

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Data structures and Algorithms

Lecture 7: Graphs [GT 13.1-3]

A/Prof Julian Mestre
School of Computer Science

*Some content is taken from material
provided by the textbook publisher Wiley.*



THE UNIVERSITY OF
SYDNEY



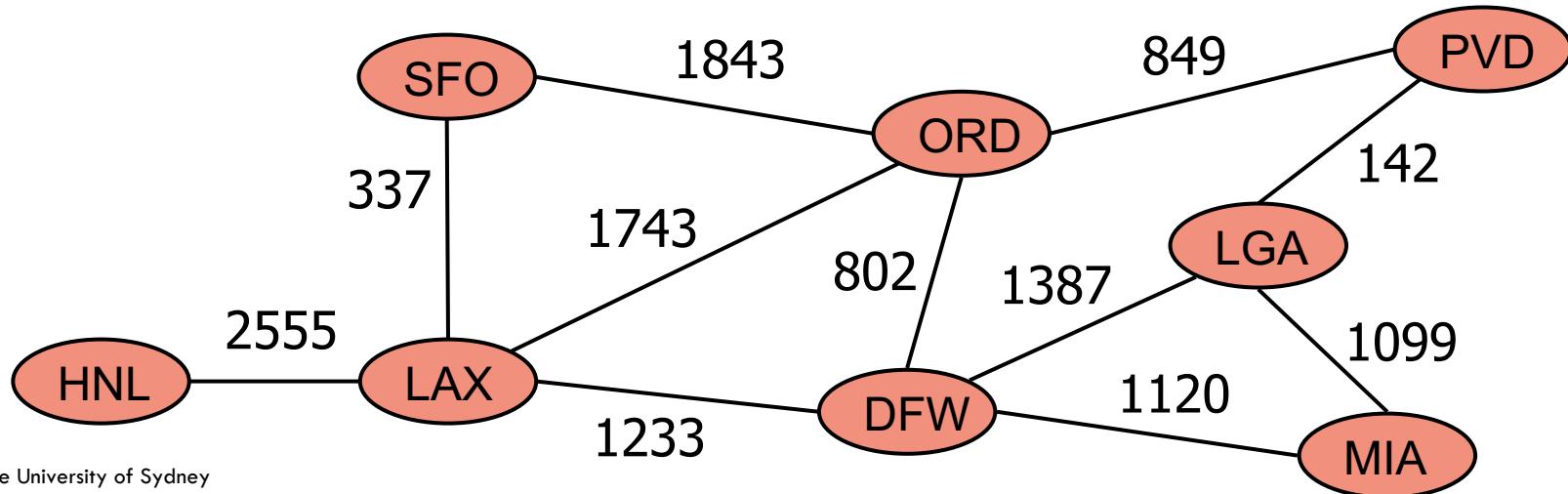
Graphs

A graph **G** is a pair (V, E) , where

- V is a set of nodes, called **vertices**
- E is a collection of pairs of vertices, called **edges**

Example:

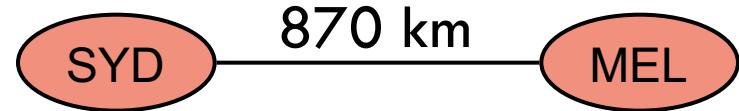
- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route



Edge Types

Directed edge

- ordered pair of vertices (u, v)
- u is the origin/tail
- v is the destination/head
- e.g., a flight



