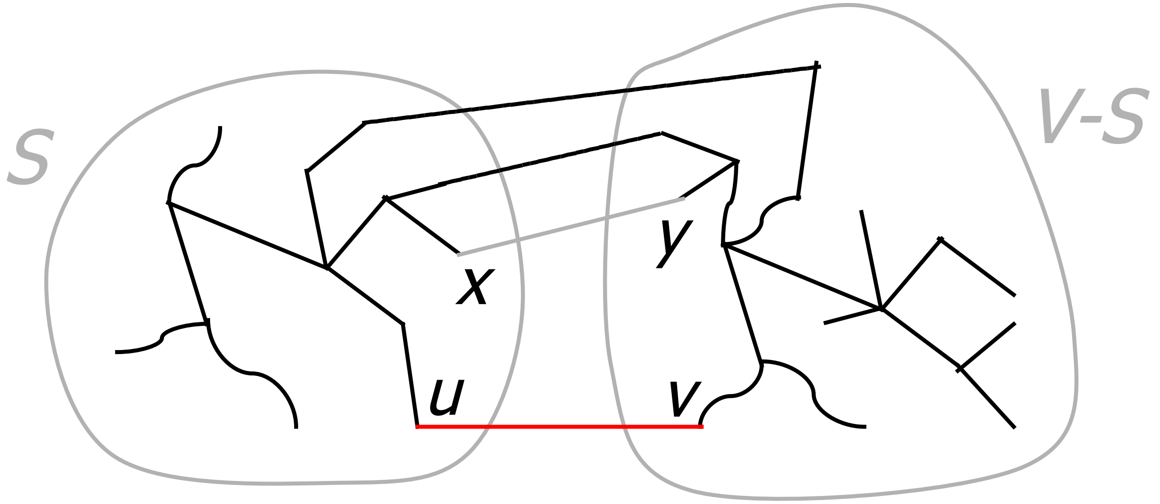
- We claim that T_1 is the MST of $G_1=(V_1,E_1)$, the subgraph of G induced by vertices in T_1
- Also, T_2 is the MST of G_2

Greedy Choice

- Greedy choice property: locally optimal (greedy) choice yields a globally optimal solution
- Theorem
  - Let $G=(V, E)$, and let $S \subseteq V$ and
  - let (u,v) be min-weight edge in G connecting S to $V - S$
  - Then $(u,v) \in T$ – some MST of G

Greedy Choice (2)

- Proof
  - suppose $(u,v) \notin T$
  - look at path from *u* to *v* in *T*
  - swap (*x*, *y*) – the first edge on path from *u* to *v* in *T* that crosses from *S* to *V* – *S*
  - this improves *T* – contradiction (*T* supposed to be MST)



Generic MST Algorithm

```
Generic-MST(G, w)
A←∅  // Contains edges that belong to a MST
while A does not form a spanning tree do
   Find an edge (u,v) that is safe for A
   A←A∪{(u,v)}
return A
```

Safe edge – edge that does not destroy *A*'s property

```
MoreSpecific-MST(G, w)
 A←∅  // Contains edges that belong to a MST
 while A does not form a spanning tree do
  Make a cut (S, V-S) of G that respects A
  Take the min-weight edge (u,v) connecting S to V-S

   A←A∪{(u,v)}
return A
```

Prim's Algorithm

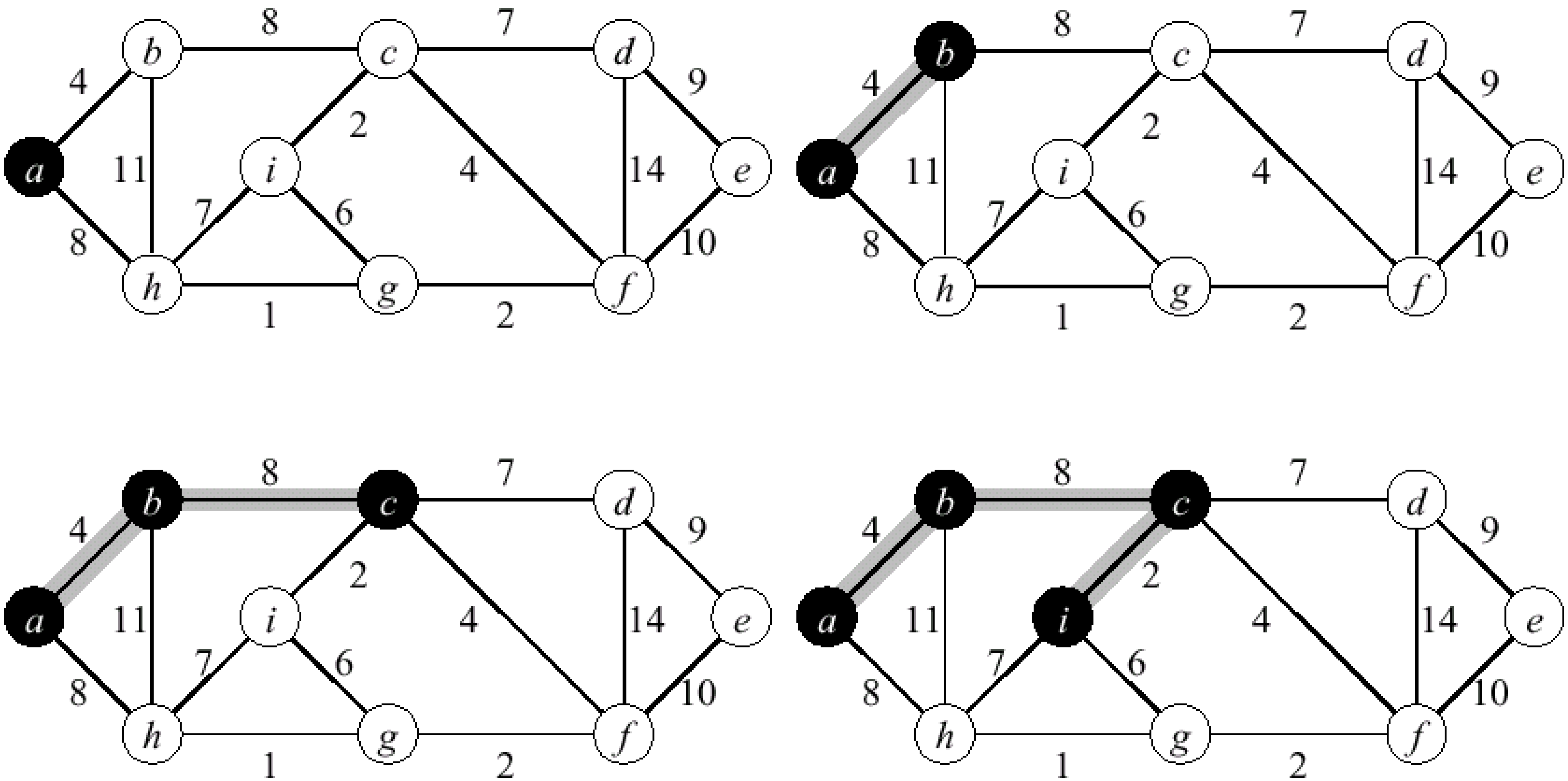
- Vertex based algorithm
- Grows one tree T, **one vertex at a time**
- A cloud covering the portion of T already computed
- Label the vertices v outside the cloud with $key[v]$ – the minimum weight of an edge connecting v to a vertex in the cloud, $key[v] = \infty$, if no such edge exists

Prim's Algorithm (2)

