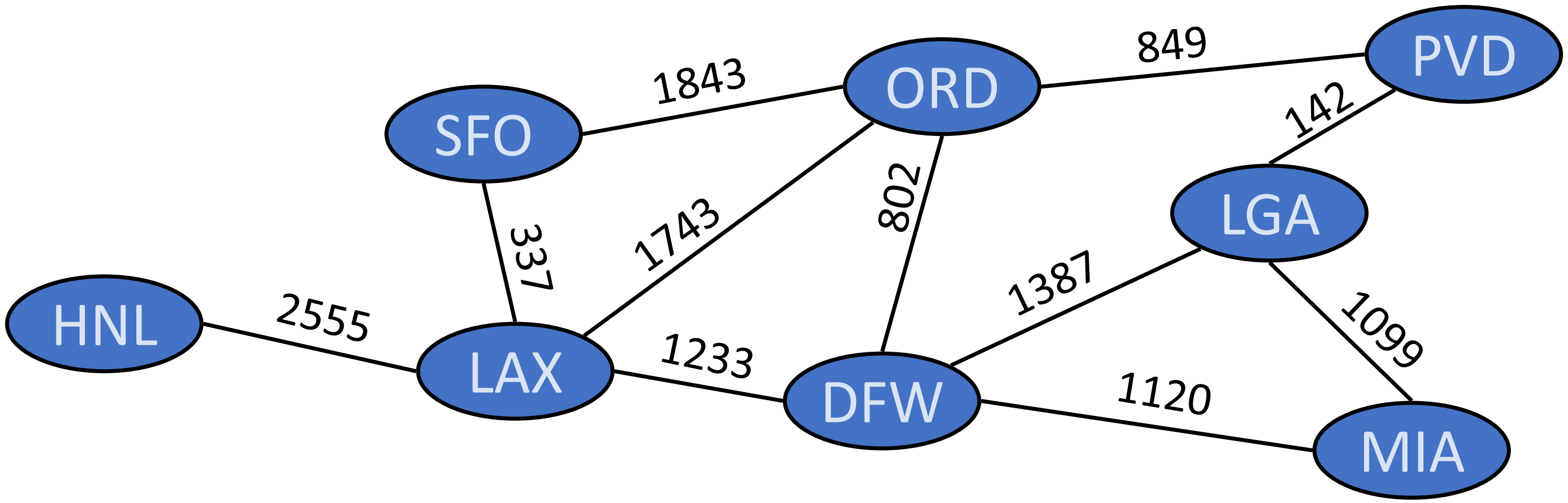
Graphs

Graph

- A graph is a pair $(\boldsymbol{V}, \boldsymbol{E})$, where
  - $\boldsymbol{V}$ is a set of nodes, called vertices
  - $\boldsymbol{E}$ is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



Edge & Graph Types

- Edge Types
  - Directed edge
    - ordered pair of vertices $(\boldsymbol{u},\boldsymbol{v})$
    - first vertex $\boldsymbol{u}$ is the origin
    - second vertex $\boldsymbol{v}$ is the destination
    - e.g., a flight
  - Undirected edge
    - unordered pair of vertices $(\boldsymbol{u},\boldsymbol{v})$
    - e.g., a flight route
  - Weighted edge

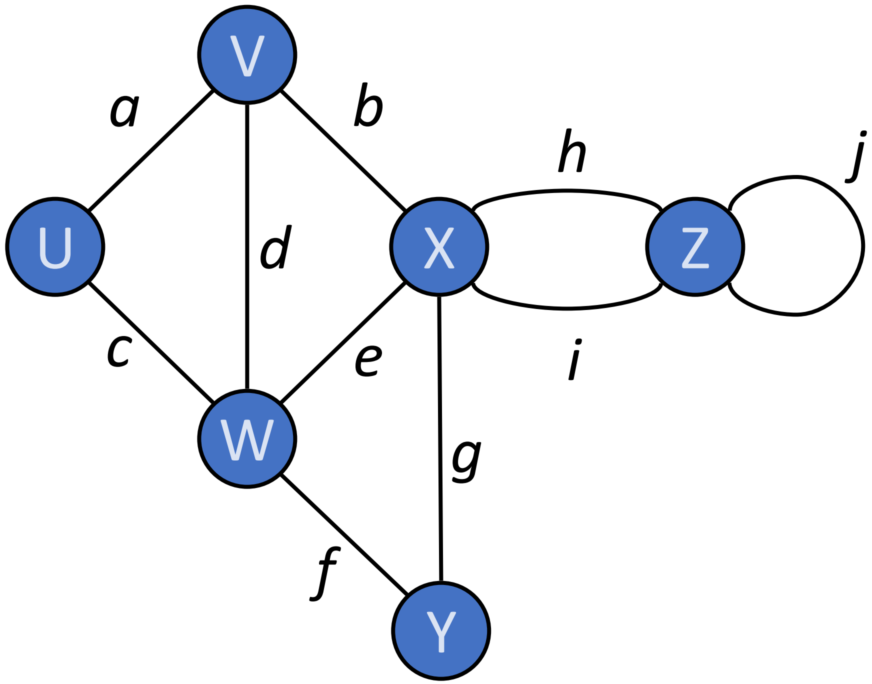
- Graph Types
  - Directed graph (Digraph)
    - all the edges are directed
  - Undirected graph
    - all the edges are undirected
  - Weighted graph
    - all the edges are weighted

A Few Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
- Databases
  - Entity-relationship diagram

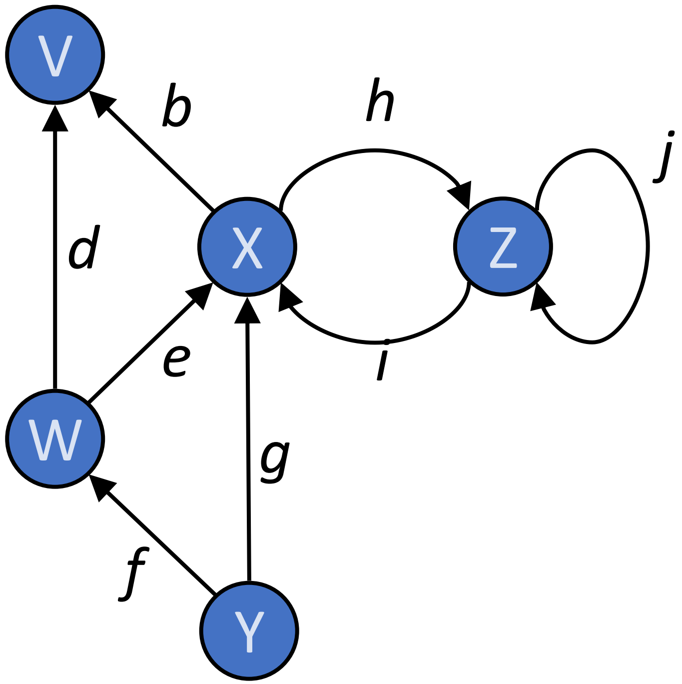
Terminology

- End points (or end vertices) of an edge
  - U and V are the *endpoints* of *a*
- Edges incident on a vertex
  - *a*, *d*, and *b* are *incident* on V
- Adjacent vertices
  - U and V are *adjacent*
- Degree of a vertex
  - X has *degree* 5
- Parallel (multiple) edges
  - *h* and *i* are *parallel edges*
- self-loop
  - j is a *self-loop*



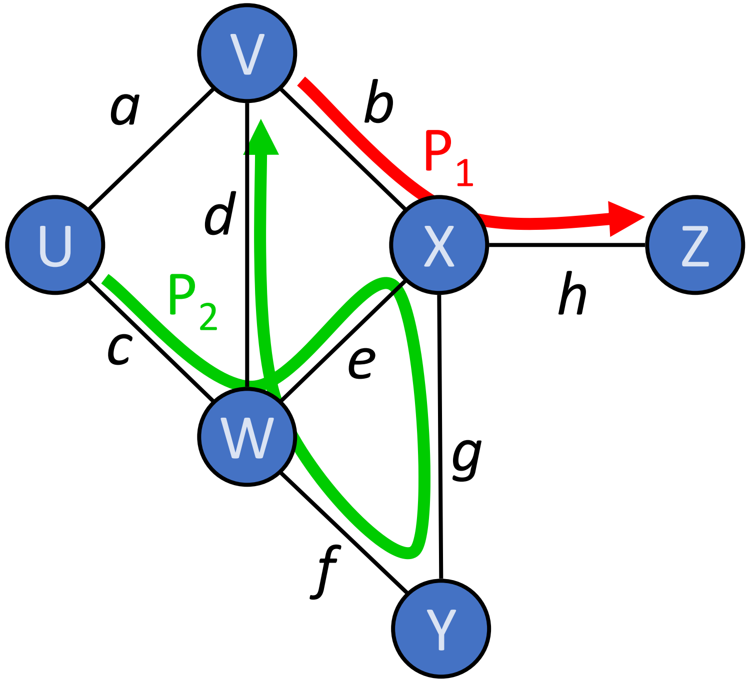
Terminology (cont.)

- outgoing edges of a vertex
  - *h* and *b* are the *outgoing edges* of X
- incoming edges of a vertex
  - *e*, *g*, and *i* are *incoming edges* of X
- in-degree of a vertex
  - X has *in-degree* 3
- out-degree of a vertex
  - X has *out-degree* 2



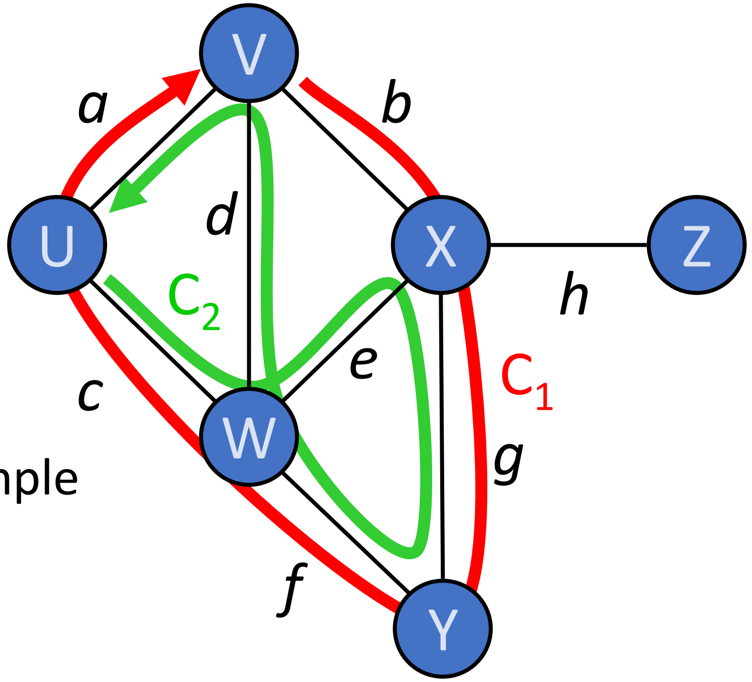
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - **C_1=(V,b,X,g,Y,f,W,c,U,a,↵)** is a simple cycle
  - **C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵)** is a cycle that is not simple



Properties of Undirected Graphs

Property 1 – Total degree

$\Sigma_{\boldsymbol{v}} \deg(\boldsymbol{v}) = ?$

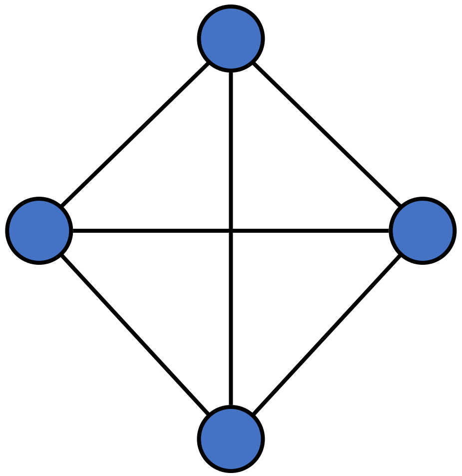
Property 2 – Total number of edges

In an undirected graph with no self-loops and no multiple edges

$\boldsymbol{m} \leq \textbf{\textit{Upper bound?}}$

Notation

$\boldsymbol{n}$	number of vertices
$\boldsymbol{m}$	number of edges
$\deg(\boldsymbol{v})$	degree of vertex $\boldsymbol{v}$



A graph with given number of vertices (4) and maximum number of edges

Example

- $n = ?$
- $m = ?$
- $\deg(v) = ?$

Properties of Undirected Graphs

Property 1 – Total degree

$\sum_{\boldsymbol{v}} \deg(\boldsymbol{v}) = 2\boldsymbol{m}$

Proof: each edge is counted twice

Property 2 – Total number of edges

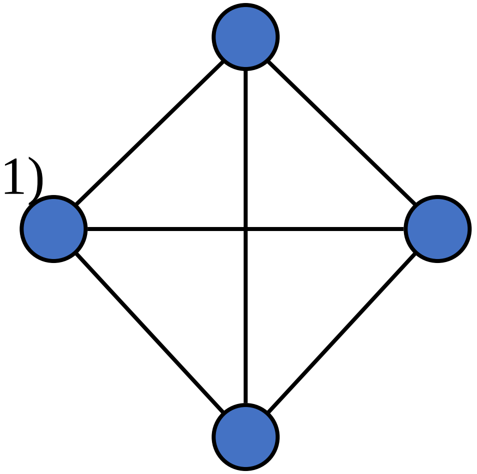
In an undirected graph with no self-loops and no multiple edges

$\boldsymbol{m} \leq \boldsymbol{n}\,(\boldsymbol{n} - 1)/2$

Proof: each vertex has degree at most $(\boldsymbol{n} - 1)$

Notation

$\boldsymbol{n}$	number of vertices
$\boldsymbol{m}$	number of edges
$\deg(\boldsymbol{v})$	degree of vertex $\boldsymbol{v}$



A graph with given number of vertices (4) and maximum number of edges

Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Properties of Directed Graphs

Property 1 – Total in-degree and out-degree

$\sum_v \text{in-deg}(\boldsymbol{v}) = ?$

$\sum_v \text{out-deg}(\boldsymbol{v}) = ?$

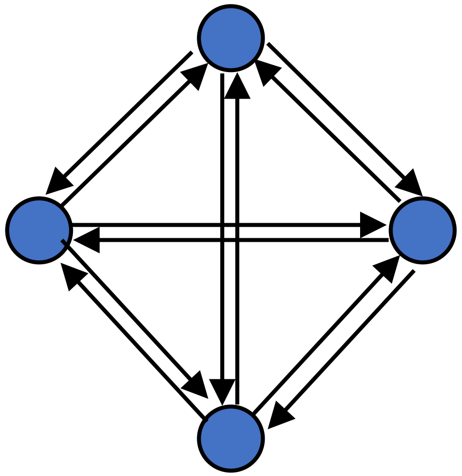
Property 2 – Total number of edges

In a directed graph with no self-loops and no multiple edges

$\boldsymbol{m \leq \textit{Upper bound?}}$

Notation

$\boldsymbol{n}$	number of vertices
$\boldsymbol{m}$	number of edges
$\deg(\boldsymbol{v})$	degree of vertex $\boldsymbol{v}$