Undirected edge

- unordered pair of vertices (u, v)
- e.g., a two-way road

Applications

Electronic circuits

- Printed circuit board
- Integrated circuit

Transportation networks

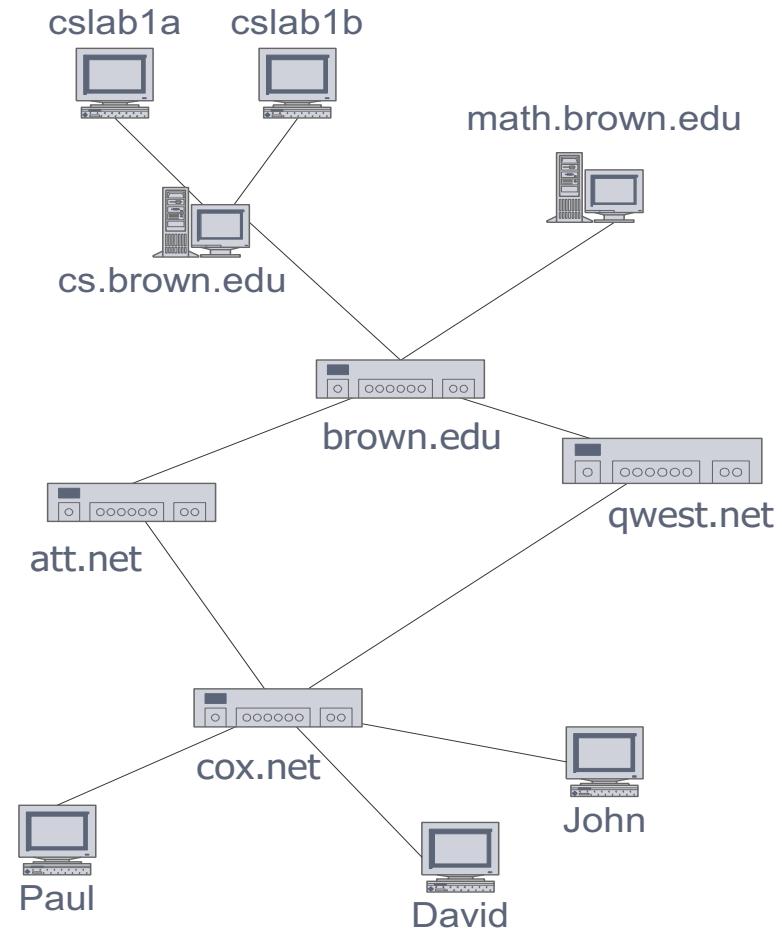
- Highway network
- Flight network

Computer networks

- Internet
- Web

Modeling

- Entity-relationship diagram
- Gantt precedence constraints



Graph concepts: Paths

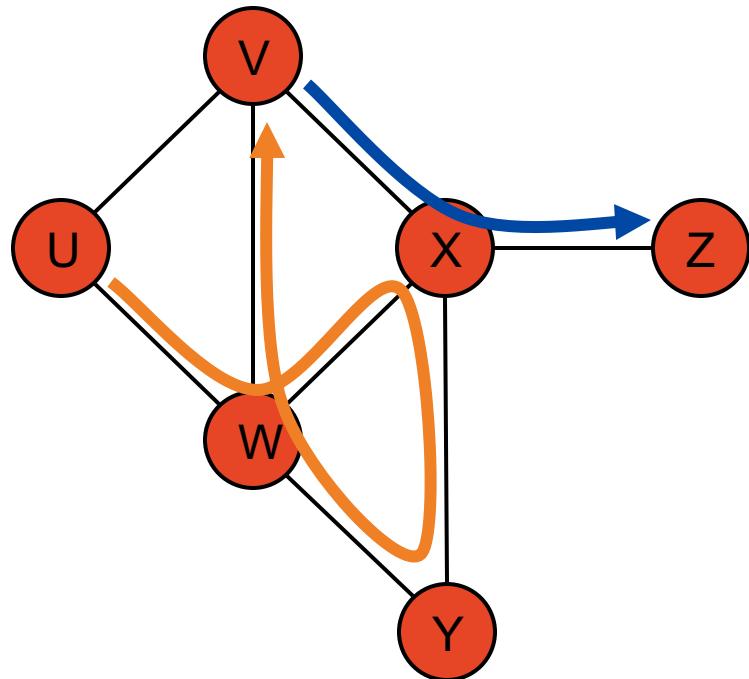
A **path** is a sequence of vertices such that every pair of consecutive vertices is connected by an edge.

A simple path is one where all vertices are distinct

Examples

- (V, X, Z) is a simple path
- (U, W, X, Y, W, V) is a path that is not simple

A (simple) path from s to t is also called an s - t path.