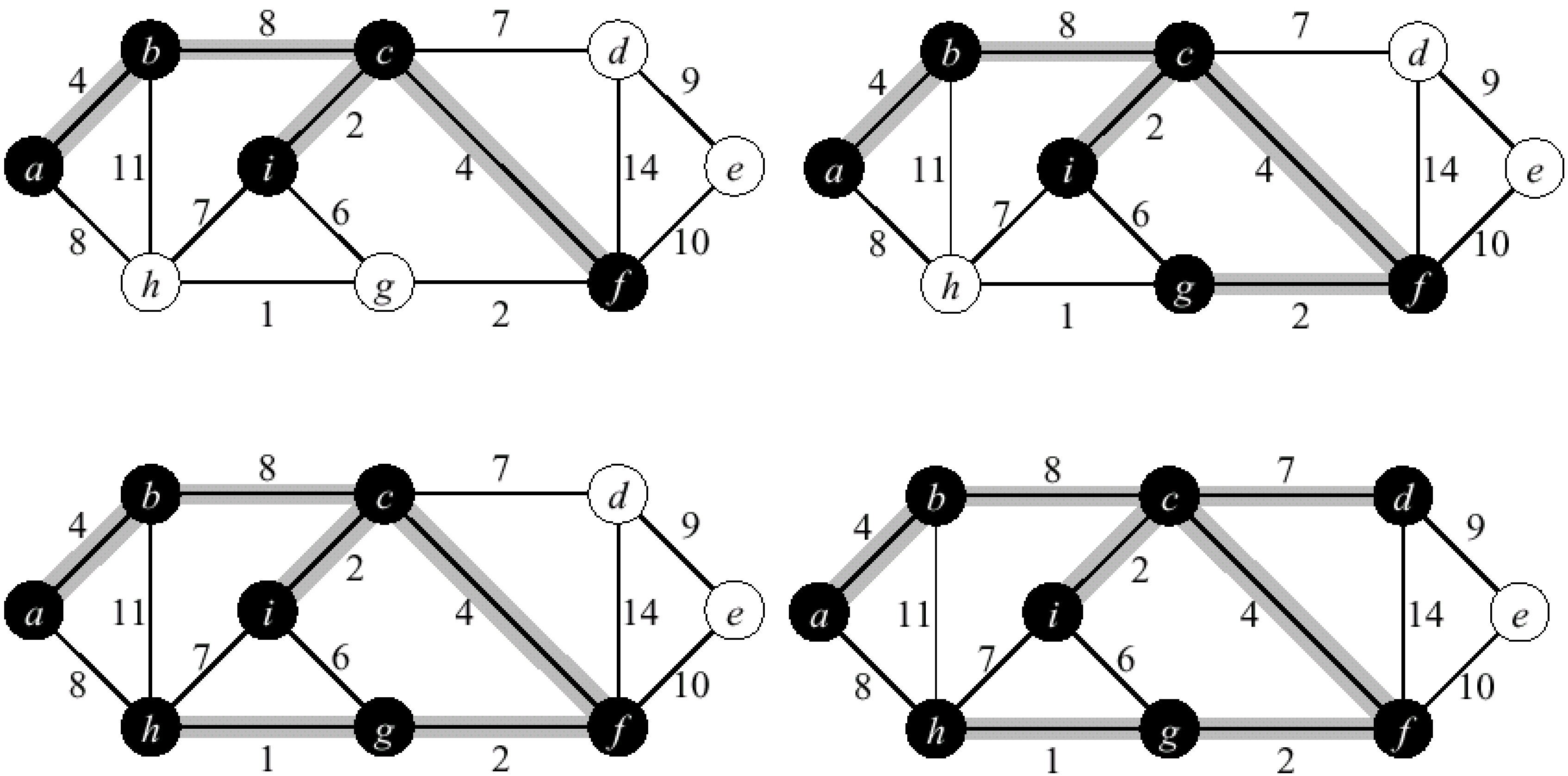
```
MST-Prim(G,w,r)
Q ← V[G]  // Q – vertices out of T
for each u ∈ Q
   key[u] ← ∞
key[r] ← 0
π[r] ← NIL
while Q ≠ ∅ do
  u ← ExtractMin(Q)  // making u part of T
     for each v ∈ Adj[u] do
        if v ∈ Q and w(u,v) < key[v] then
           π[v] ← u
           key[v] ← w(u,v)
```

updating keys

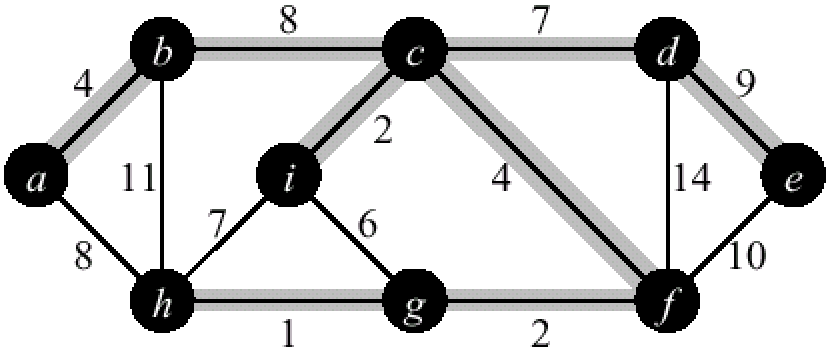
Prim Example



Prim Example (2)



Prim Example (3)



Priority Queues

- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called key
- We need PQ to support the following operations
  - BuildPQ(S) – initializes PQ to contain elements of S
  - ExtractMin(S) returns and removes the element of S with the smallest key
  - ModifyKey(S,x,newkey) – changes the key of x in S
- A min heap can be used to implement a PQ
  - BuildPQ – $O(n)$
  - ExtractMin and ModifyKey – $O(\lg n)$

Prim's Running Time

- Time = $|V|T(\text{ExtractMin}) + O(|E|)T(\text{ModifyKey})$
- Time = $O(|V| \lg|V| + |E| \lg|V|) = O(|E| \lg|V|)$

Q	T(ExtractMin)	T(DecreaseKey)	Total
array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
min heap	$O(\lg \|V\|)$	$O(\lg \|V\|)$	$O(\|E\| \lg\|V\|)$
Fibonacci heap	$O(\lg \|V\|)$	$O(1)$ amortized	$O(\|V\| \lg\|V\| + \|E\|)$

Kruskal's Algorithm

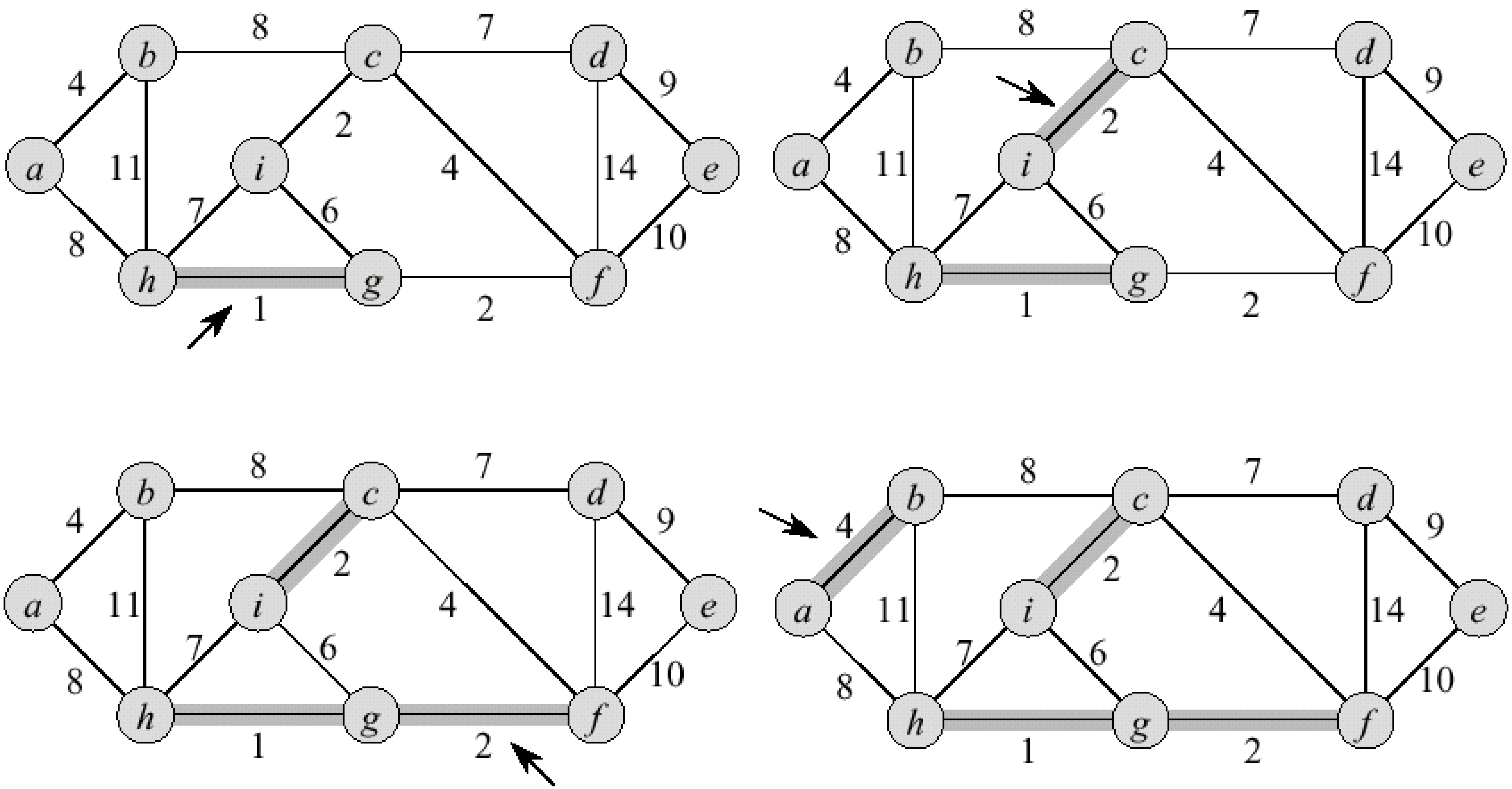
- Edge based algorithm
- Add the edges one at a time, in increasing weight order
- The algorithm maintains *A* – a **forest of trees**. An edge is accepted it if connects vertices of distinct trees
- We need an ADT that maintains a partition, i.e.,a collection of disjoint sets
  - MakeSet(S,x): $S \leftarrow S \cup \{\{x\}\}$
  - Union(S_i,S_j): $S \leftarrow S - \{S_i,S_j\} \cup \{S_i \cup S_j\}$
  - FindSet(S, x): returns unique $S_i \in S$, where $x \in S_i$