A graph with given number of vertices (4) and maximum number of edges

Example

- $n = ?$
- $m = ?$
- $\deg(v) = ?$

Properties of Directed Graphs

Property 1 – Total in-degree and out-degree

$\sum_v \text{in-deg}(\boldsymbol{v}) = m$

$\sum_v \text{out-deg}(\boldsymbol{v}) = m$

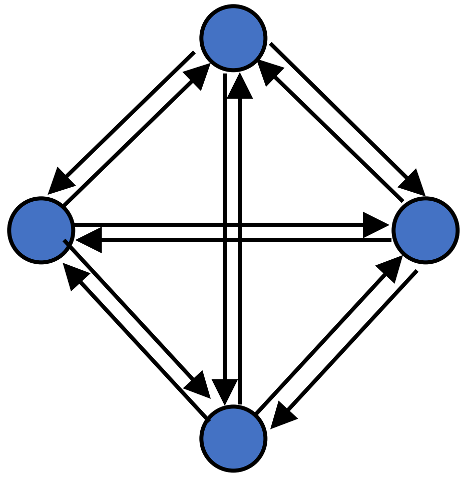
Property 2 – Total number of edges

In a directed graph with no self-loops and no multiple edges

$\boldsymbol{m} \leq \boldsymbol{n}\,(\boldsymbol{n} - 1)$

Notation

$\boldsymbol{n}$	number of vertices
$\boldsymbol{m}$	number of edges
$\deg(\boldsymbol{v})$	degree of vertex $\boldsymbol{v}$



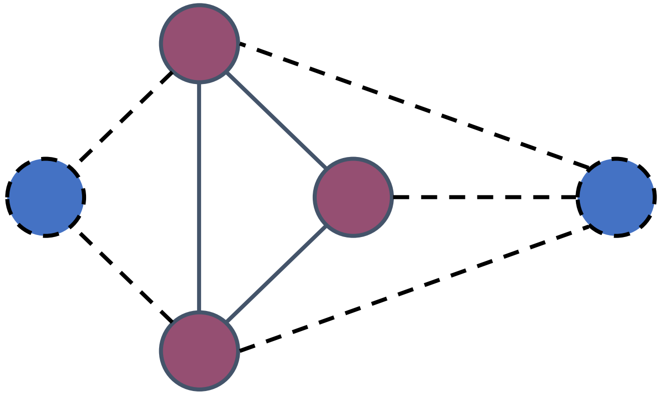
A graph with given number of vertices (4) and maximum number of edges

Example

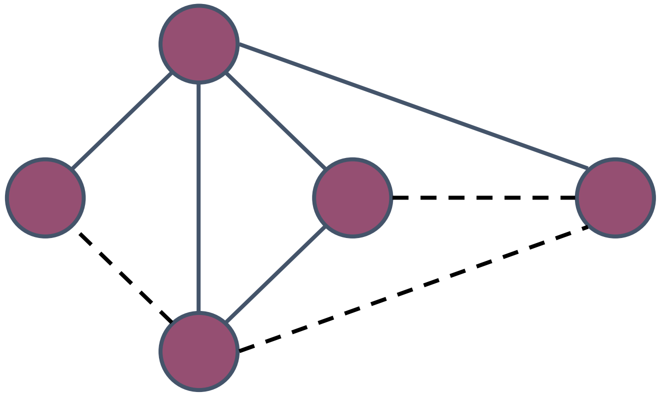
- $n = 4$
- $m = 12$
- $\deg(v) = 6$

Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



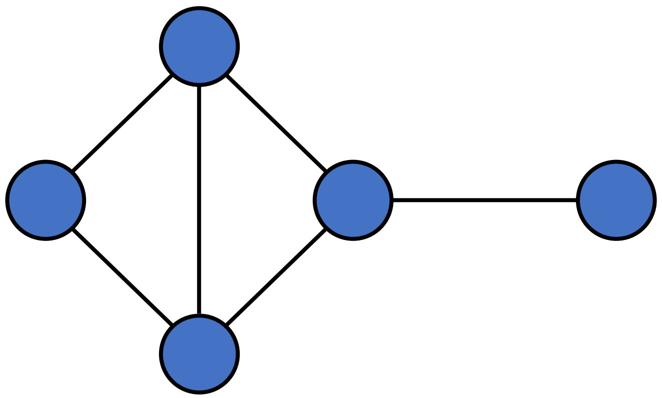
Subgraph



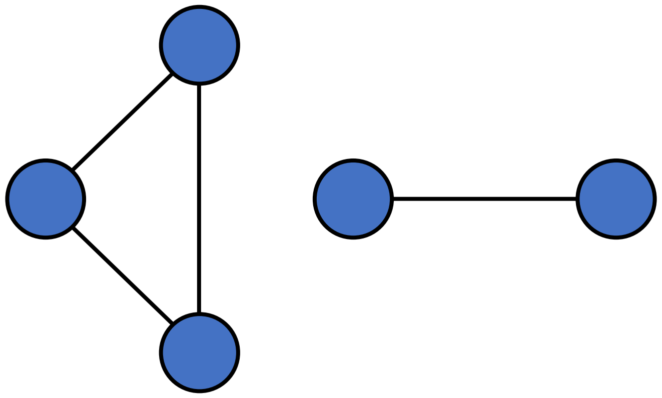
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



Connected graph

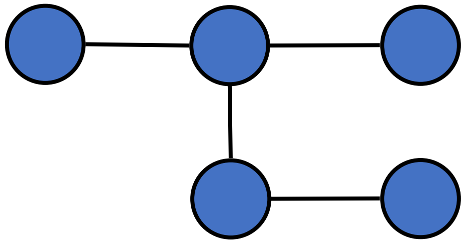


Non connected graph with two connected components

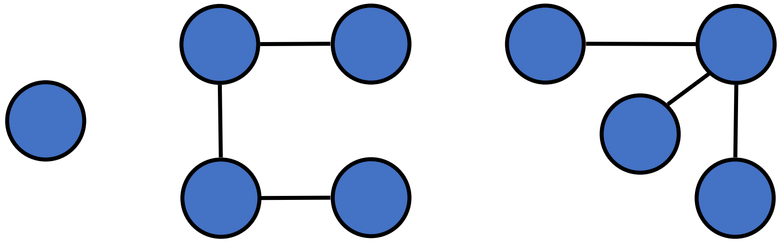
Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



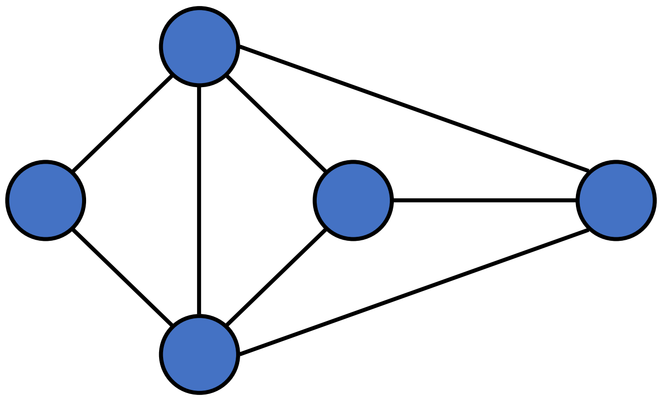
Tree



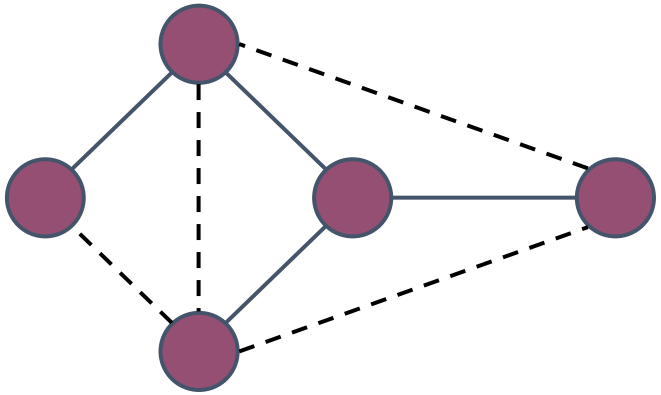
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks



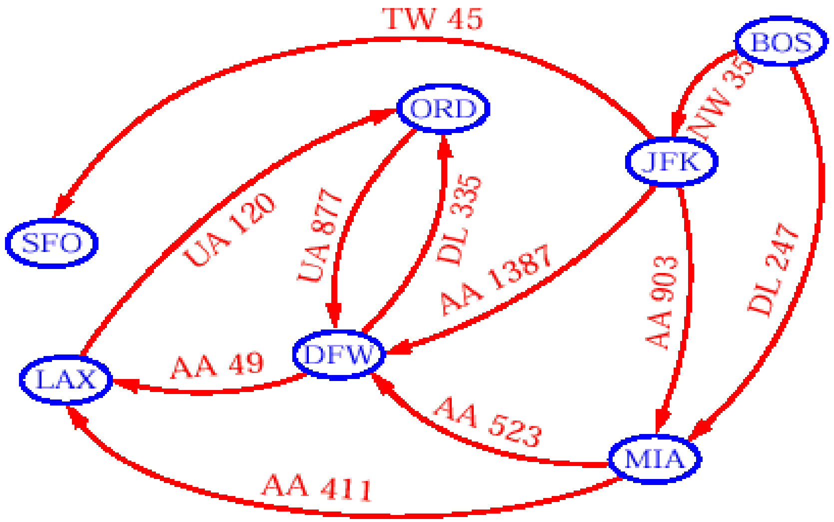
Graph



Spanning tree

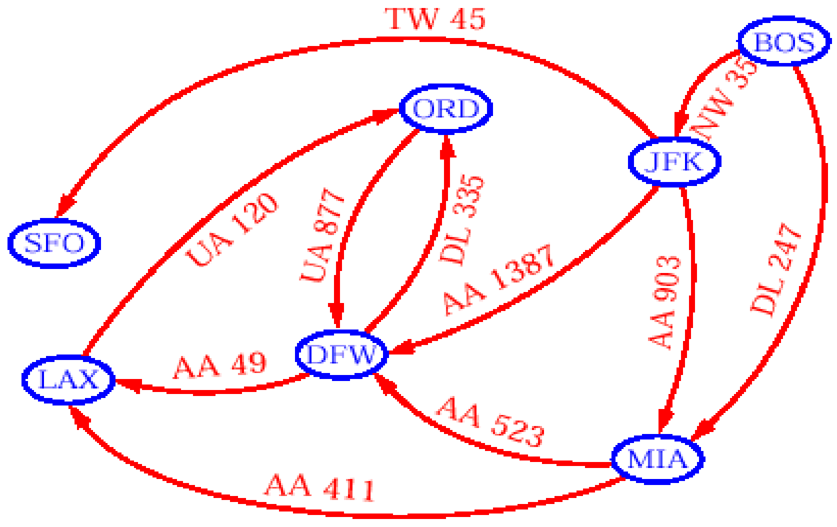
Data Structures for Graphs

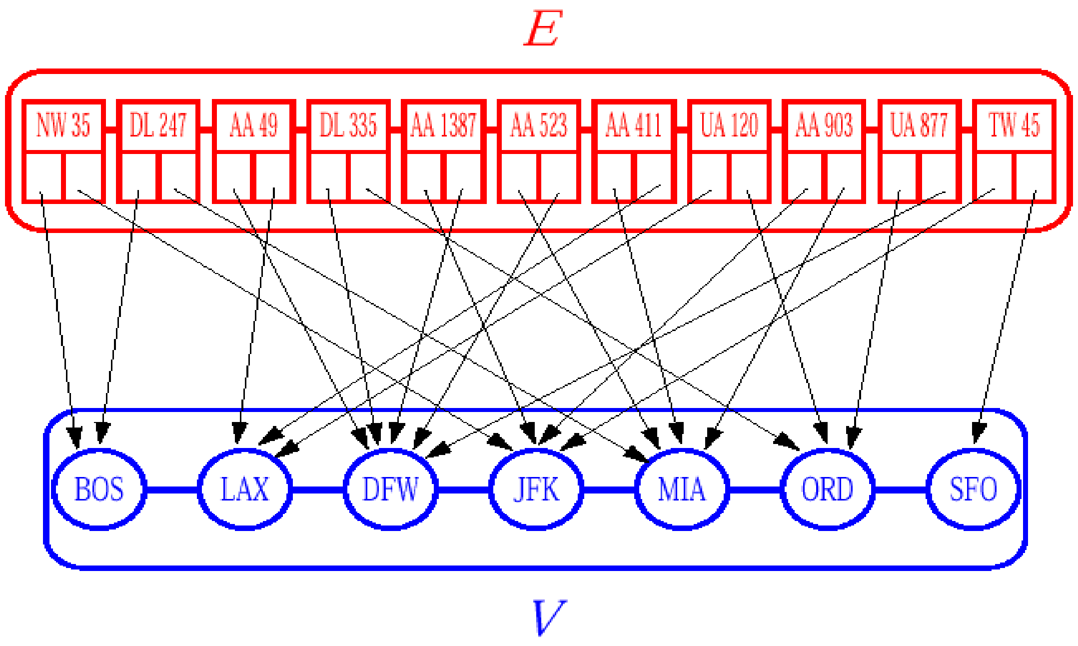
- How can we represent a graph?
  - To start with, we can store the vertices and the edges into two containers, and we store with each edge object references to its start and end vertices



Edge List (Basic)

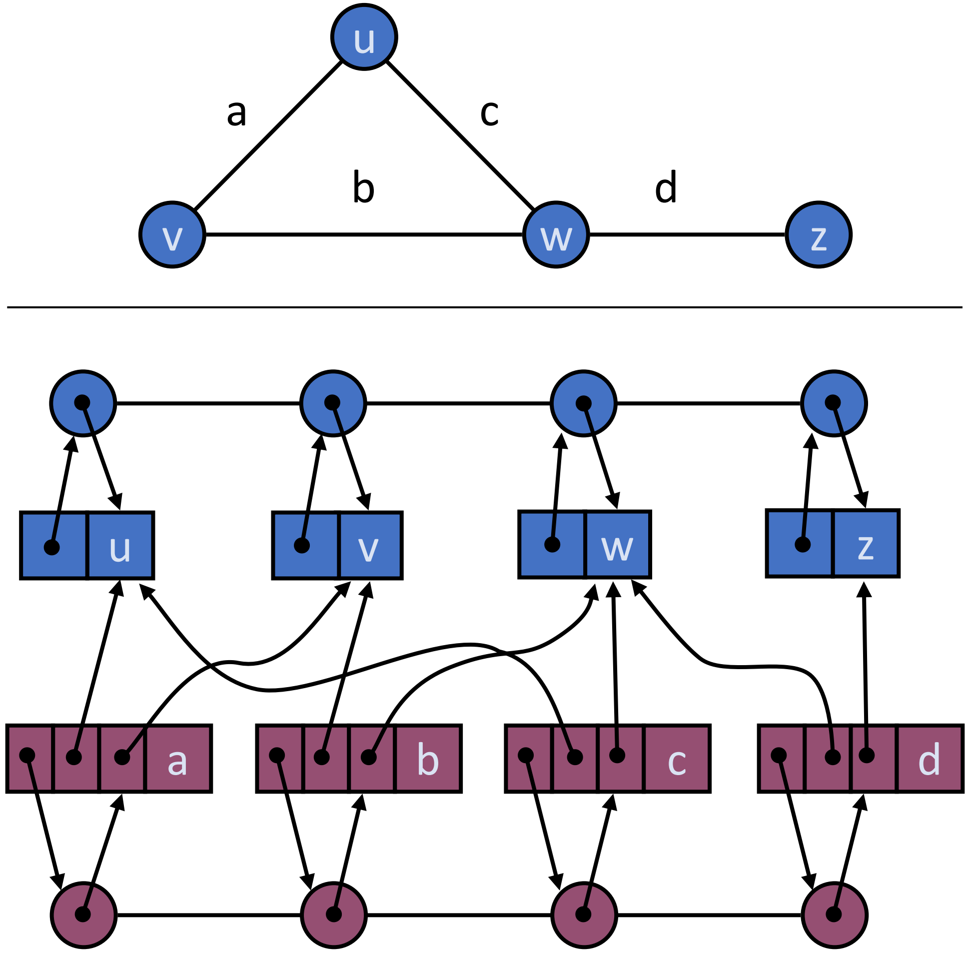
- The **edge list**
  - Easy to implement
  - Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence