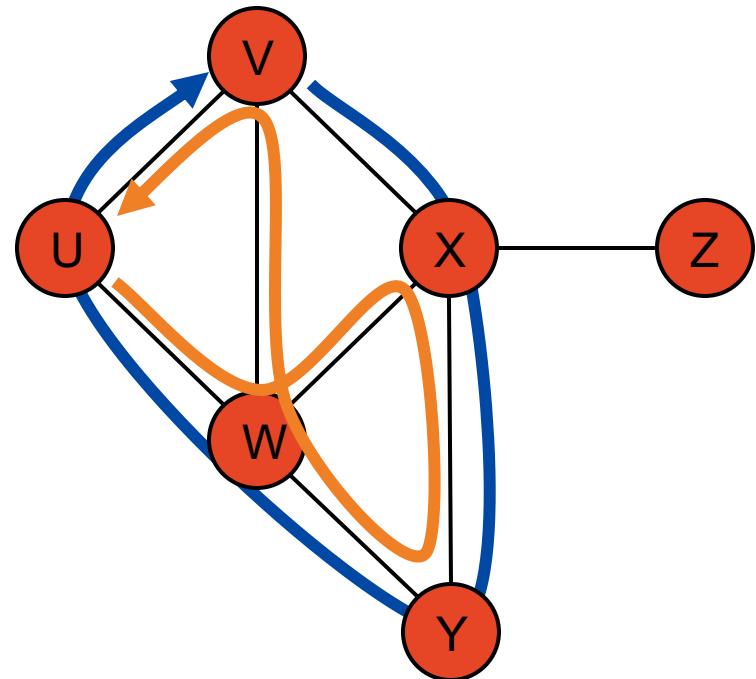
Graph concepts: Cycles

A **cycle** is defined by a path that starts and ends at the same vertex

A **simple cycle** is one where all vertices are distinct

Examples

- (V, X, Y, W, U, V) is a simple cycle
- (U, W, X, Y, W, V, U) is a cycle that is not simple



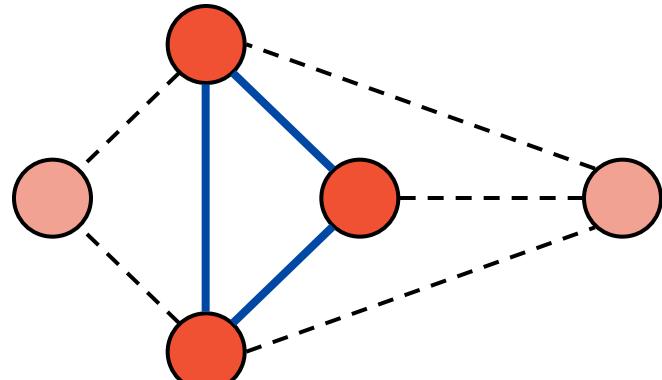
An **acyclic graph** has no cycles

Graph concepts: Subgraphs

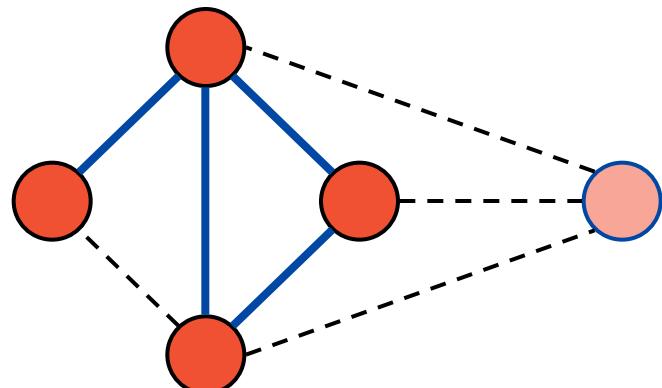
Let $G=(V, E)$ be a graph. We say $S=(U, F)$ is a subgraph of G if $U \subseteq V$ and $F \subseteq E$

A subset $U \subseteq V$ induces a graph $G[U] = (U, E[U])$ where $E[U]$ are the edges in E with endpoints in U

A subset $F \subseteq E$ induces a graph $G[F] = (V[F], F)$ where $V[F]$ are the endpoints of edges in F