Kruskal's Algorithm

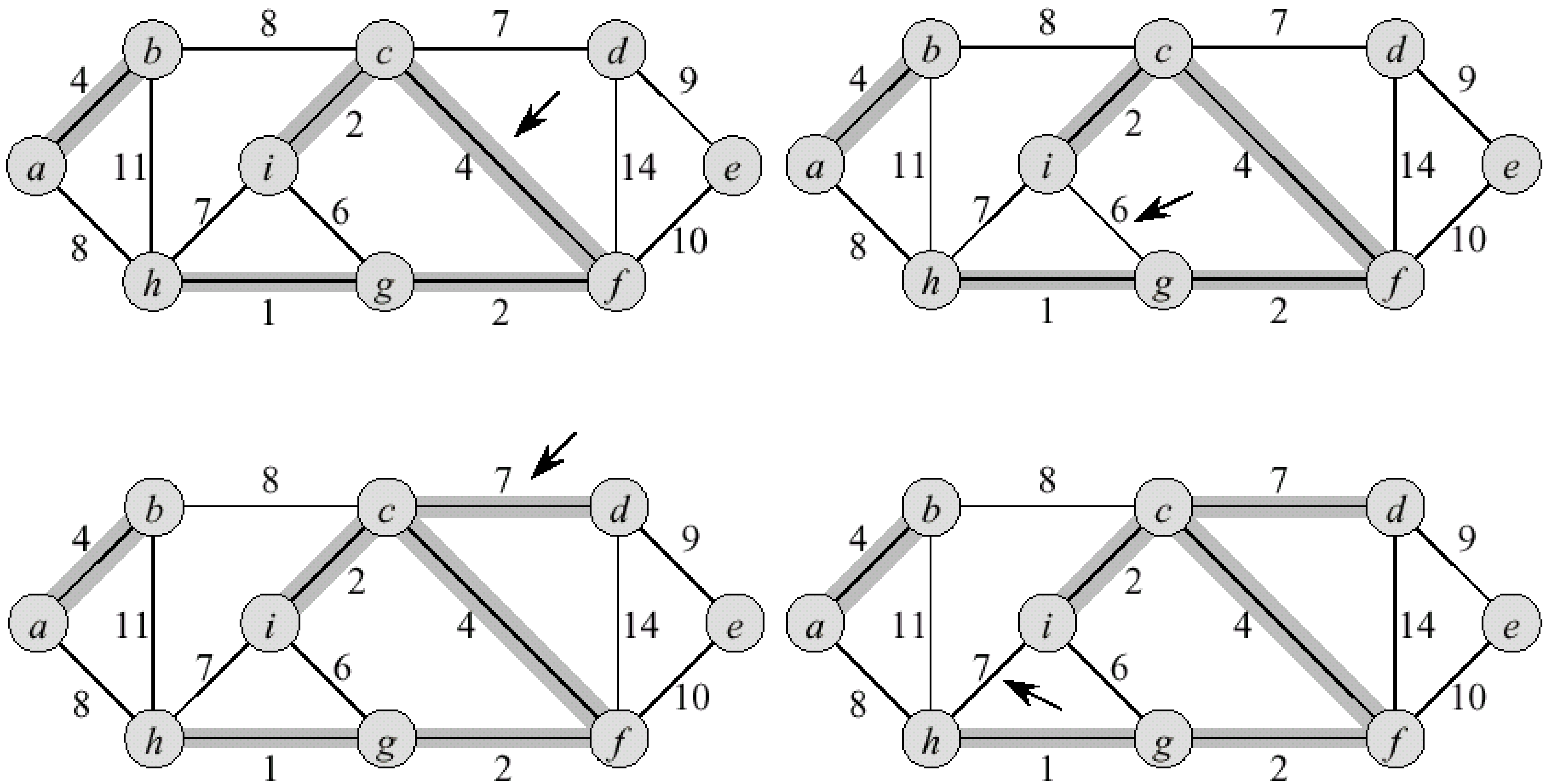
- The algorithm keeps adding the cheapest edge that connects two trees of the forest

```
MST-Kruskal(G,w)
A ← ∅
for each vertex v ∈ V[G] do
    Make-Set(v)
sort the edges of E by non-decreasing weight w
for each edge (u,v)∈ E, in order by non-decreasing weight do
   if Find-Set(u) ≠ Find-Set(v) then
      A ← A ∪ {(u,v)}
      Union(u,v)
return A
```

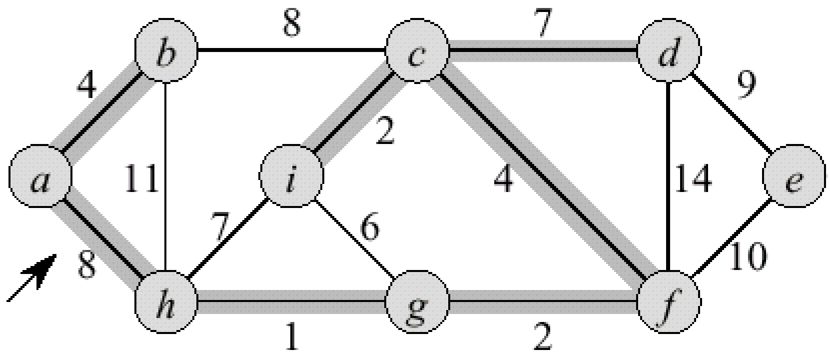
Kruskal's Algorithm: example

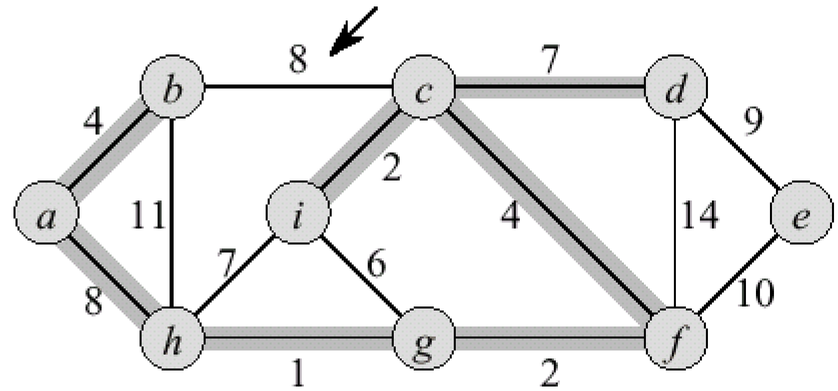


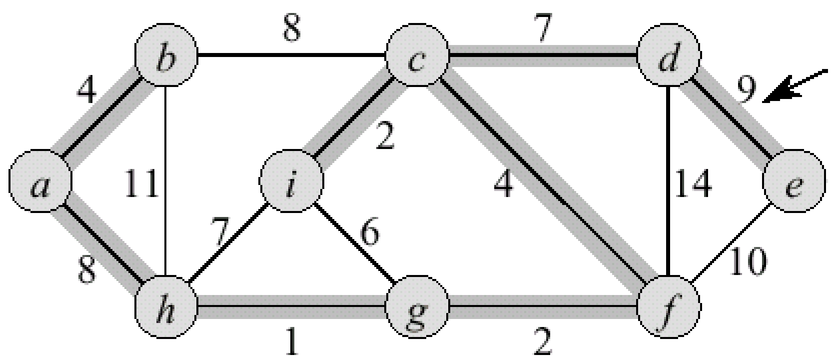
Kruskal's Algorithm: example (2)

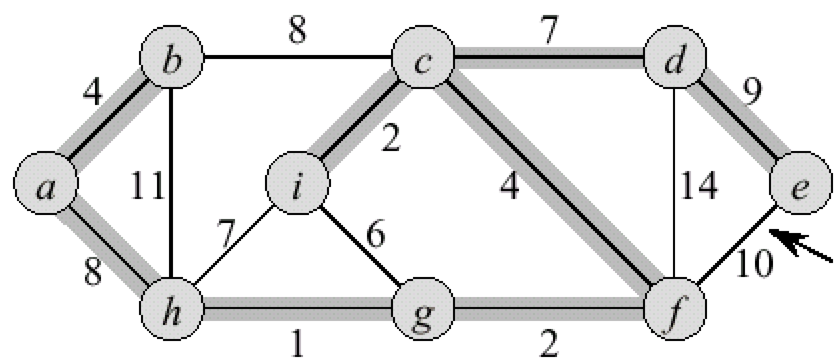


Kruskal's Algorithm: example (3)