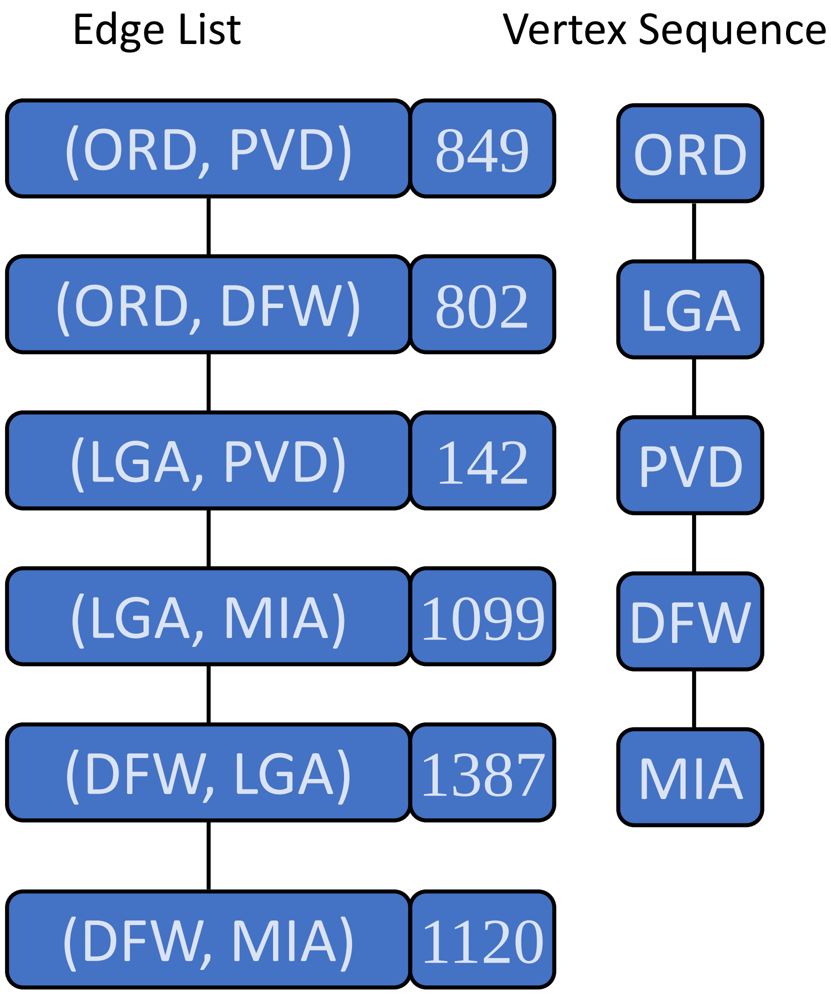


Edge List Structure

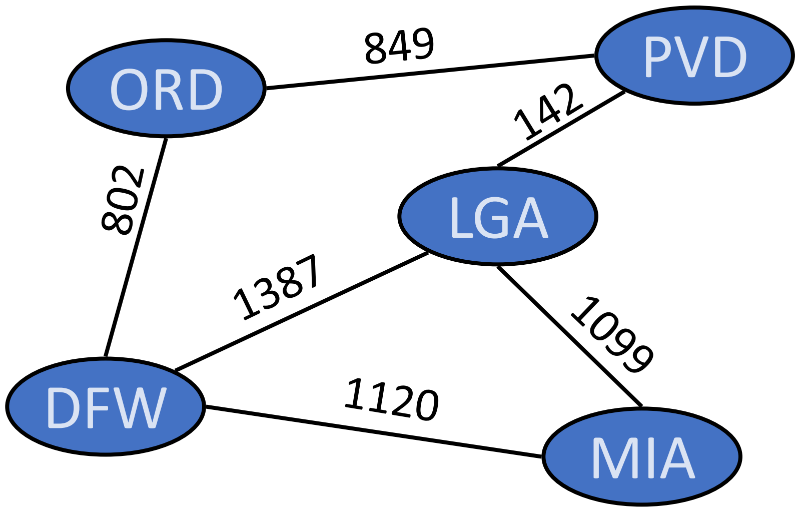
- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects



Edge List Structure (Augment)

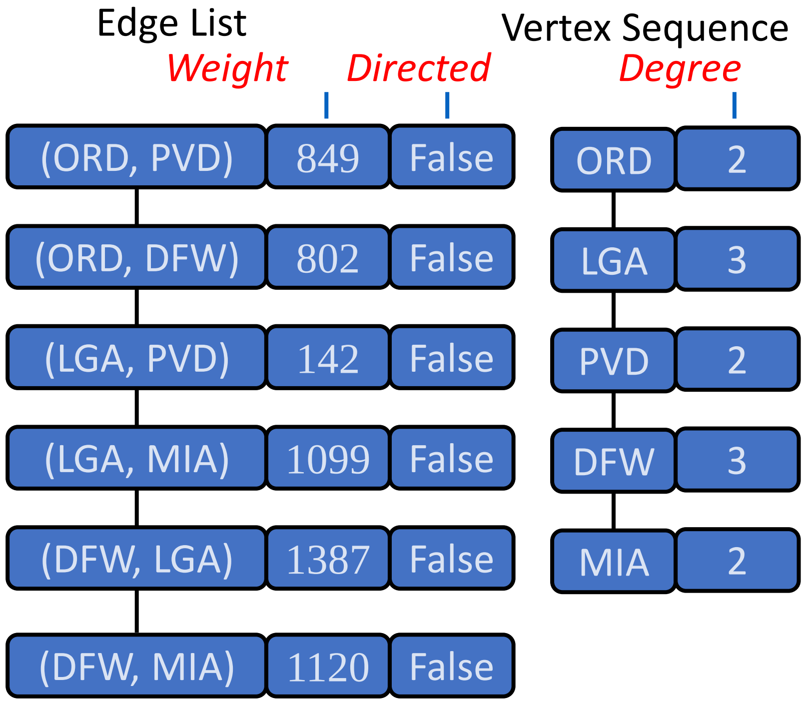


- An edge list can be stored in a sequence, a vector, a list or a dictionary such as a hash table



Asymptotic Performance of Edge List Structure

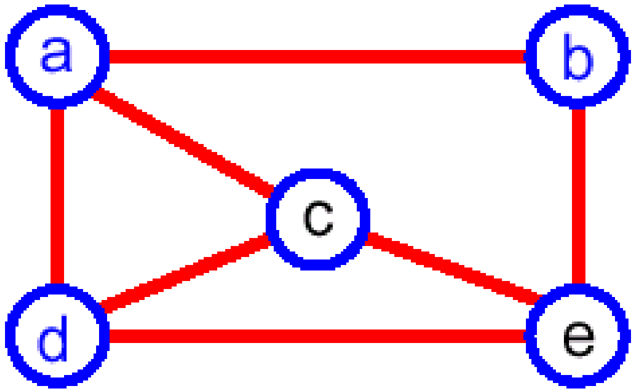
◆ n vertices, m edges ◆ no parallel edges ◆ no self-loops ◆ Bounds are "big-Oh"	Edge List
Space	$n + m$
incidentEdges(v) adjacentVertices(v)	m
areAdjacent (v, w)	m

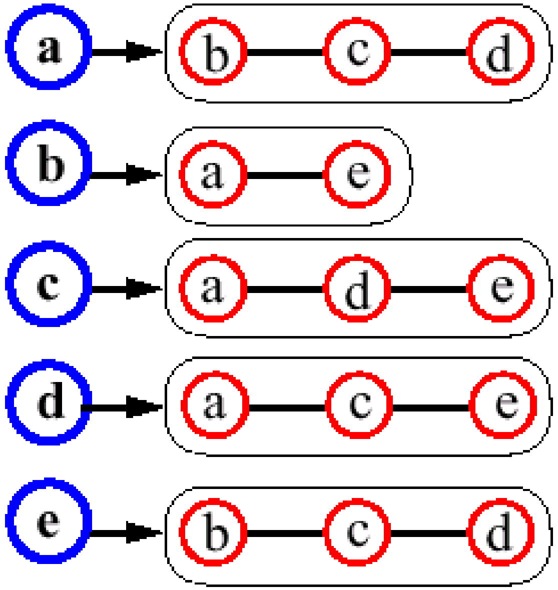


Ways to augment....

Adjacency List (Traditional)

- 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

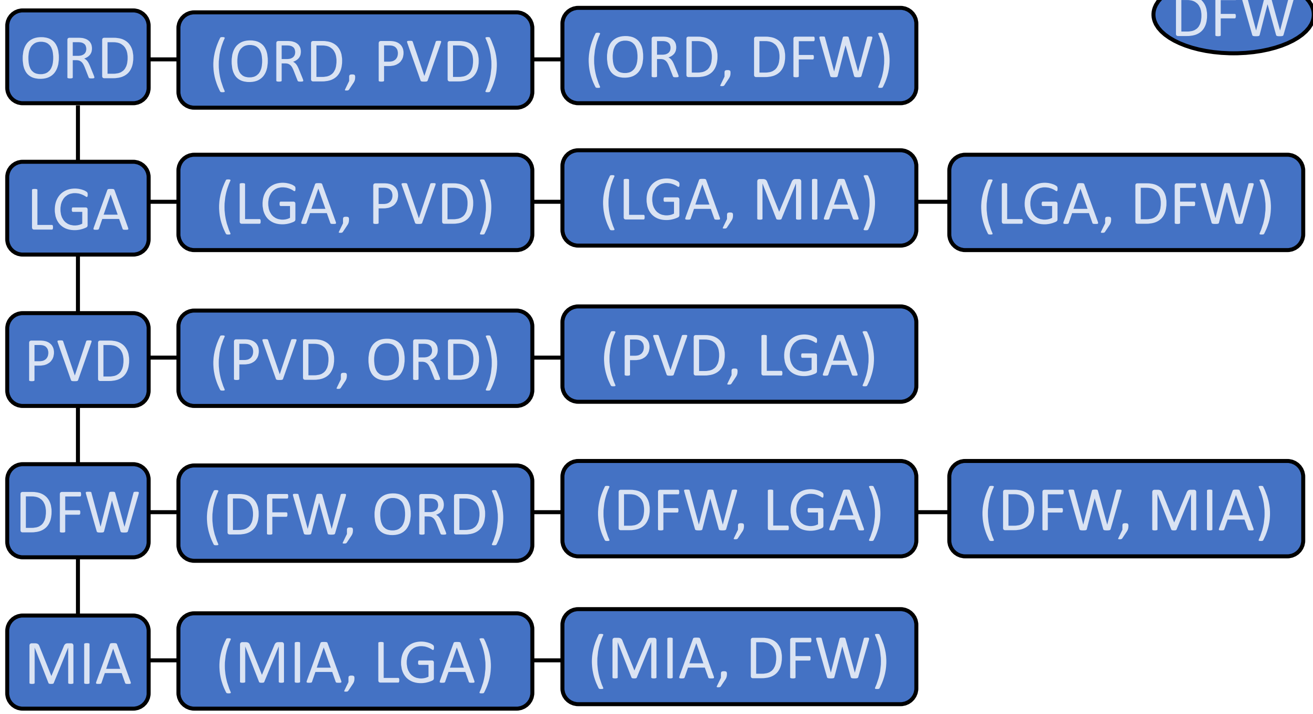


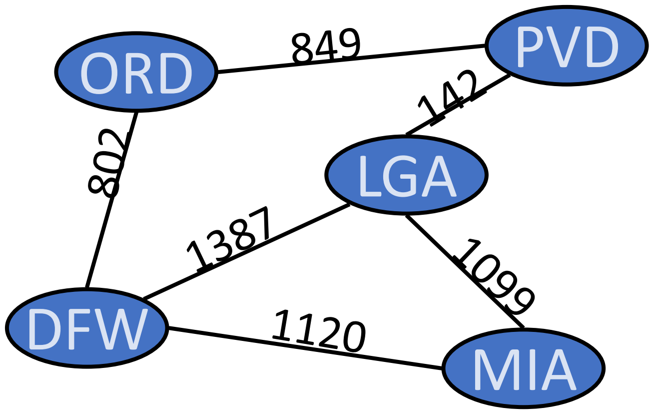


$$\text{Space} = \Theta(n + \sum \deg(v)) = \Theta(n+m)$$

Adjacency List Structure (Modern)

Adjacency List

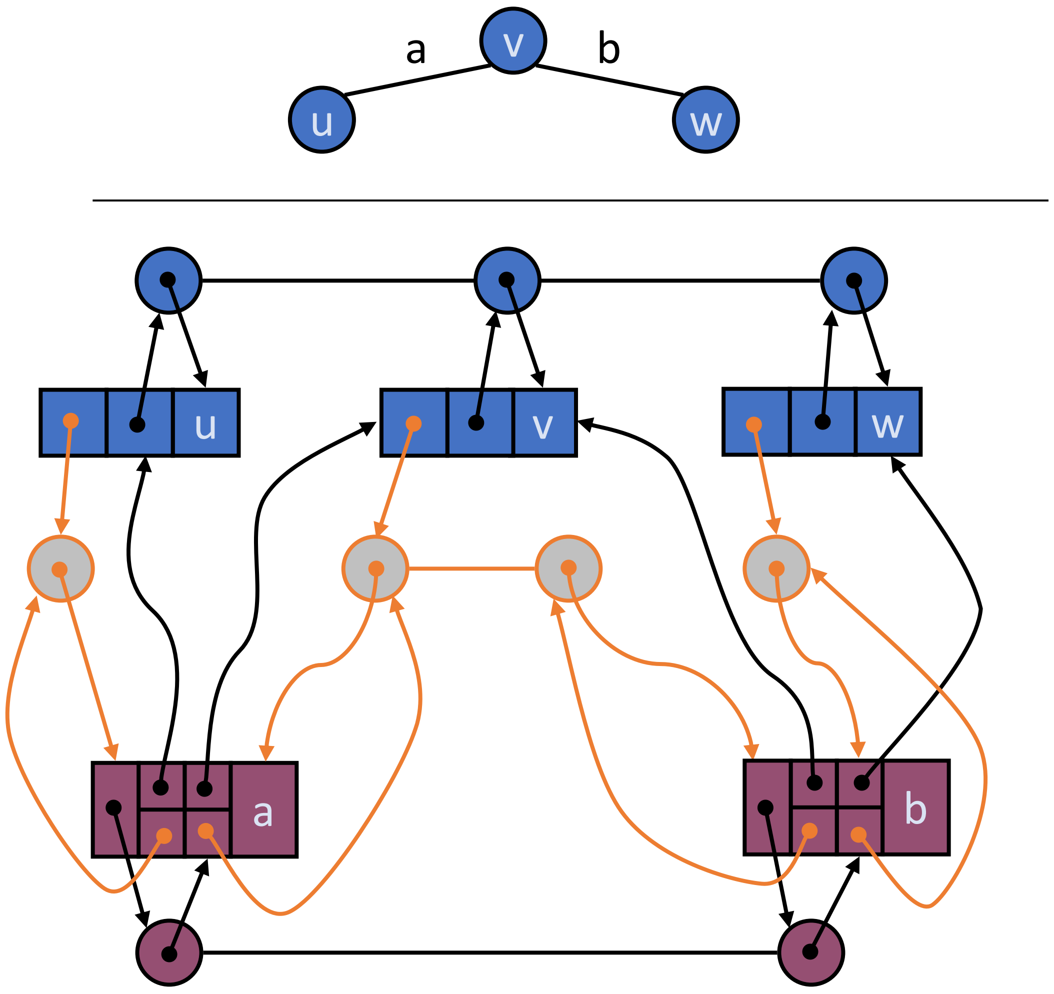




Augmenting edge information…..