Subgraph induced by red vertices

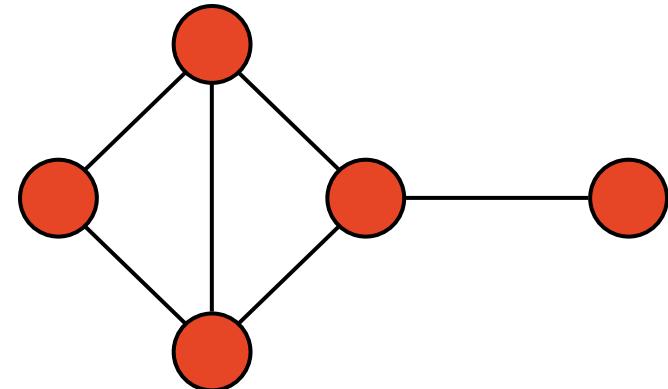


Subgraph induced by blue edges

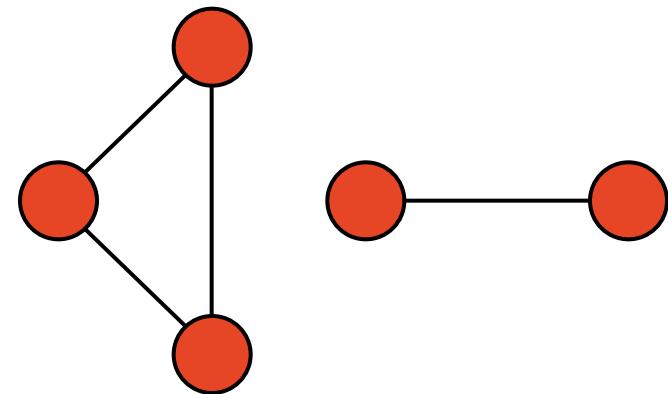
Graph concepts: Connectivity

A graph $G=(V, E)$ is connected if there is a path between every pair of vertices in V

A connected component of a graph G is a maximal connected subgraph of G



Connected graph



Graph with two connected components

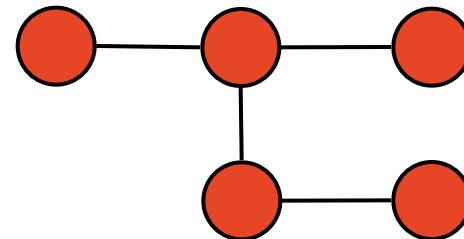
Graph concepts: Trees and Forests

An unrooted tree T is a graph such that

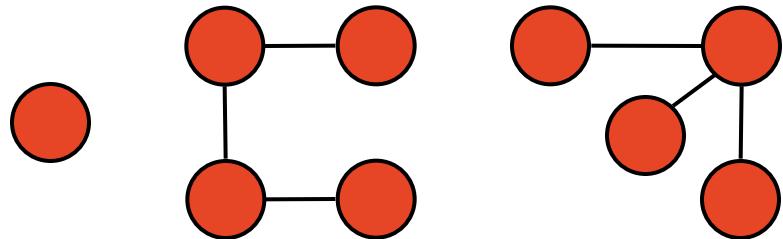
- T is connected
- T has no cycles

A forest is a graph without cycles. In other words, its connected components are trees

Fact: Every tree on n vertices has $n-1$ edges



Tree



Forest

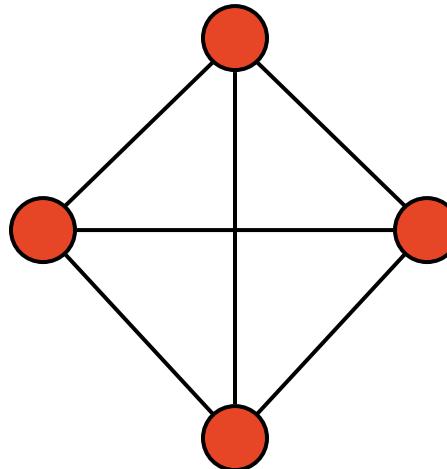
Graph Properties

Fact: $\sum_{v \in V} \deg(v) = 2m$

Fact: In a simple undirected graph $m \leq n(n - 1)/2$

Notation

n	number of vertices
m	number of edges
Δ	maximum degree



Example: K_4

$$n = 4$$

$$m = 6$$

$$\max \deg = 3$$

Graph ADT

We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.

A Vertex stores an associated object (e.g., an airport code) that is retrieved with a getElement() method.

An Edge stores an associated object (e.g., a flight number, travel distance) that is retrieved with a getElement() method.