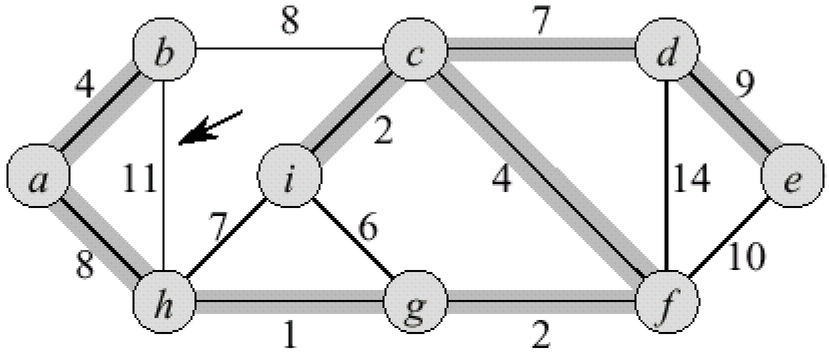


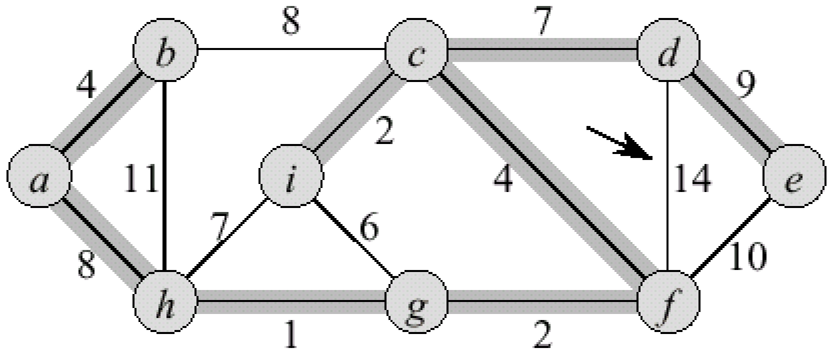






Kruskal's Algorithm: example (4)





Kruskal running time

- Initialization $O(|V|)$ time
- Sorting the edges $\Theta(|E| \lg |E|) = \Theta(|E| \lg |V|)$ (why?)
- $O(|E|)$ calls to FindSet
- Union costs
  - Let $t(v)$ – the number of times v is moved to a new cluster
  - Each time a vertex is moved to a new cluster the size of the cluster containing the vertex at least doubles: $t(v) \leq \log |V|$
  - Total time spent doing Union $\sum_{v \in V} t(v) \leq |V| \log |V|$
- Total time: $O(|E| \lg |V|)$

Next: Graph Algorithms

- Graphs
- Graph representations
  - adjacency list
  - adjacency matrix
- Traversing graphs
  - Breadth-First Search
  - Depth-First Search

Graph Searching Algorithms

- Systematic search of every edge and vertex of the graph
- Graph G = (V,E) is either directed or undirected
- Today's algorithms assume an adjacency list representation
- Applications
  - Compilers
  - Graphics
  - Maze-solving
  - Mapping
  - Networks: routing, searching, clustering, etc.

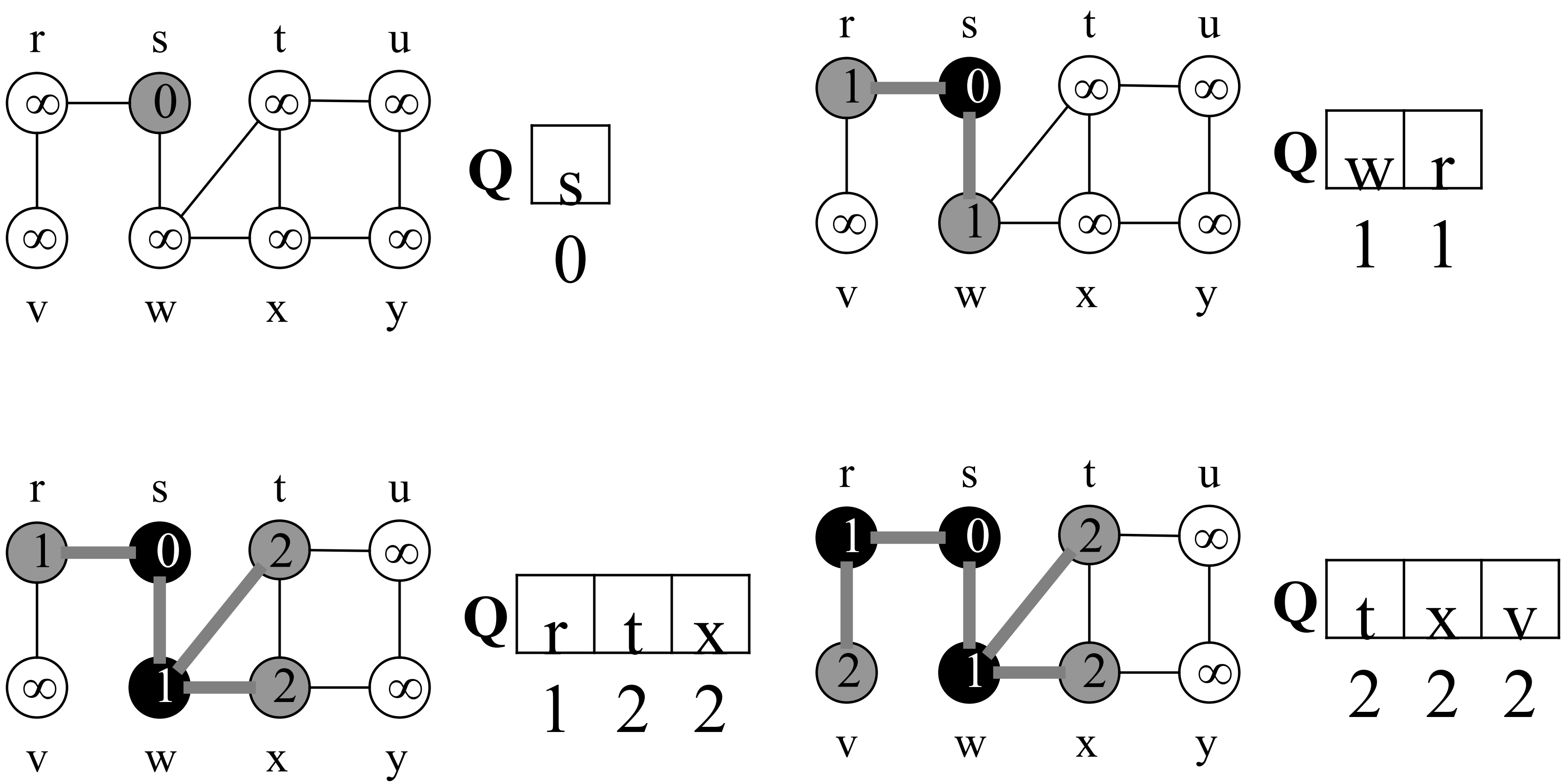
Breadth First Search

- A **Breadth-First Search (BFS)** traverses a **connected component** of a graph, and in doing so defines a **spanning tree** with several useful properties
- BFS in an **undirected** graph G is like wandering in a labyrinth with a string.
- The starting vertex *s*, it is assigned a distance 0.
- In the first round, the string is unrolled the length of one edge, and all of the edges that are only one edge away from the anchor are visited (**discovered**), and assigned distances of 1

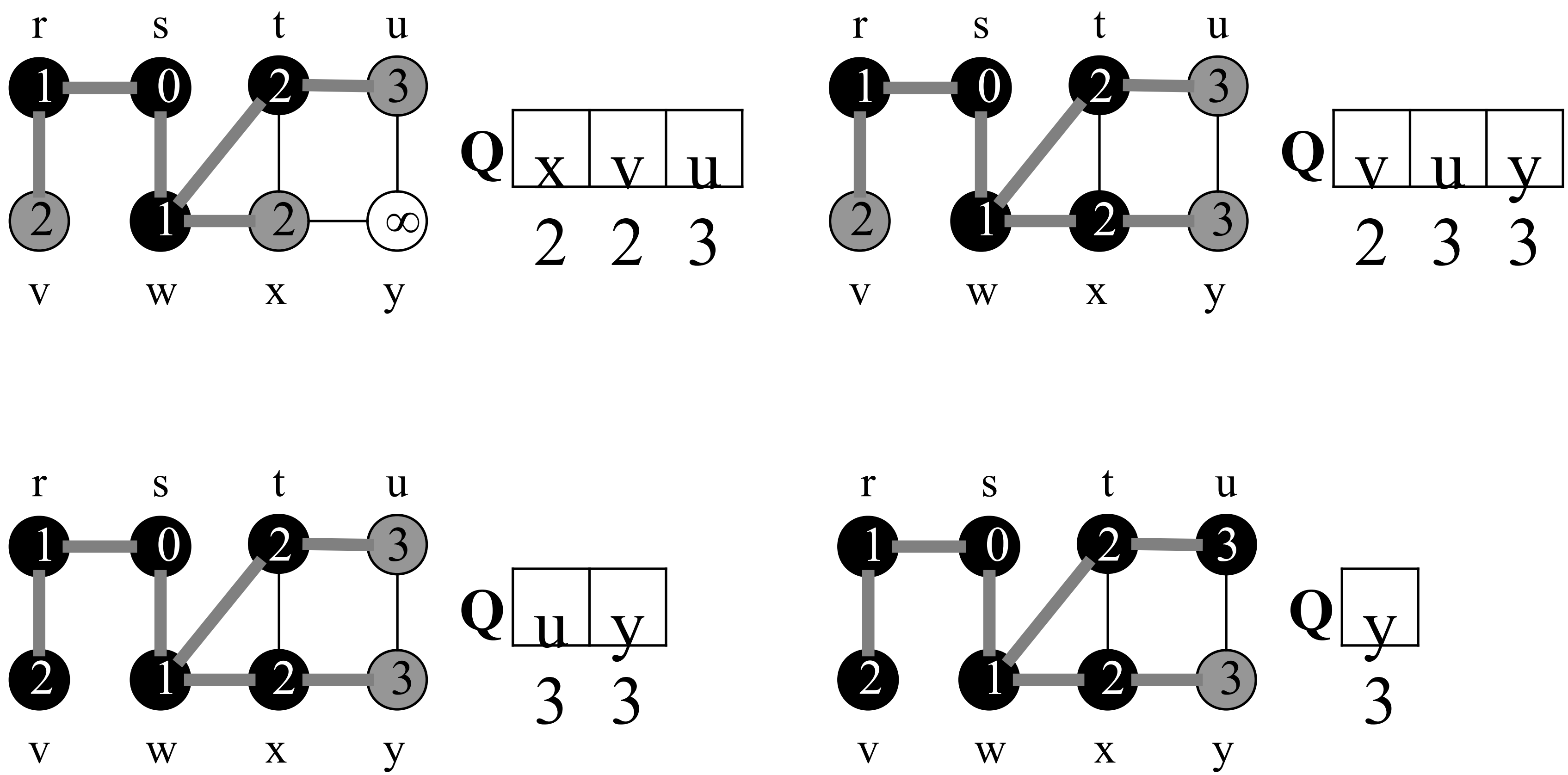
Breadth First Search (2)