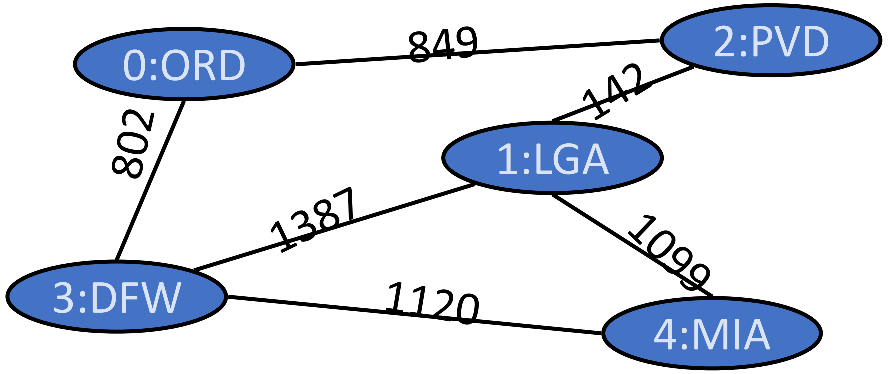
Adjacency List Structure

- Edge list structure
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



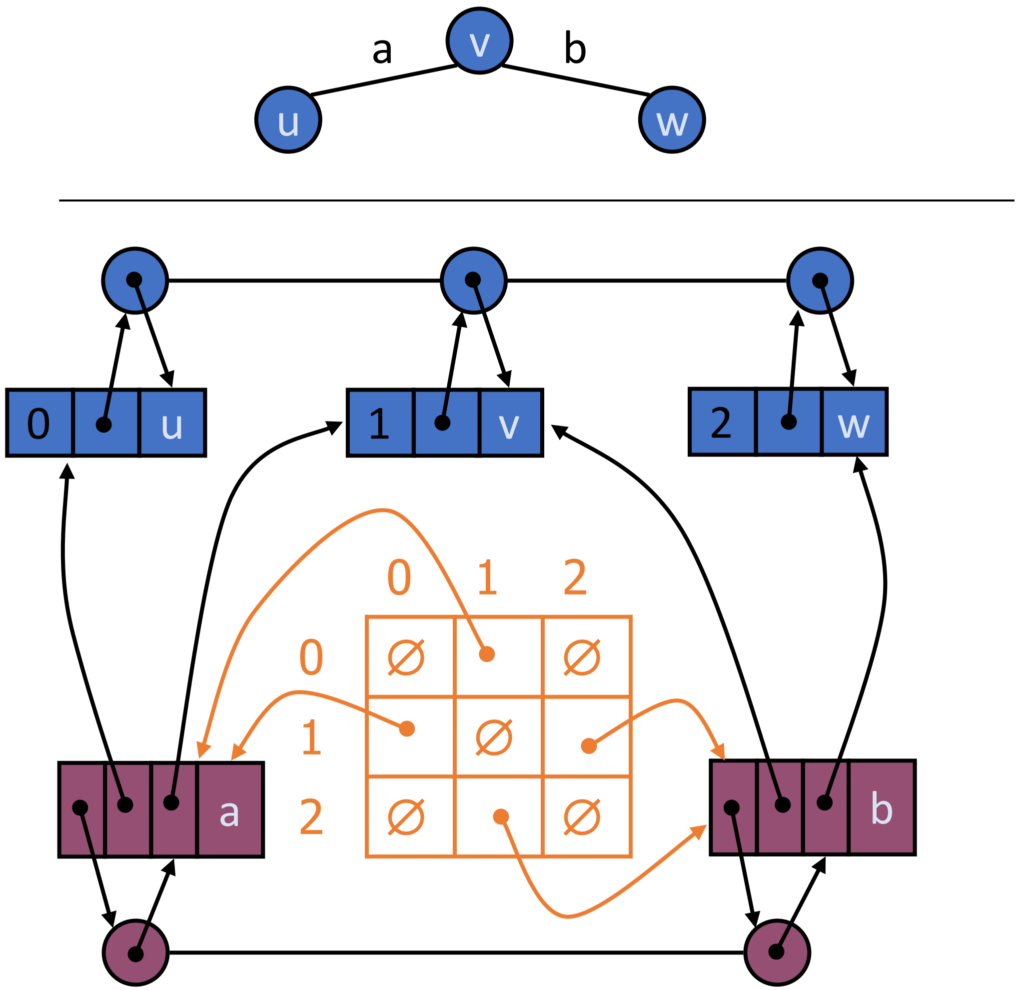
Adjacency Matrix Structure (Traditional)

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



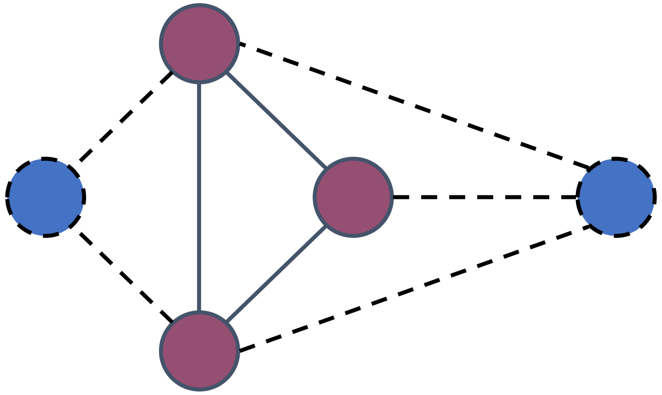
Adjacency Matrix Structure (Modern)

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge

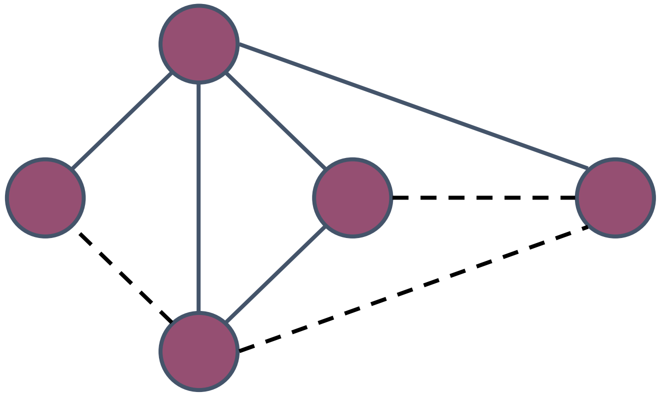


Recall: Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



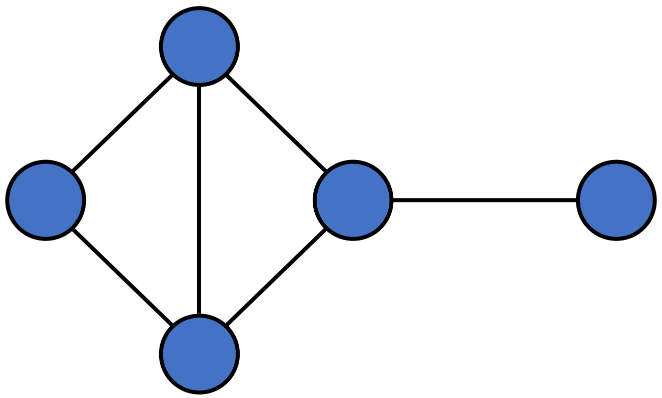
Subgraph



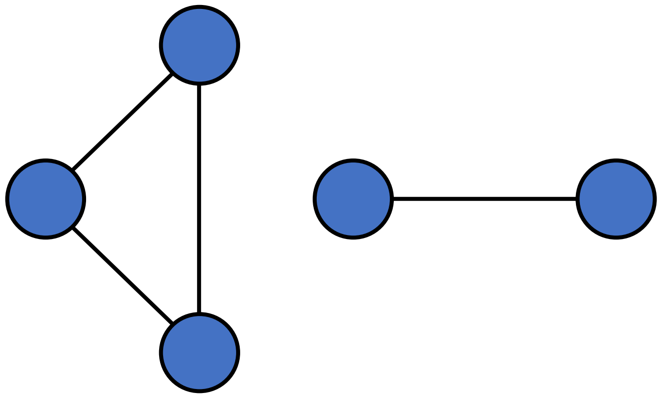
Spanning subgraph

Recall: Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



Connected graph

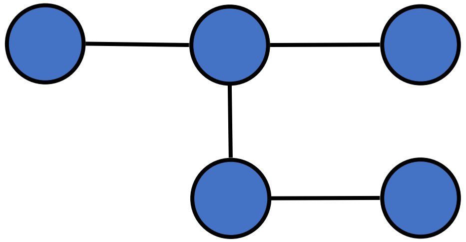


Non connected graph with two connected components

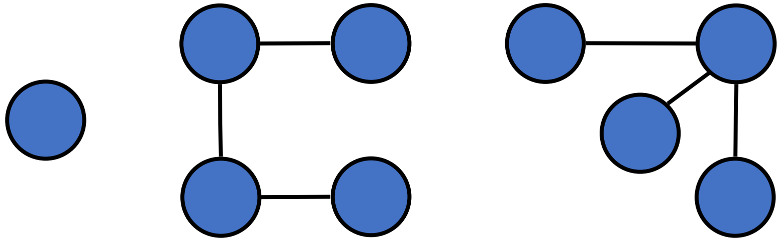
Recall: Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



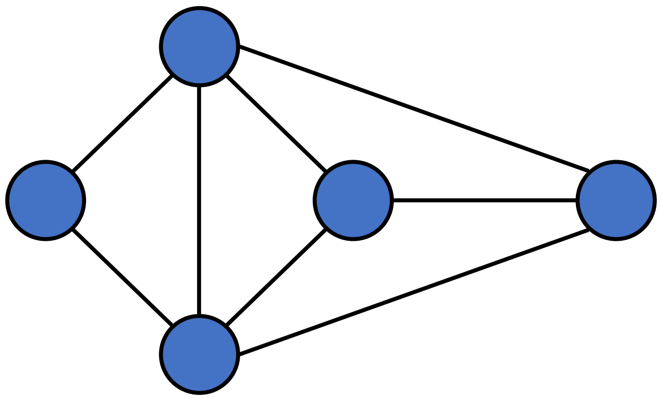
Tree



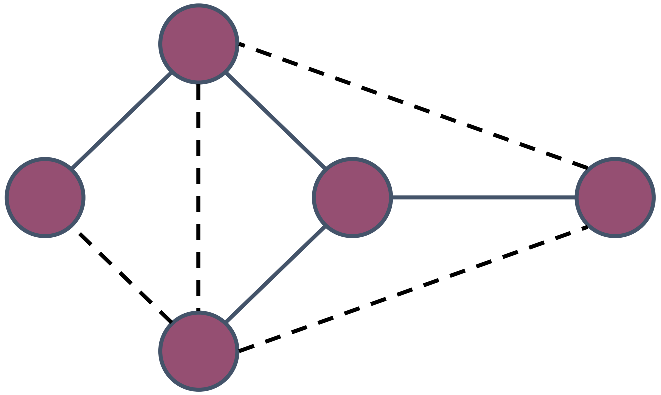
Forest

Recall: Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks



Graph



Spanning tree

Connectivity

Assuming m and n denote the number of edges and vertices, respectively.

- The range of m is zero to ${}^{n}C_{2}$
- In case of tree, $m=n-1$.
- If $m<n-1$, the graph is not connected.

Graph Searching Algorithms

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

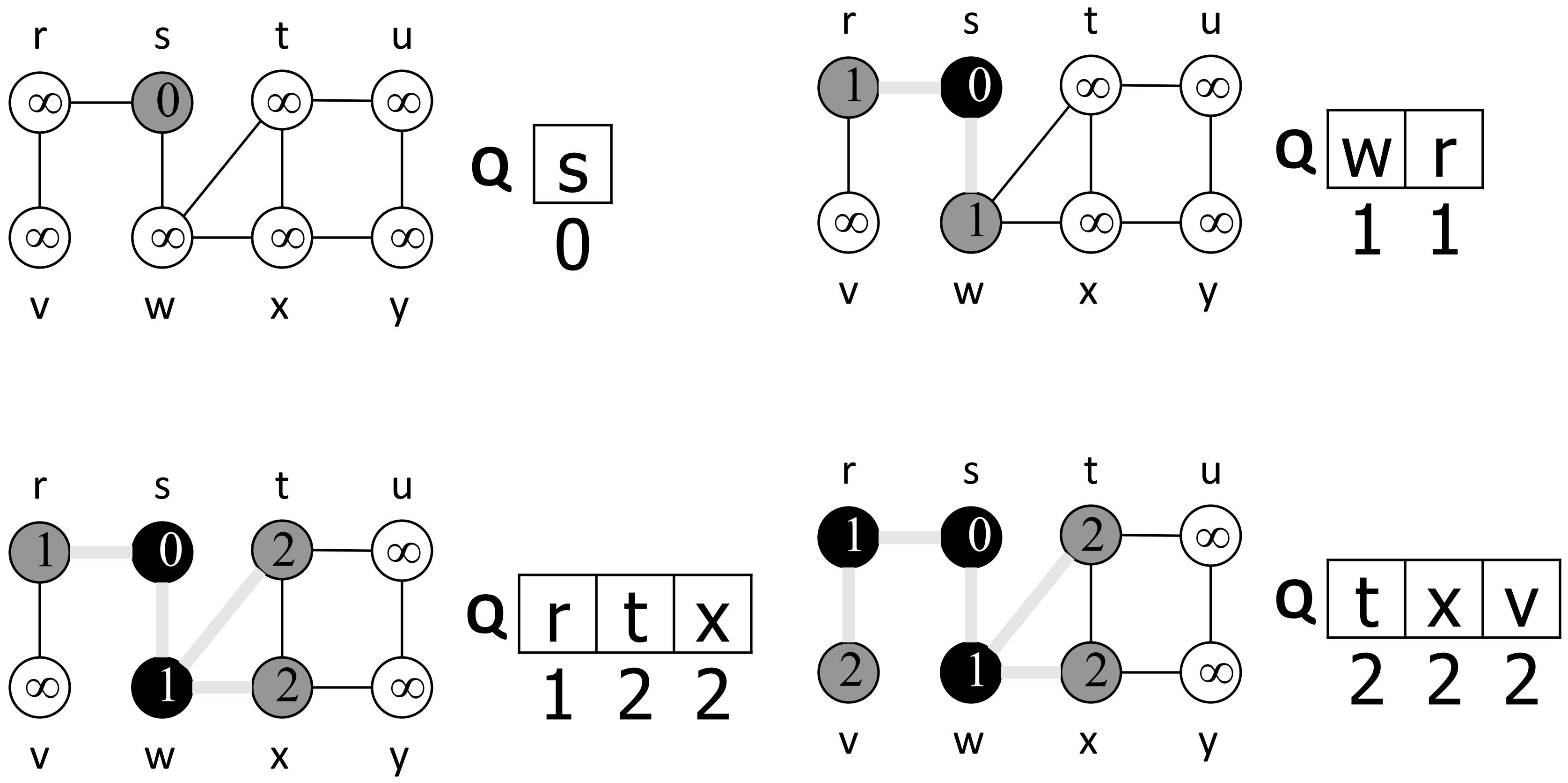
Breadth First Search

- 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

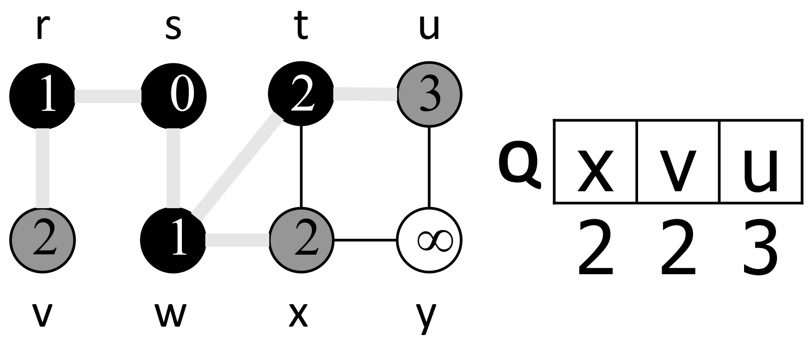
Breadth-First Search (2)