- In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and assigned a distance of 2
- This continues until every vertex has been assigned a level
- The label of any vertex *v* corresponds to the length of the shortest path (in terms of edges) from *s* to *v*

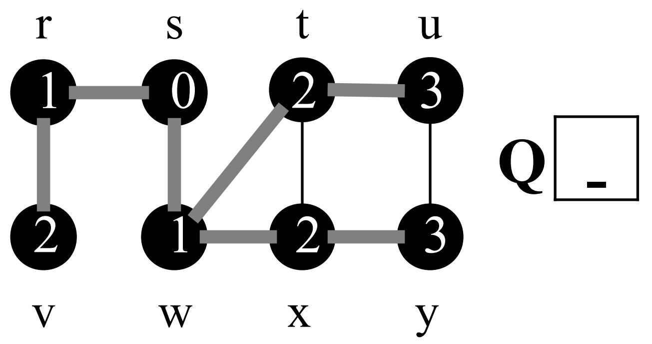
Breadth First Search: example



Breadth First Search: example



Breadth First Search: example



BFS Algorithm

```
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[u] ← NIL
Q ← {s}
while Q ≠ ∅ do
   u ← head[Q]
   for each v ∈ Adj[u] do
      if color[v] = white then
         color[v] ← gray
         d[v] ← d[u] + 1
         π[v] ← u
         Enqueue(Q,v)
   Dequeue(Q)
   color[u] ← black
```

Init all vertices

Init BFS with *s*

Handle all *u*'s children before handling any children of children

BFS Algorithm: running time

- Given a graph G = (V,E)
  - Vertices are enqueued if there color is white
  - Assuming that en- and dequeuing takes O(1) time the total cost of this operation is O(|V|)
  - Adjacency list of a vertex is scanned when the vertex is dequeued (and only then…)
  - The sum of the lengths of all lists is O(|E|). Consequently, O(|E|) time is spent on scanning them
  - Initializing the algorithm takes O(|V|)
- **Total running time O(|V|+|E|)** (linear in the size of the adjacency list representation of G)

BFS Algorithm: properties

- Given a graph G = (V,E), BFS **discovers all vertices reachable from a source vertex *s***
- It computes the **shortest distance** to all reachable vertices
- It computes a **breadth-first tree** that contains all such reachable vertices
- For any vertex *v* reachable from *s*, the path in the breadth first tree from s to v, corresponds to a **shortest path** in G

BFS Tree

- Predecessor subgraph of G

$$G_\pi = (V_\pi, E_\pi)$$
$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$$
$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$$

- G_p is a breadth-first tree
  - V_p consists of the vertices reachable from s, and
  - for all $v \in V_p$, there is a unique simple path from *s* to *v* in G_p that is also a shortest path from *s* to *v* in G
- The edges in G_p are called tree edges

Depth-first search (DFS)

- **A depth-first search (DFS)** in an undirected graph G is like wandering in a labyrinth with a **string** and a **can of paint**
  - We start at vertex *s*, tying the end of our string to the point and painting *s* "visited (discovered)". Next we label *s* as our current vertex called *u*
  - Now, we travel along an arbitrary edge (*u*,*v*).
  - If edge (*u*,*v*) leads us to an already visited vertex *v* we return to *u*
  - If vertex *v* is unvisited, we unroll our string, move to *v*, paint *v* "visited", set *v* as our current vertex, and repeat the previous steps

Depth-first search (2)

- Eventually, we will get to a point where **all incident edges on *u* lead to visited vertices**
- We then **backtrack** by unrolling our string to a previously visited vertex *v*. Then *v* becomes our current vertex and we repeat the previous steps
- Then, if all incident edges on *v* lead to visited vertices, we backtrack as we did before. We **continue to backtrack along the path we have traveled**, finding and exploring unexplored edges, and repeating the procedure

Depth-first search algorithm

- Initialize – color all vertices white
- Visit each and every white vertex using DFS-Visit
- Each call to DFS-Visit(u) roots a new tree of the depth-first forest at vertex u
- A vertex is **white** if it is undiscovered
- A vertex is **gray** if it has been discovered but not all of its edges have been discovered
- A vertex is **black** after all of its adjacent vertices have been discovered (the adj. list was examined completely)

Depth-first search algorithm (2)

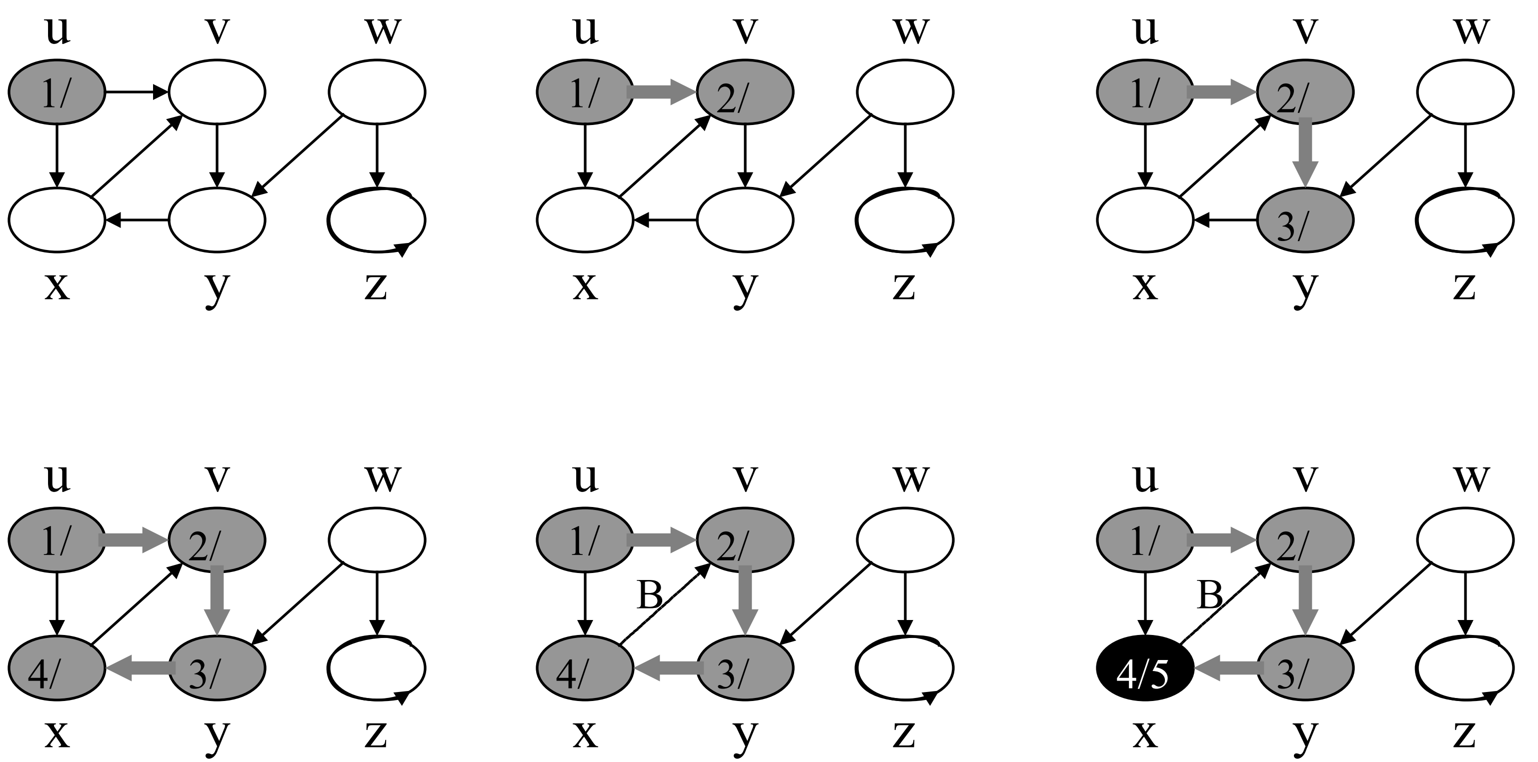
```
DFS(G)
for each vertex u ∈ V[G]
        do color[u] ← WHITE
time ← 0
for each vertex u ∈ V[G]
        do if color[u] = WHITE
                then DFS-VISIT(u)
```

Init all vertices

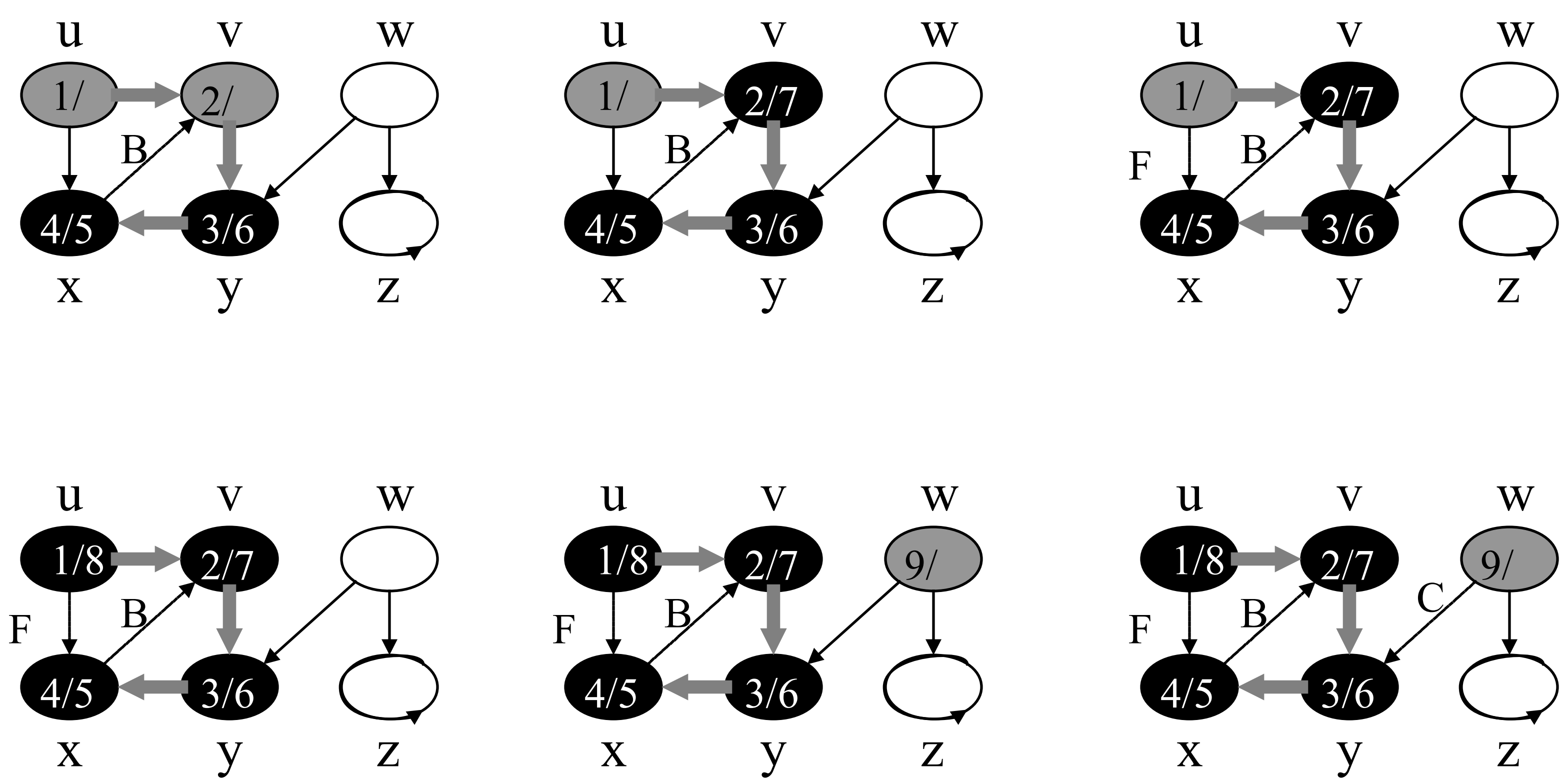
```
DFS-VISIT(u)
color[u] ← GRAY                 ▷ White vertex u discovered.
d[u] ← time                     ▷ Mark with discovery time.
time ← time + 1                 ▷ Tick global time.
for each v ∈ Adj[u]             ▷ Explore all edges (u, v).
        do if color[v] = WHITE
                then DFS-VISIT(v)
color[u] ← BLACK                ▷ Blacken u; it is finished.
f[u] ← time                     ▷ Mark with finishing time.
time ← time + 1                 ▷ Tick global time.
```

Visit all children recursively

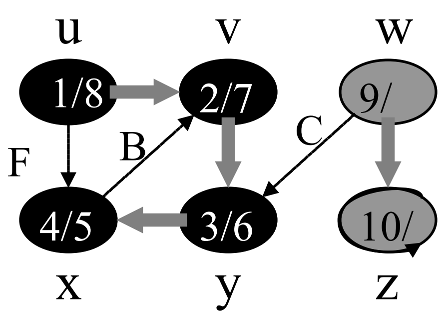
Depth-first search example

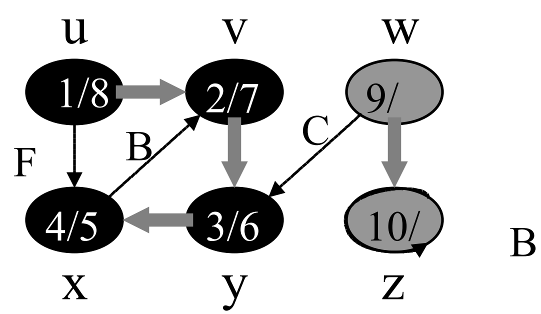


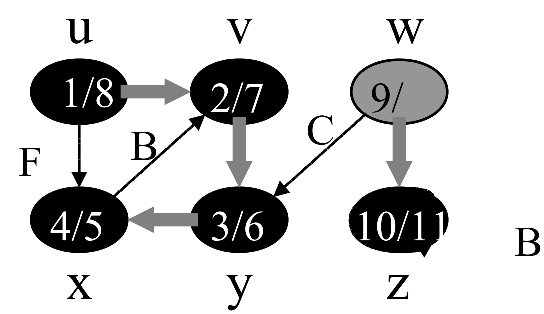
Depth-first search example (2)

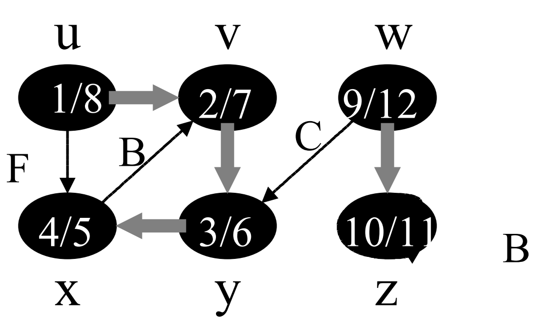


Depth-first search example (3)









Depth-first search example (4)

- When DFS returns, every vertex *u* is assigned
  - a discovery time *d*[*u*], and a finishing time *f*[*u*]
- Running time
  - the loops in DFS take time $\Theta(V)$ each, excluding the time to execute DFS-Visit
  - DFS-Visit is called once for every vertex
    - its only invoked on white vertices, and
    - paints the vertex gray immediately
  - for each DFS-visit a loop interates over all Adj[*v*]
  - the total cost for DFS-Visit is $\Theta(E)$

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

  - **the running time of DFS is $\Theta(V+E)$**

Predecessor Subgraph

- Defined slightly different from BFS

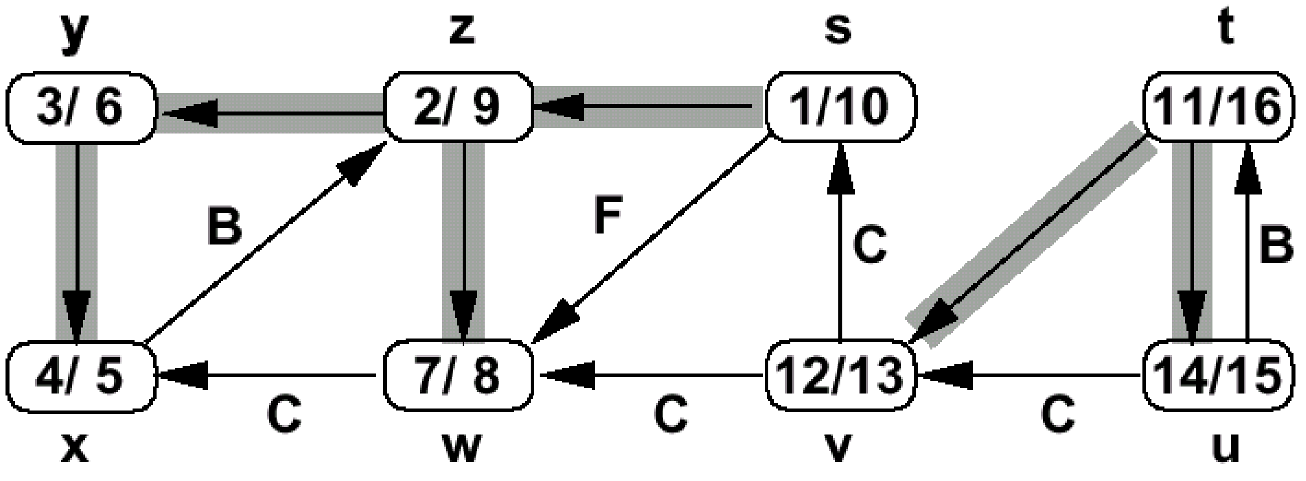
$$G_\pi = (V, E_\pi)$$

$$E_\pi = \left\{ (\pi[v], v) \in E : v \in V \text{ and } \pi[v] \neq \text{NIL} \right\}$$