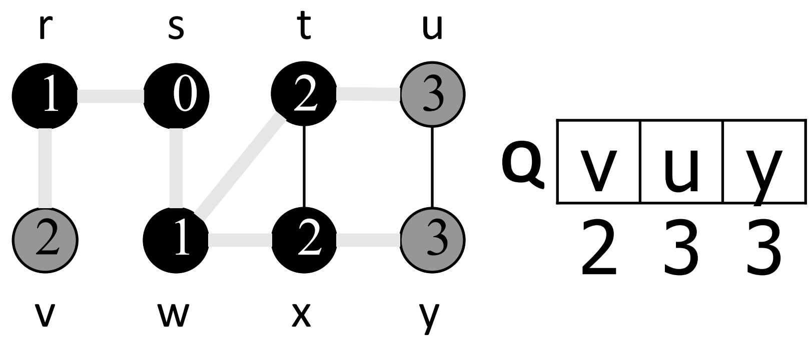
- In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and assigned a distance of 2
- This continues until every vertex has been assigned a level
- The label of any vertex *v* corresponds to the length of the shortest path (in terms of edges) from *s*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

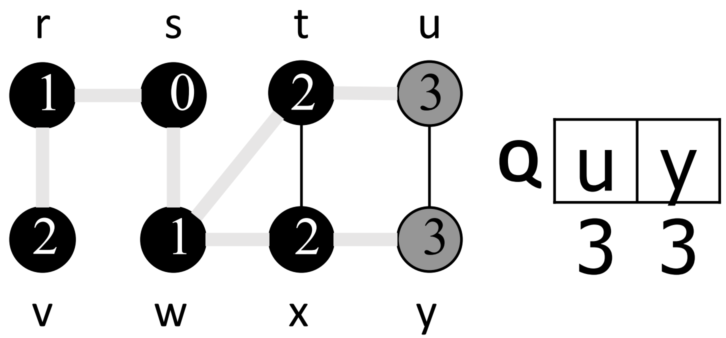
BFS Example

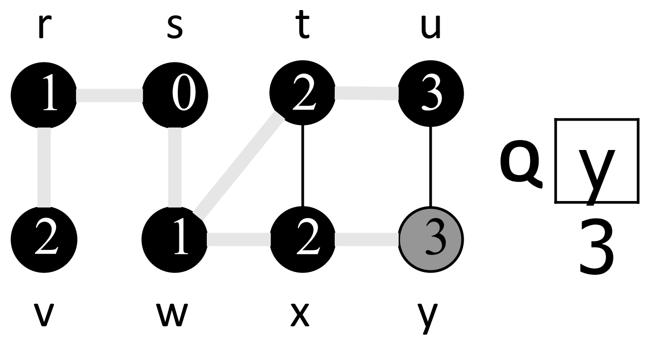


BFS Example

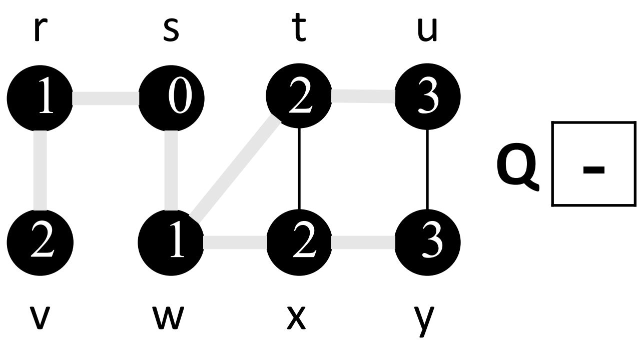

r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q
x v u
2 2 3


r s t u
1 0 2 3
2 1 2 3
v w x y
Q
v u y
2 3 3


r s t u
1 0 2 3
2 1 2 3
v w x y
Q
u y
3 3


r s t u
1 0 2 3
2 1 2 3
v w x y
Q
y
3

BFS Example: Result



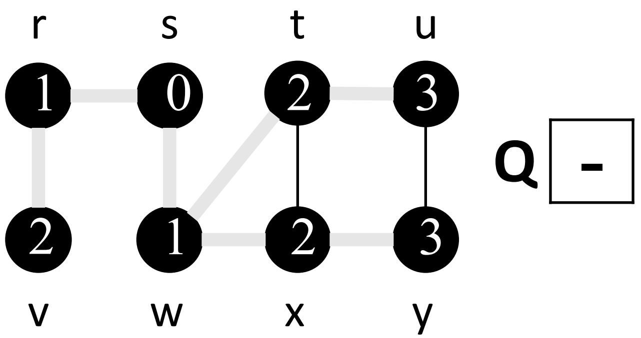
Breadth First Tree

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

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[u] ← u
            Enqueue(Q,v)
    Dequeue(Q)
    color[u] ← black
```

Init all vertices

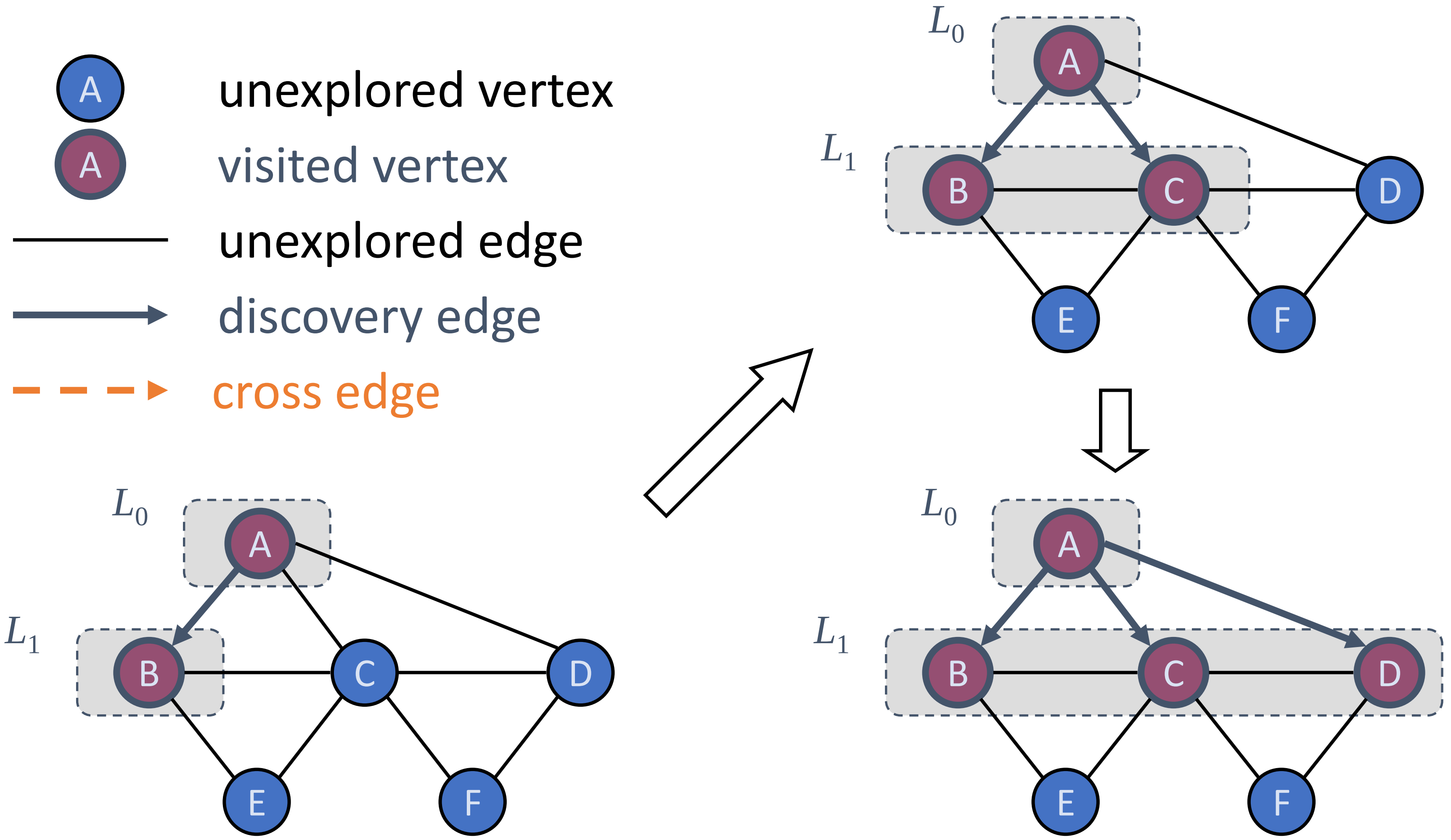
Init BFS with *s*

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

BFS Running Time

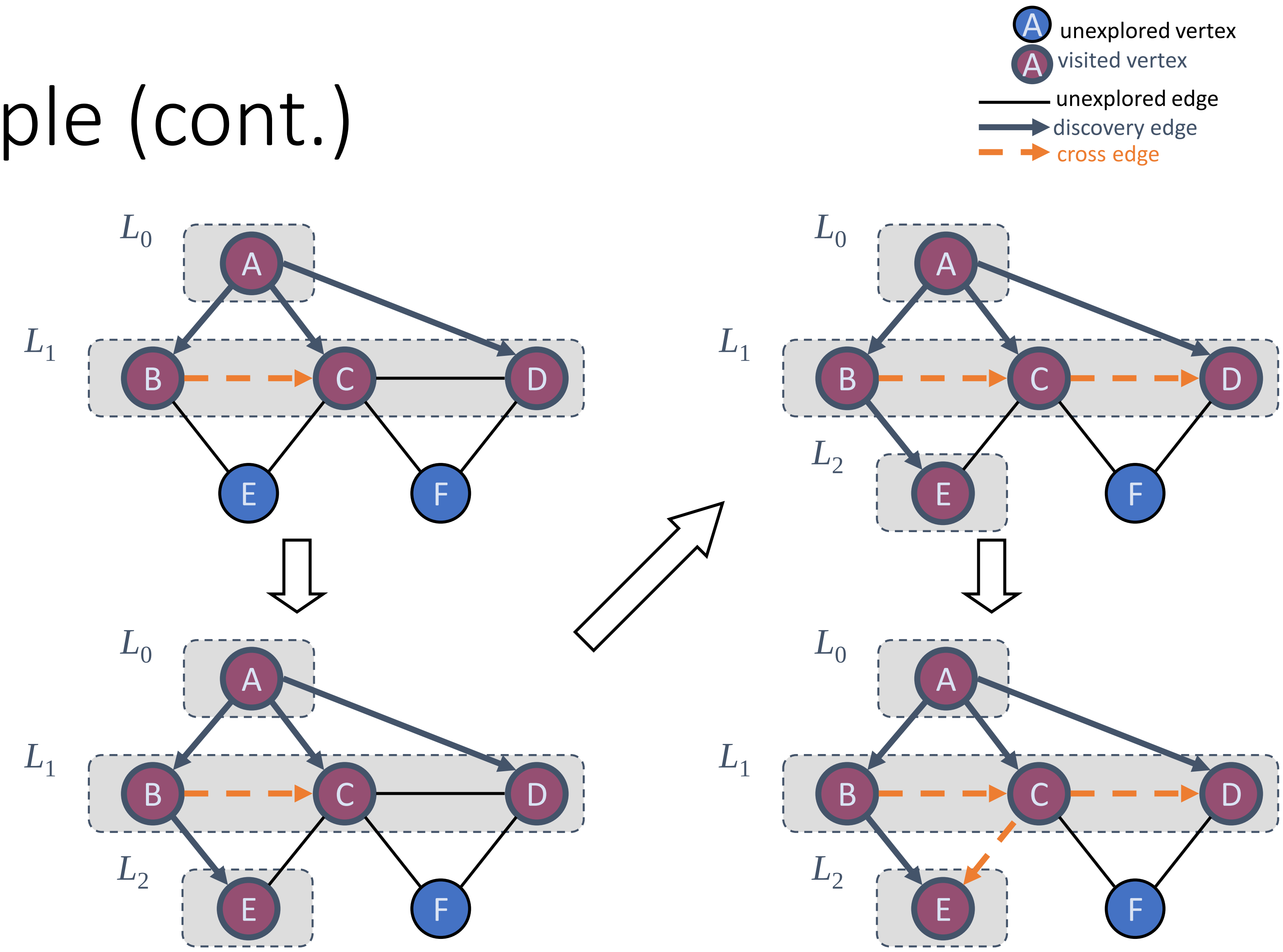
- Given a graph G = (V,E)
  - Vertices are enqueued if there color is white
  - Assuming that en- and dequeuing takes O(1) time the total cost of this operation is O(V)
  - Adjacency list of a vertex is scanned when the vertex is dequeued (and only then…)
  - The sum of the lengths of all lists is Θ(E). 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)

Example

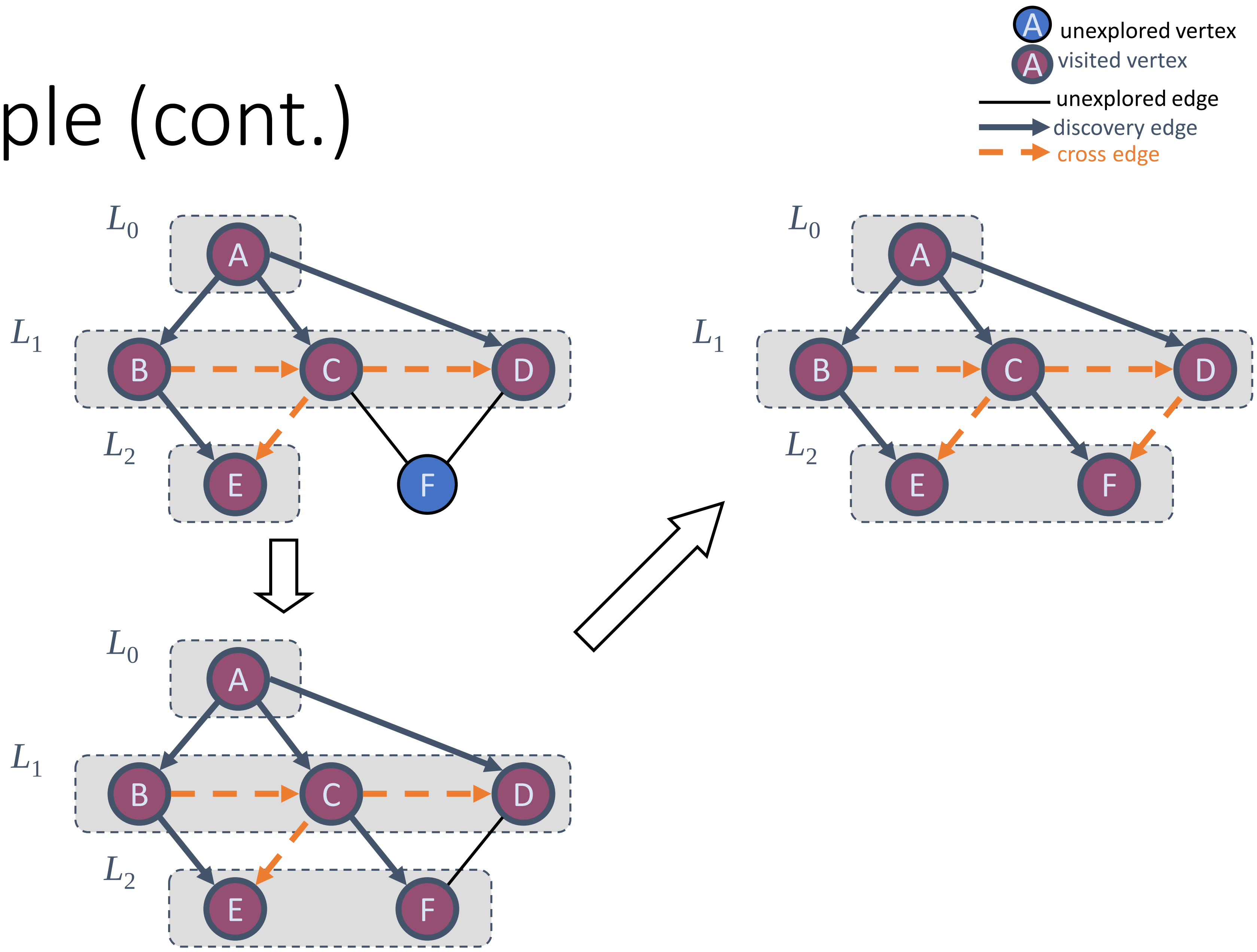


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F

Example (cont.)



Example (cont.)



BFS Properties

All the edges go between adjacent levels or same levels. There cannot be level skipping. Why? If it happen, the node will be previously discovered.

BFS Properties

It computes the **shortest distance of s** to all reachable vertices. The **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 Properties

The level numbers should be unique indicating the length of shortest path. There can be multiple paths to reach but length will be unique.

BFS Properties

Given a graph G = (V,E), BFS **discovers all vertices reachable from a source vertex** ***s. How it can be used to find connected components?***

BFS Properties

It can be used to compute the cycles in the undirected graph. How?

BFS Properties

Notation

G_s: connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

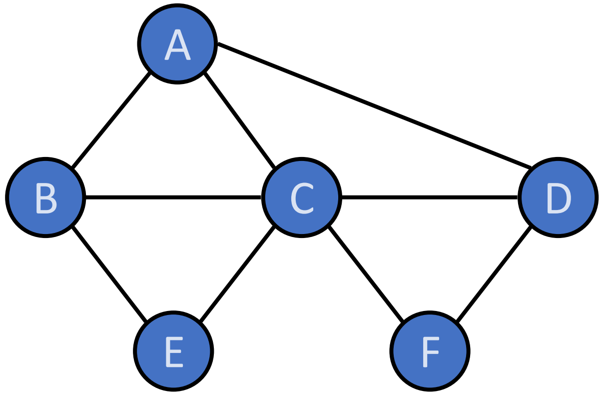
Property 2

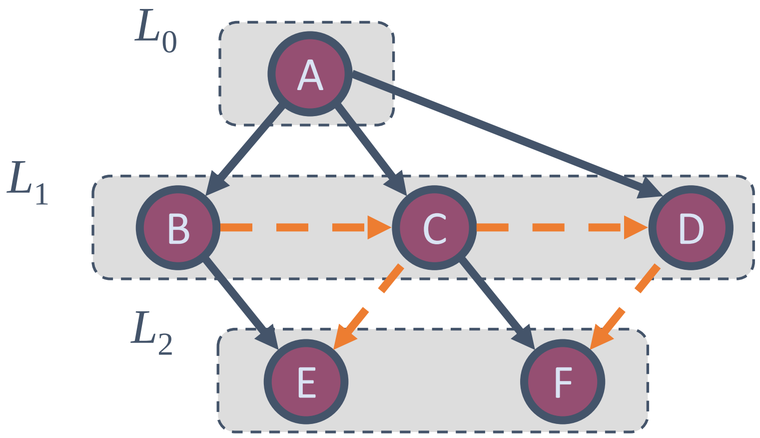
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

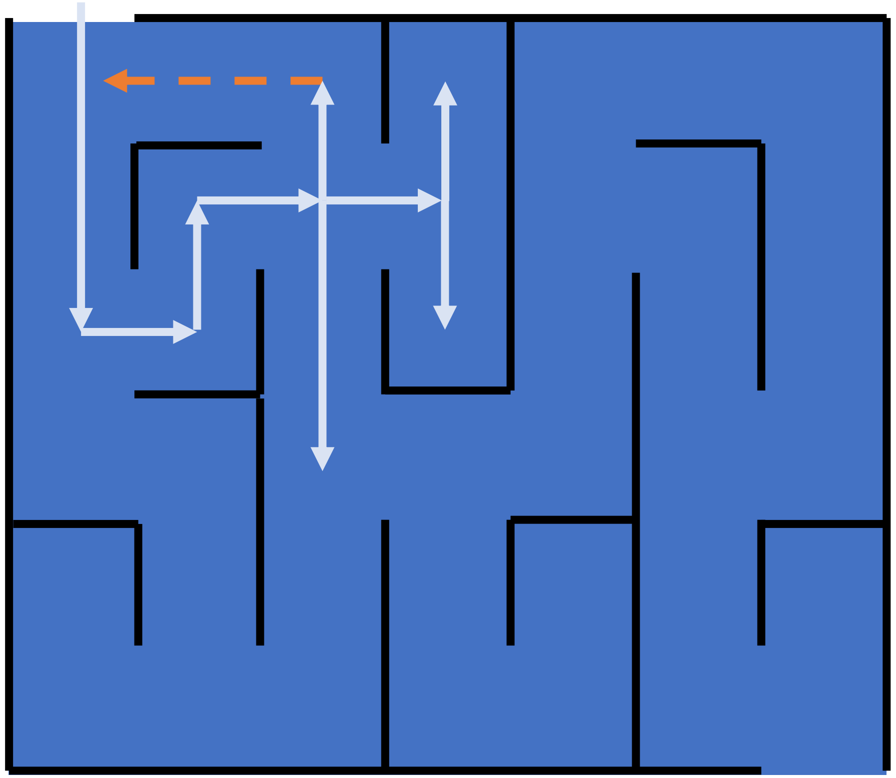
- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges





DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



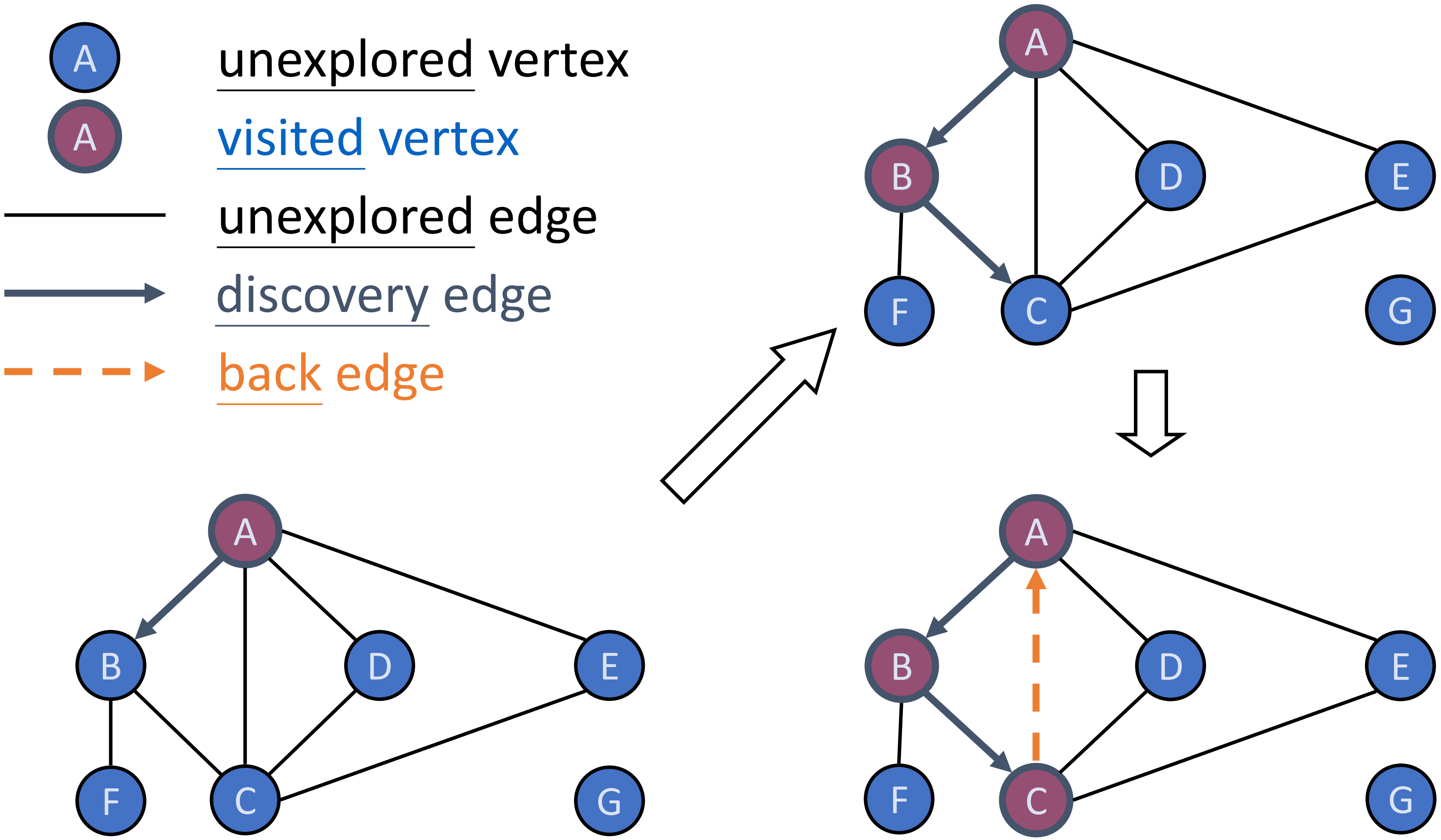
Depth-First Search