- The PD subgraph of a depth-first search forms a **depth-first forest** composed of several depth-first trees
- The edges in G_p are called tree edges

DFS Timestamping

- The DFS algorithm maintains a monotonically increasing global clock
  - discovery time *d*[*u*] and finishing time *f*[*u*]
- For every vertex *u*, the inequality *d*[*u*] < *f*[*u*] must hold



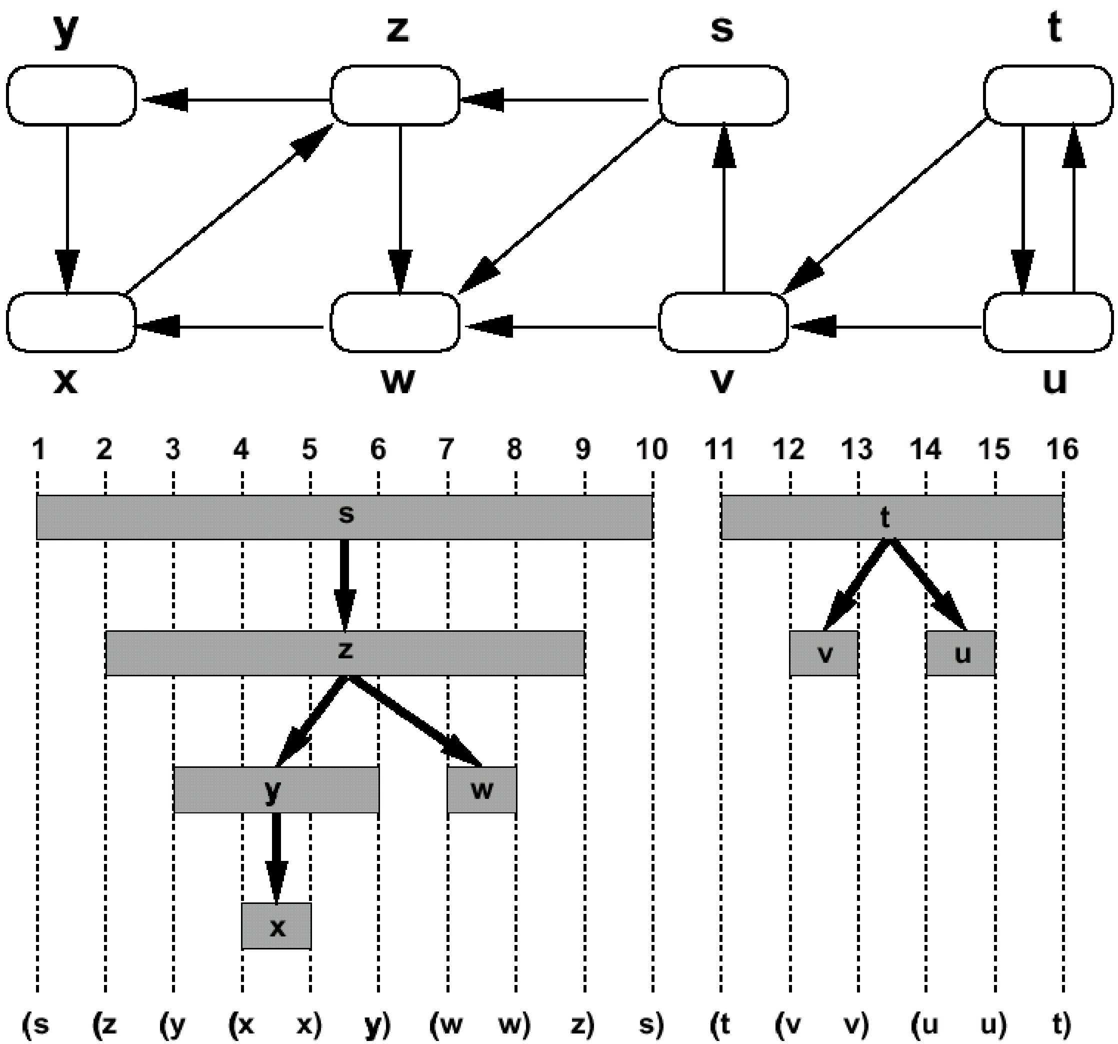
DFS Timestamping

- Vertex *u* is
  - white before time *d*[*u*]
  - gray between time *d*[*u*] and time *f*[*u*], and
  - black thereafter
- Notice the structure througout the algorithm.
  - gray vertices form a linear chain
  - correponds to a stack of vertices that have not been exhaustively explored (DFS-Visit started but not yet finished)

DFS Parenthesis Theorem

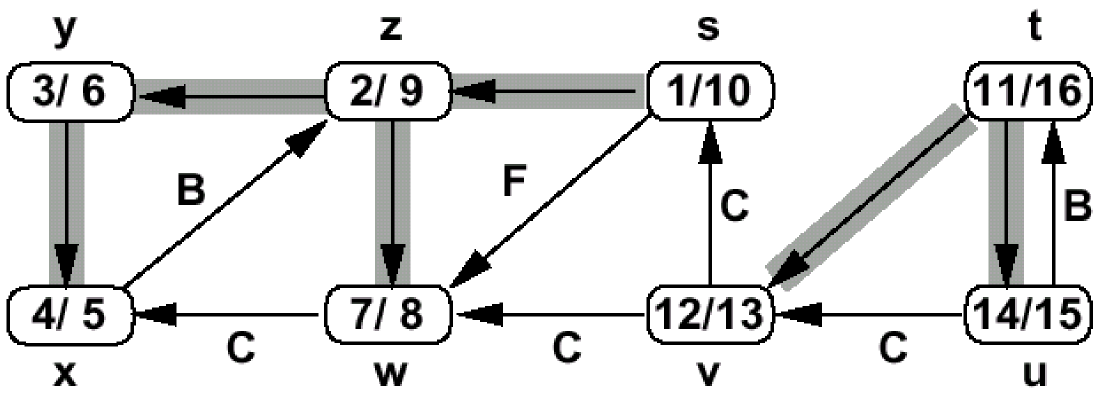
- Discovery and finish times have parenthesis structure
  - represent discovery of *u* with left parenthesis "(u"
  - represent finishin of *u* with right parenthesis "u)"
  - history of discoveries and finishings makes a well-formed expression (parenthesis are properly nested)
- Intuition for proof: any two intervals are either disjoint or enclosed
  - Overlaping intervals would mean finishing ancestor, before finishing descendant or starting descendant without starting ancestor

DFS Parenthesis Theorem (2)



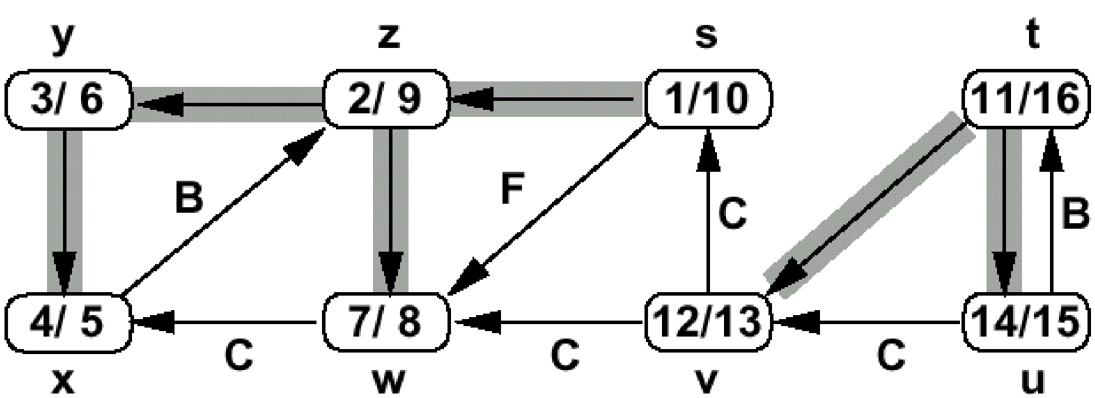
DFS Edge Classification

- Tree edge (gray to white)
  – encounter new vertices (white)
- Back edge (gray to gray)
  – from descendant to ancestor



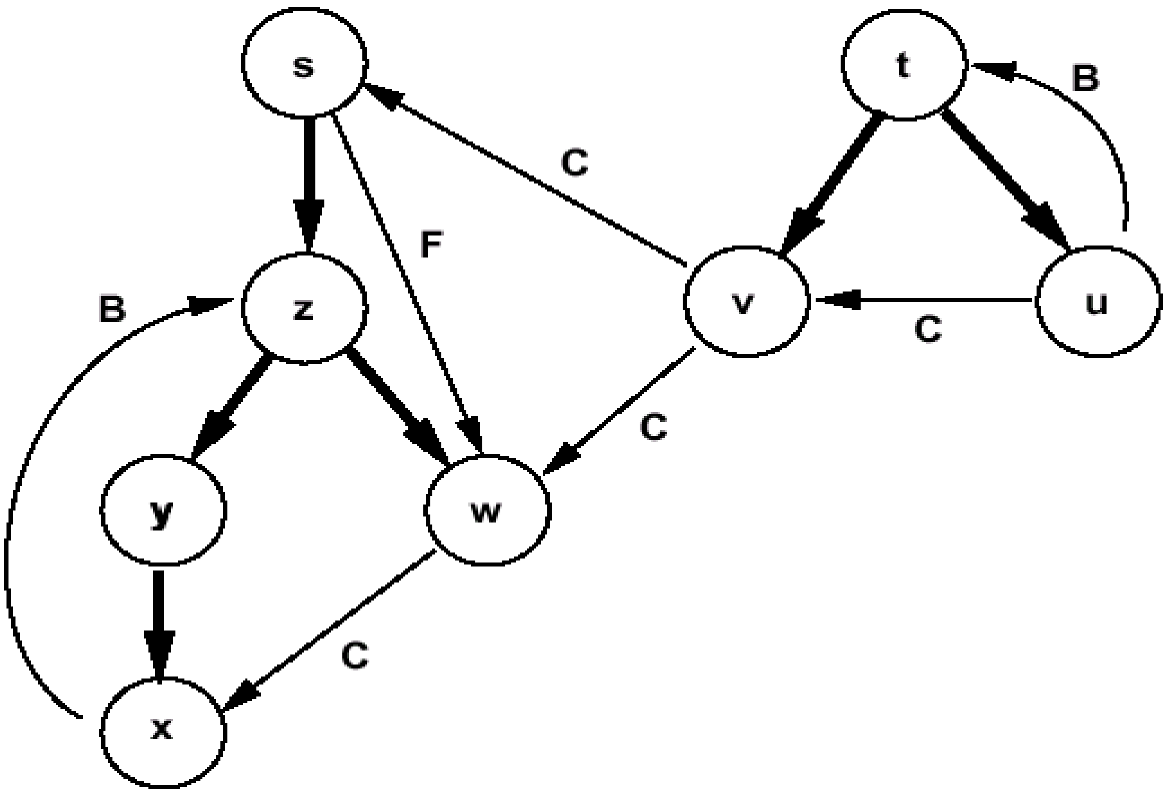
DFS Edge Classification (2)

- Forward edge (gray to black)
  - from ancestor to descendant
- Cross edge (gray to black)
  - remainder – between trees or subtrees



DFS Edge Classification (3)

- Tree and back edges are important
- Most algorithms do not distinguish between forward and cross edges

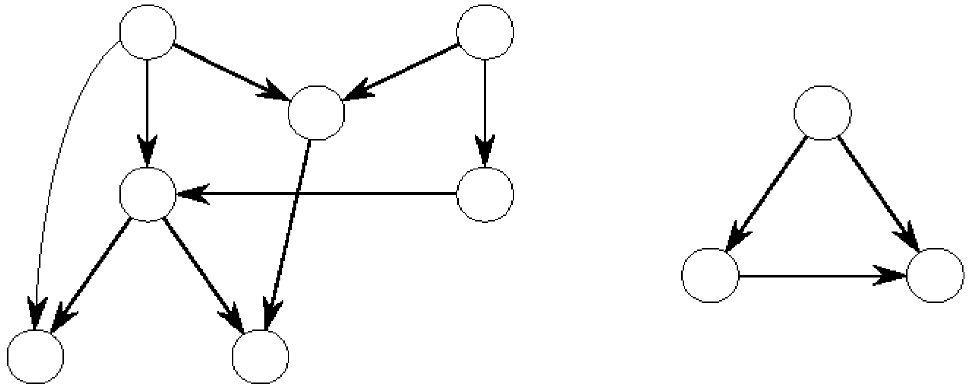


Next:

- Application of DFS: Topological Sort

Directed Acyclic Graphs

- A DAG is a directed graph with no cycles



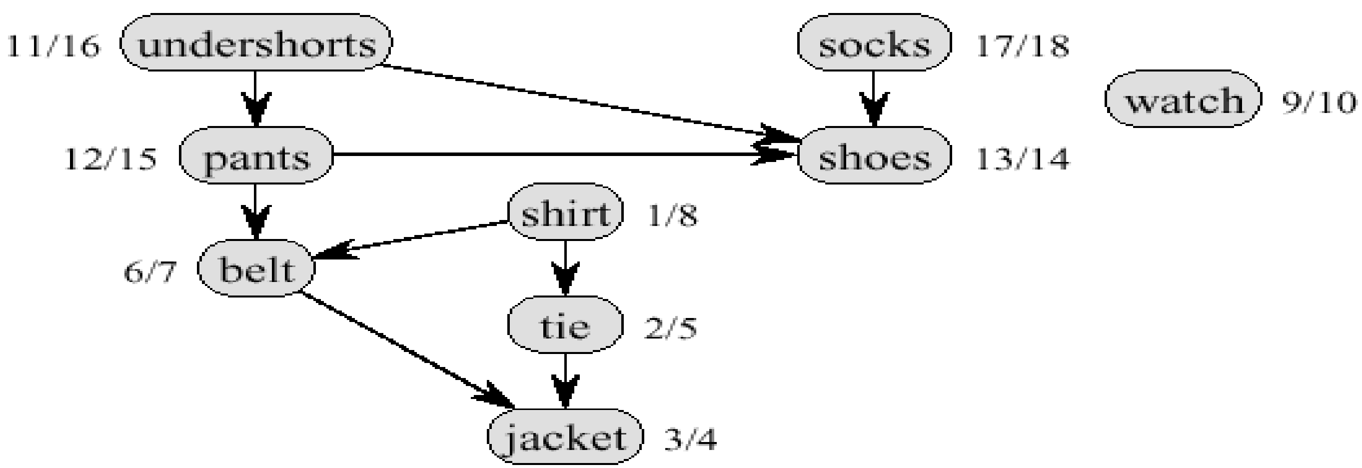
- Often used to indicate precedences among events, i.e., event *a* must happen before *b*
- An example would be a parallel code execution
- Total order can be introduced using **Topological Sorting**

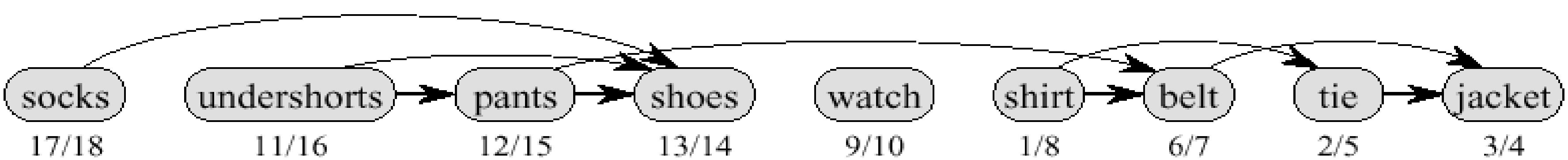
DAG Theorem

- A directed graph *G* is acyclic if and only if a DFS of *G* yields no back edges. Proof:
  - **suppose there is a back edge (*u*,*v*);** *v* is an ancestor of *u* in DFS forest. Thus, there is a path from *v* to *u* in G and (*u*,*v*) completes the cycle
  - **suppose there is a cycle *c*;** let *v* be the first vertex in *c* to be discovered and *u* is a predecessor of *v* in *c*.
    - Upon discovering *v* the whole cycle from *v* to *u* is white
    - We must visit all nodes reachable on this white path before return DFS-Visit(*v*), i.e., vertex *u* becomes a descendant of *v*
    - Thus, (*u*,*v*) is a back edge
- Thus, we can verify a DAG using DFS!

Topological Sort Example

- Precedence relations: an edge from *x* to *y* means one must be done with *x* before one can do *y*
- Intuition: can schedule task only when all of its subtasks have been scheduled





Topological Sort

- Sorting of a directed acyclic graph (DAG)
- A topological sort of a DAG is a linear ordering of all its vertices such that for any edge (u,v) in the DAG, u appears before v in the ordering
- The following algorithm topologically sorts a DAG

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

- The linked lists comprises a total ordering

Topological Sort

- Running time
  - depth-first search: $O(V+E)$ time
  - insert each of the $|V|$ vertices to the front of the linked list: $O(1)$ per insertion
- Thus the total running time is $O(V+E)$

Topological Sort Correctness

- Claim: for a DAG, an edge $(u,v) \in E \Rightarrow f[u] > f[v]$
- When (u,v) explored, u is gray. We can distinguish three cases
  - v = gray
    $\Rightarrow$ (u,v) = back edge (cycle, contradiction)
  - v = white
    $\Rightarrow$ v becomes descendant of u
    $\Rightarrow$ v will be finished before u
    $\Rightarrow$ $f[v] < f[u]$
  - v = black
    $\Rightarrow$ v is already finished
    $\Rightarrow$ $f[v] < f[u]$
- The definition of topological sort is satisfied