- **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

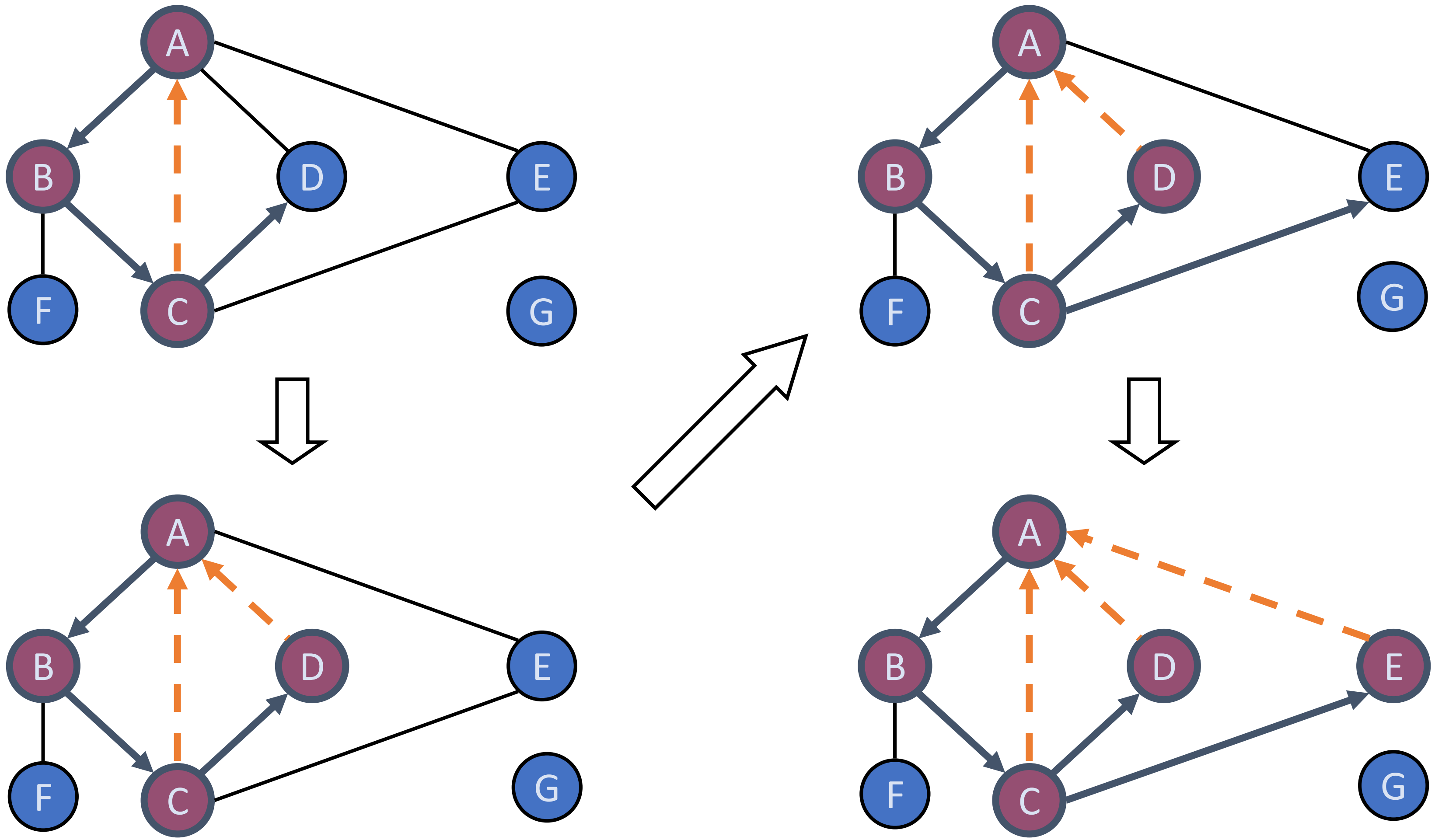
Depth-First Search (2)

- 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

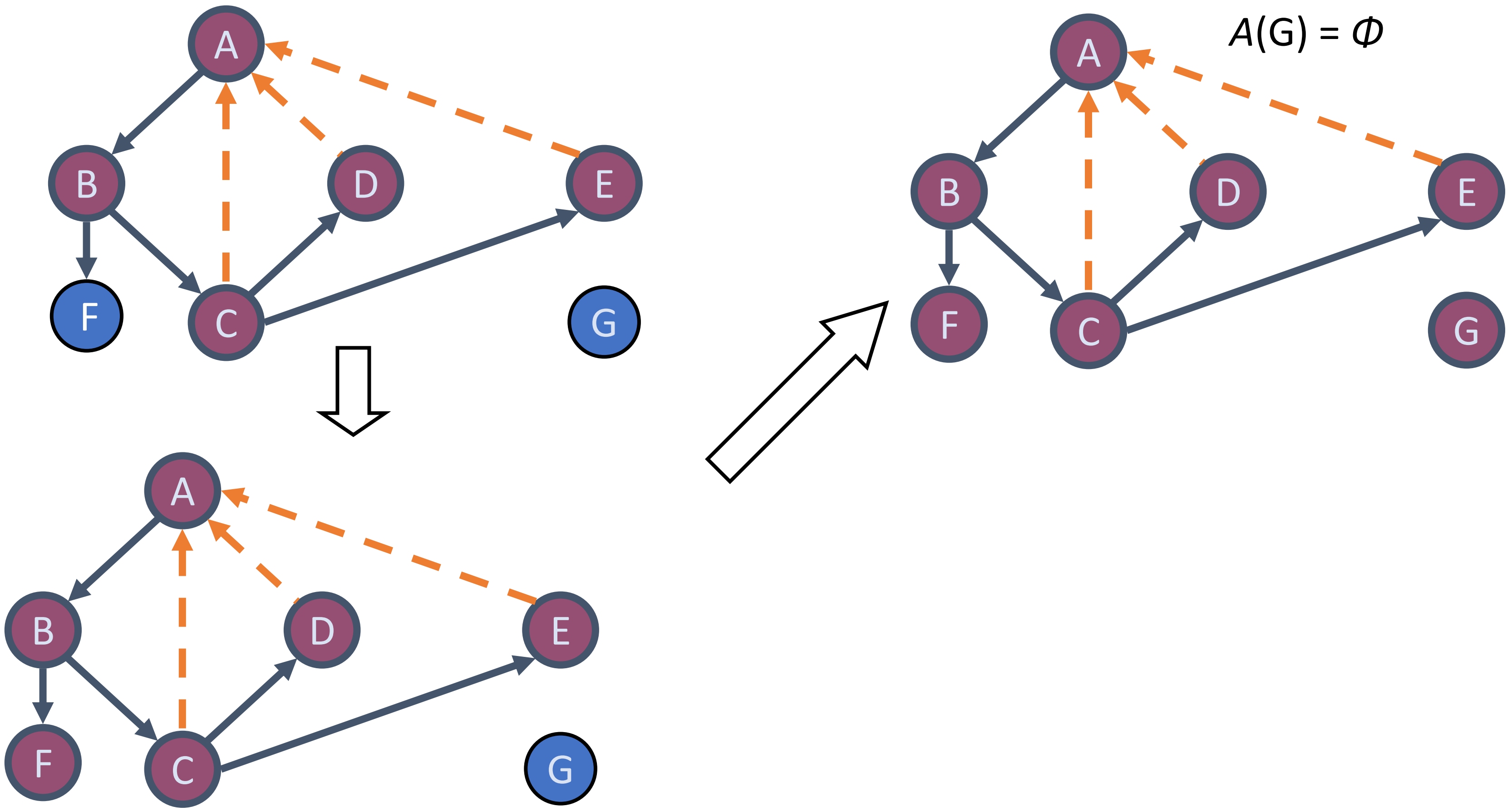
Example 1


A
A
unexplored vertex
visited vertex
unexplored edge
discovery edge
back edge
A
B
C
D
E
F
G
A
B
C
D
E
F
G
A
B
C
D
E
F
G

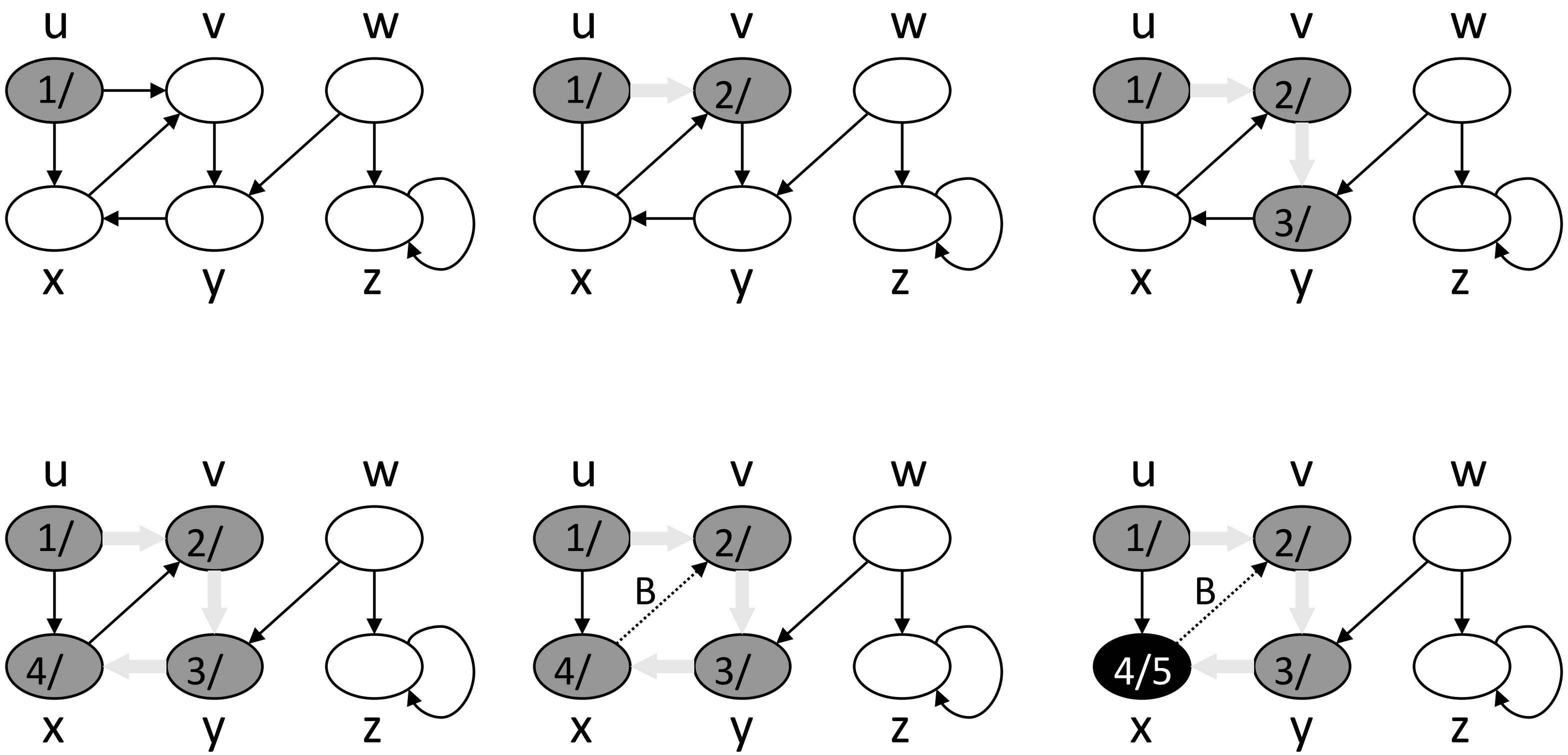
Example:1 (cont.)



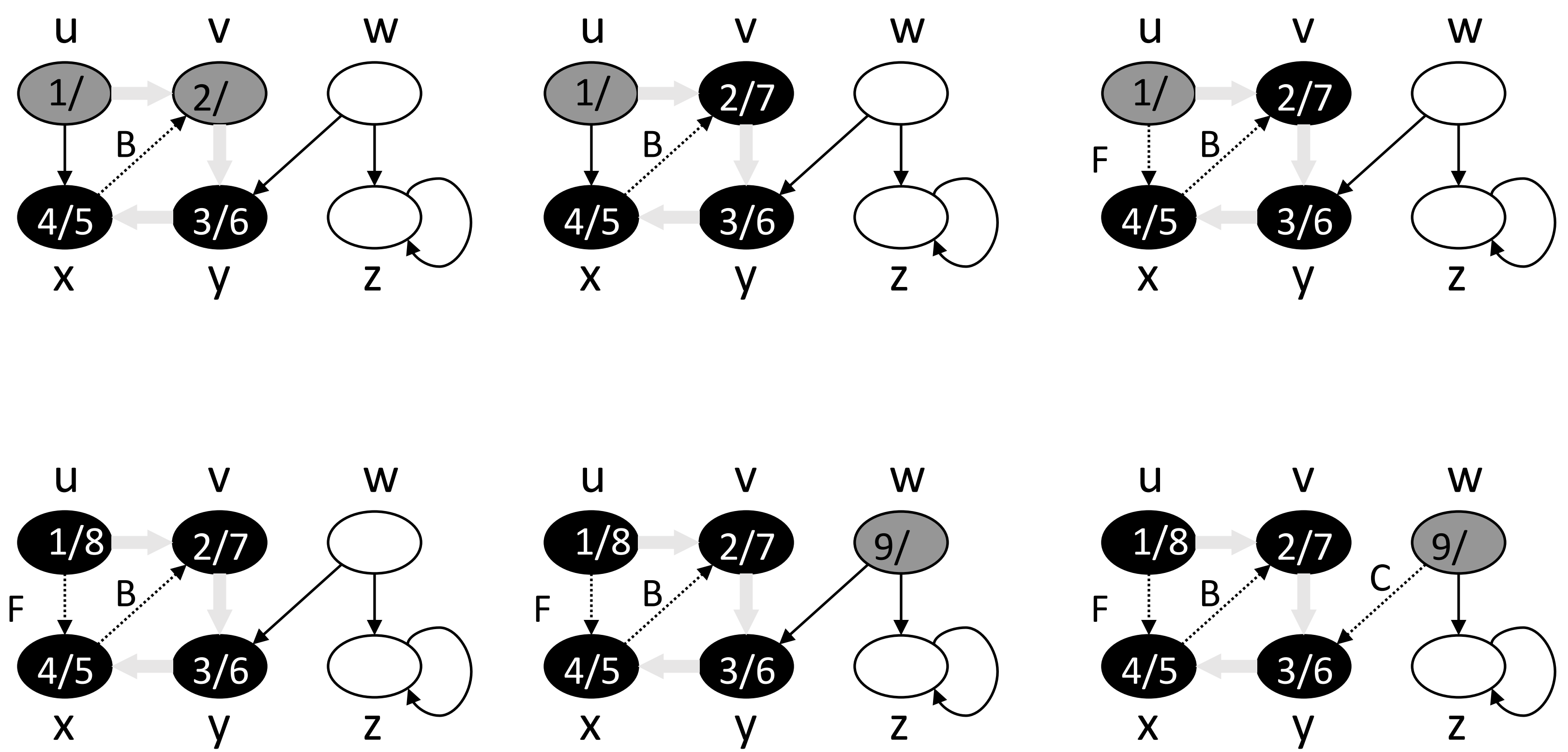
Example: 1 (cont.)



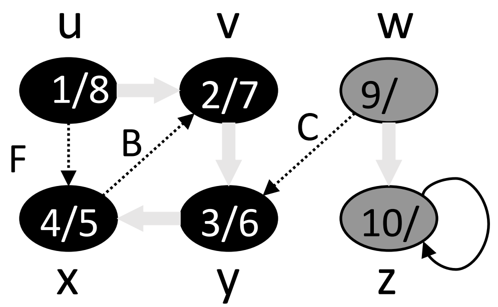
DFS Example using timestamping for directed graph:2 (cont.)

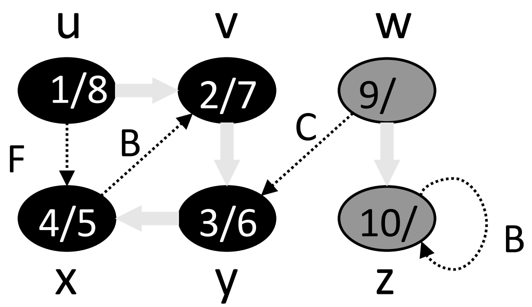


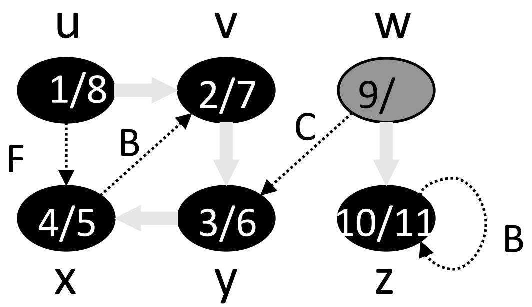
DFS Example:2 (cont.)

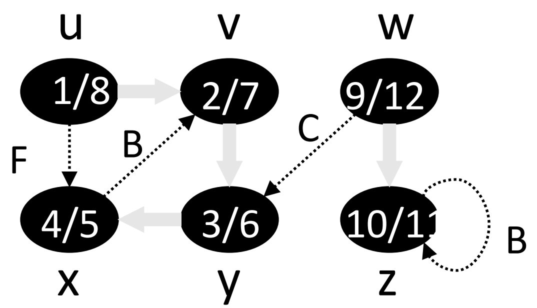


DFS Example:2 (cont.)









Predecessor Subgraph

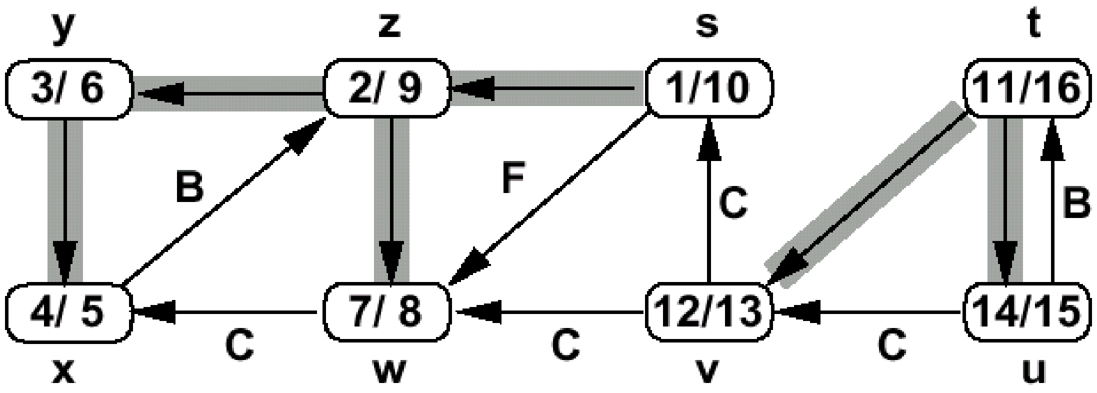
- Define slightly different from BFS

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

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

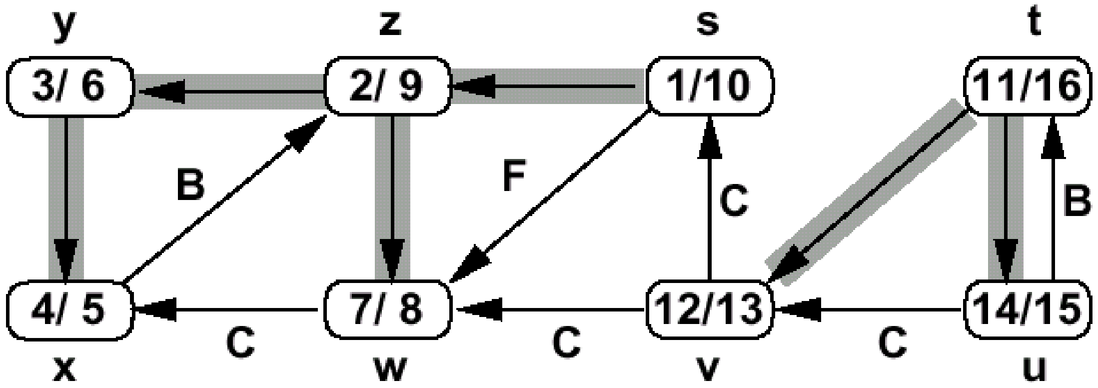
DFS Edge Classification

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



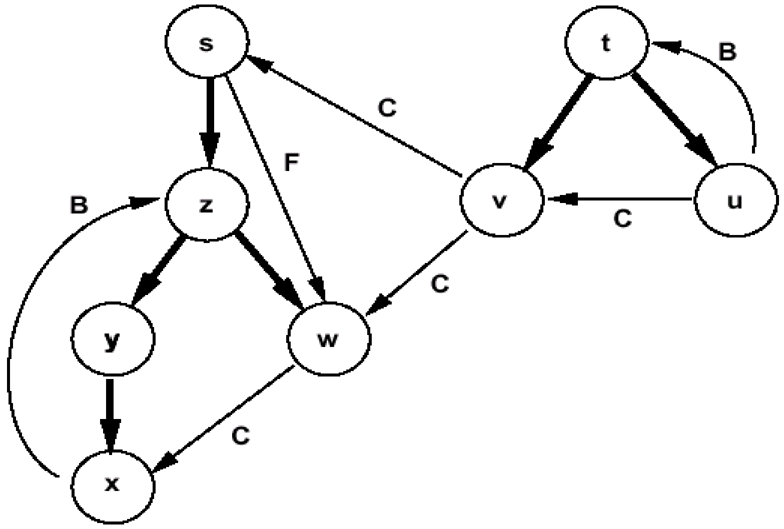
DFS Edge Classification (2)

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



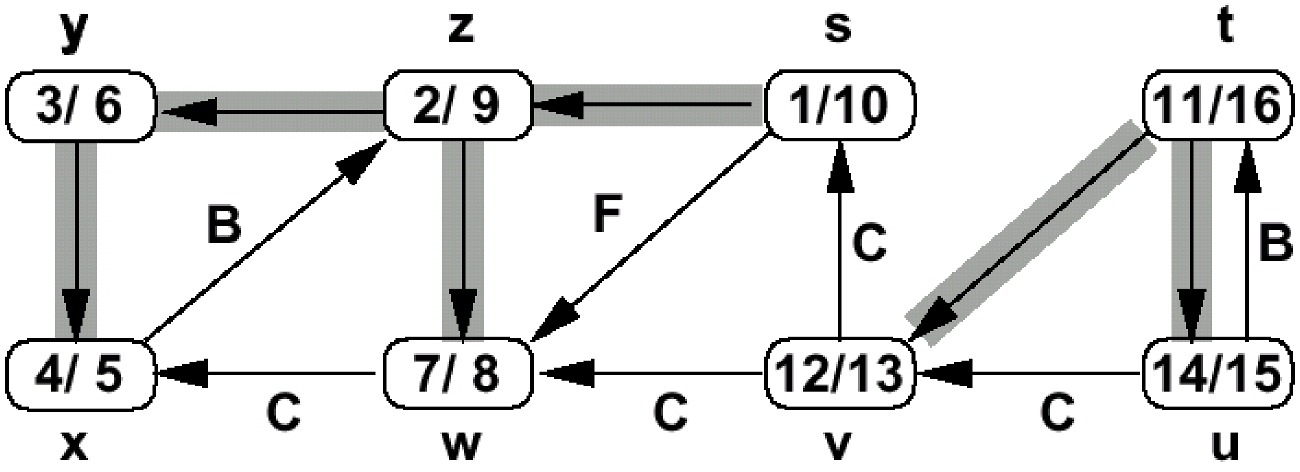
DFS Edge Classification (3)

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



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 Algorithm

```
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

DFS Algorithm (2)

- 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)

DFS Algorithm (3)

- 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 Θ(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 Θ(E)
  - **the running time of DFS is Θ(V+E)**

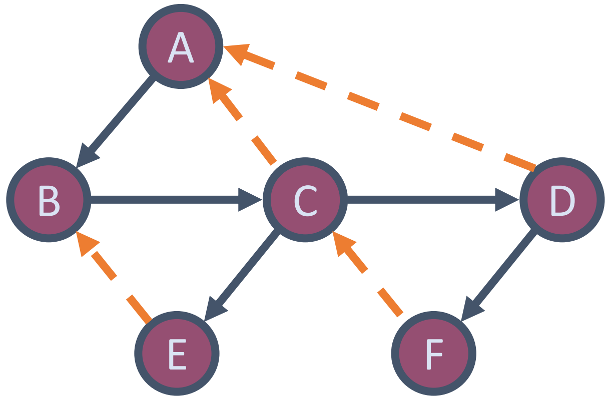
$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

Path Finding using DFS

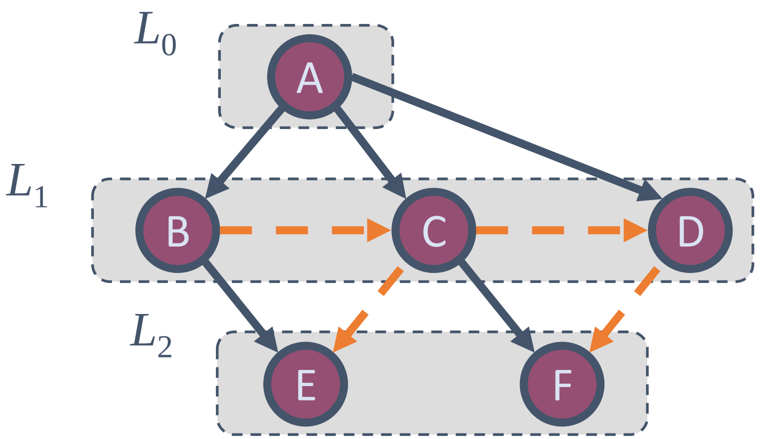
- We can specialize the DFS algorithm to find a path between two given vertices $\boldsymbol{v}$ and $\boldsymbol{z}$ using the template method pattern
- We call $\boldsymbol{DFS}(\boldsymbol{G}, \boldsymbol{v})$ with $\boldsymbol{v}$ as the start vertex
- We use a stack $\boldsymbol{S}$ to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex $\boldsymbol{z}$ is encountered, we return the path as the contents of the stack

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√



DFS

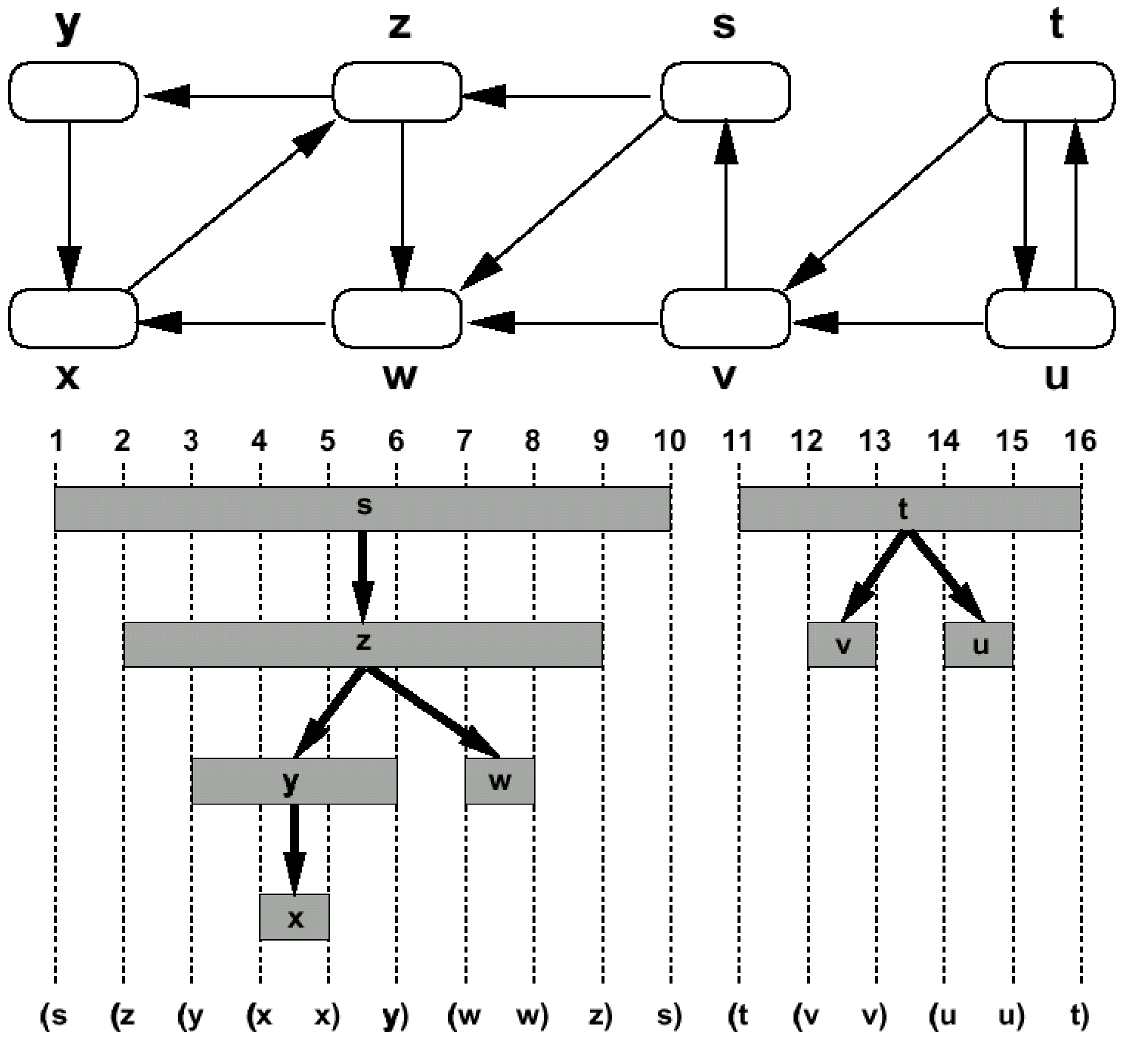


BFS

DFS Parenthesis Theorem

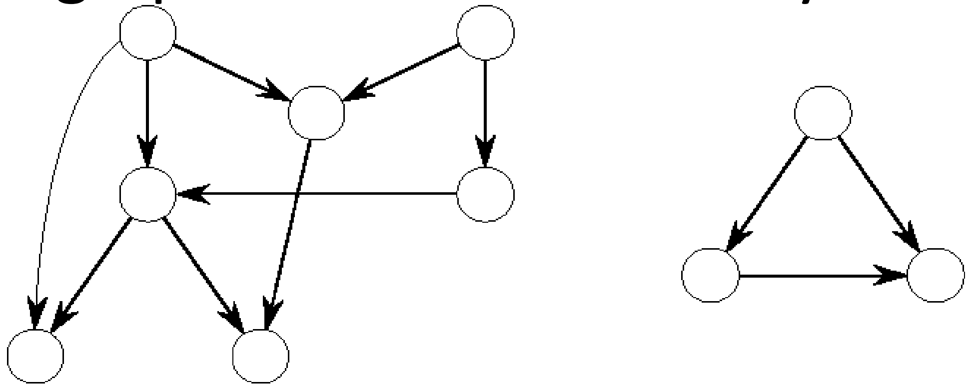
- Discovery and finish times have parenthesis structure
  - represent discovery of *u* with left parenthesis "(u"
  - represent finishin of *u* with right parenthesis "u)"
  - history of discoveries and finishings makes a well-formed expression (parenthesis are properly nested)
- Intuition for proof: 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)



Directed Acyclic Graphs

- A DAG is a directed graph with no directed cycles



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

DAG Theorem

- A directed graph *G* is acyclic iff 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

Topological Sort

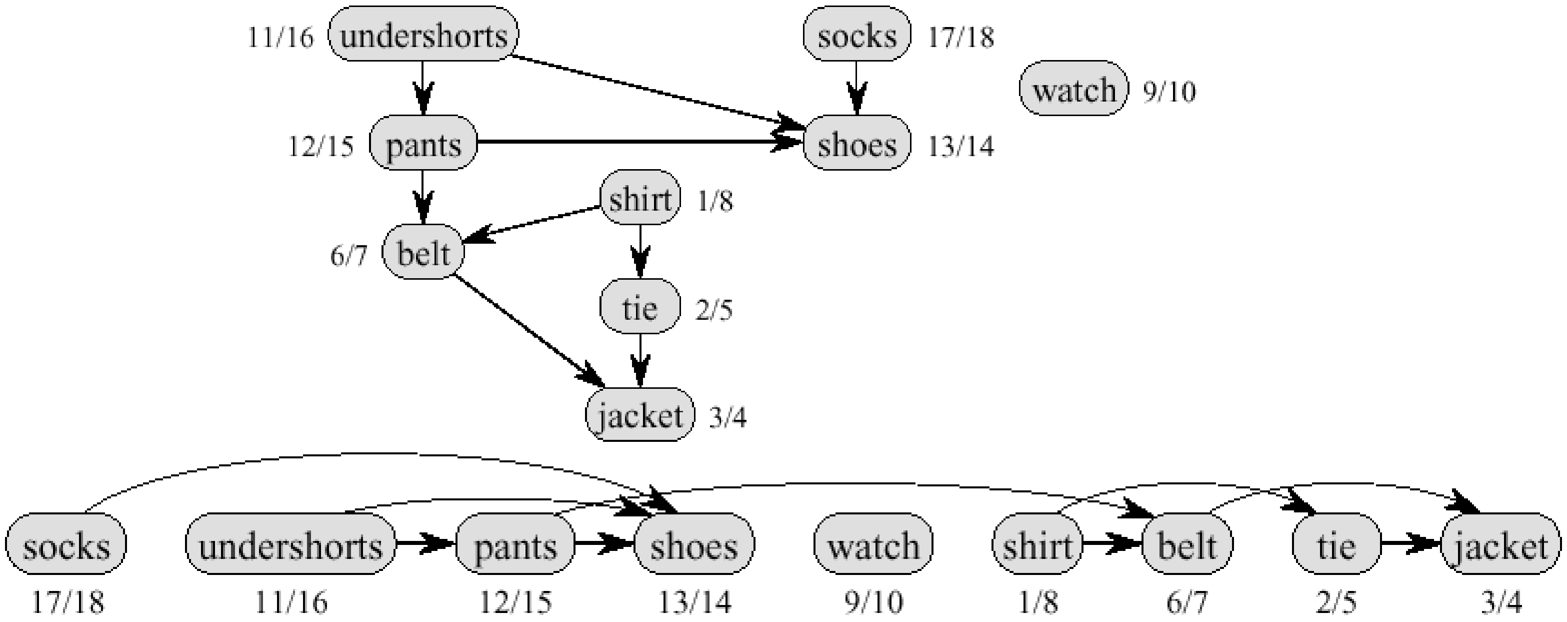
- Sorting of a directed acyclic graph (DAG)
- A topological sort of a DAG is a linear ordering of all its vertices such that for any edge (u,v) in the DAG, u appears before v in the ordering
- The following algorithm topologically sorts a DAG

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 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

- Running time
  - depth-first search: $O(V+E)$ time
  - insert each of the $|V|$ vertices onto 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