Directed Graph ADT

Undirected
Graph
alternatives

degree(v)



incidentEdges(v)

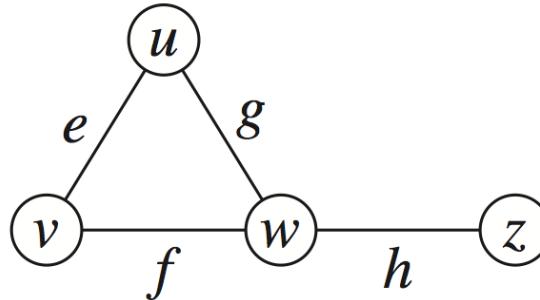


- numVertices():** Returns the number of vertices of the graph.
- vertices():** Returns an iteration of all the vertices of the graph.
- numEdges():** Returns the number of edges of the graph.
- edges():** Returns an iteration of all the edges of the graph.
- getEdge(u, v):** Returns the edge from vertex u to vertex v , if one exists; otherwise return null. For an undirected graph, there is no difference between $\text{getEdge}(u, v)$ and $\text{getEdge}(v, u)$.
- endVertices(e):** Returns an array containing the two endpoint vertices of edge e . If the graph is directed, the first vertex is the origin and the second is the destination.
- opposite(v, e):** For edge e incident to vertex v , returns the other vertex of the edge; an error occurs if e is not incident to v .
- outDegree(v):** Returns the number of outgoing edges from vertex v .
- inDegree(v):** Returns the number of incoming edges to vertex v . For an undirected graph, this returns the same value as does $\text{outDegree}(v)$.
- outgoingEdges(v):** Returns an iteration of all outgoing edges from vertex v .
- incomingEdges(v):** Returns an iteration of all incoming edges to vertex v . For an undirected graph, this returns the same collection as does $\text{outgoingEdges}(v)$.
- insertVertex(x):** Creates and returns a new Vertex storing element x .
- insertEdge(u, v, x):** Creates and returns a new Edge from vertex u to vertex v , storing element x ; an error occurs if there already exists an edge from u to v .
- removeVertex(v):** Removes vertex v and all its incident edges from the graph.
- removeEdge(e):** Removes edge e from the graph.

Edge List Structure

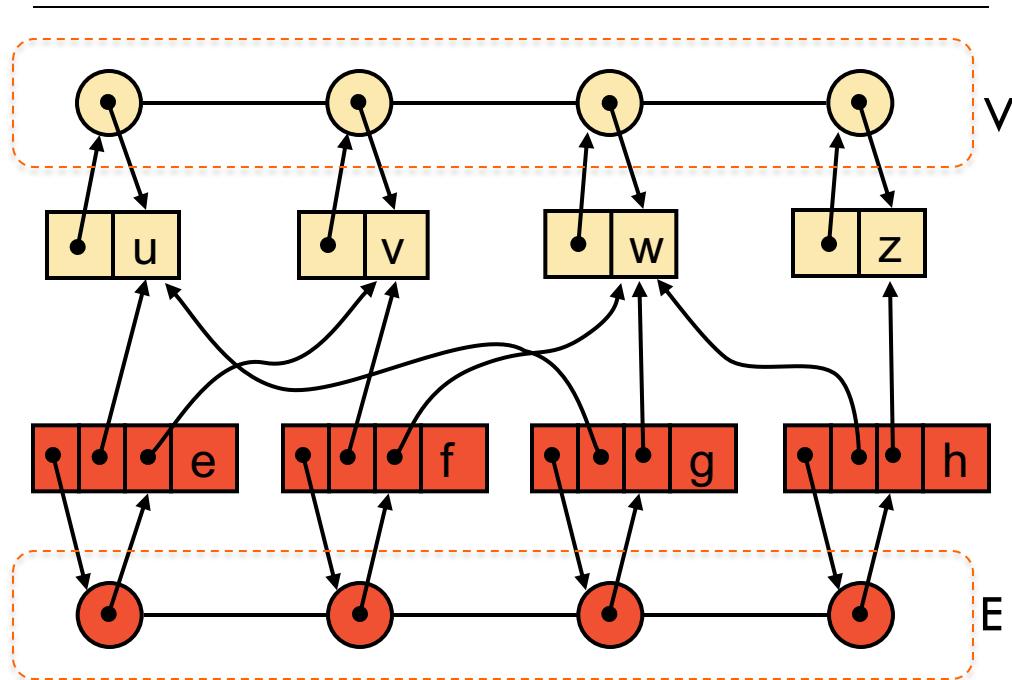
Vertex sequence holds

- sequence of vertices
- vertex object keeps track of its position in the sequence



Edge sequence

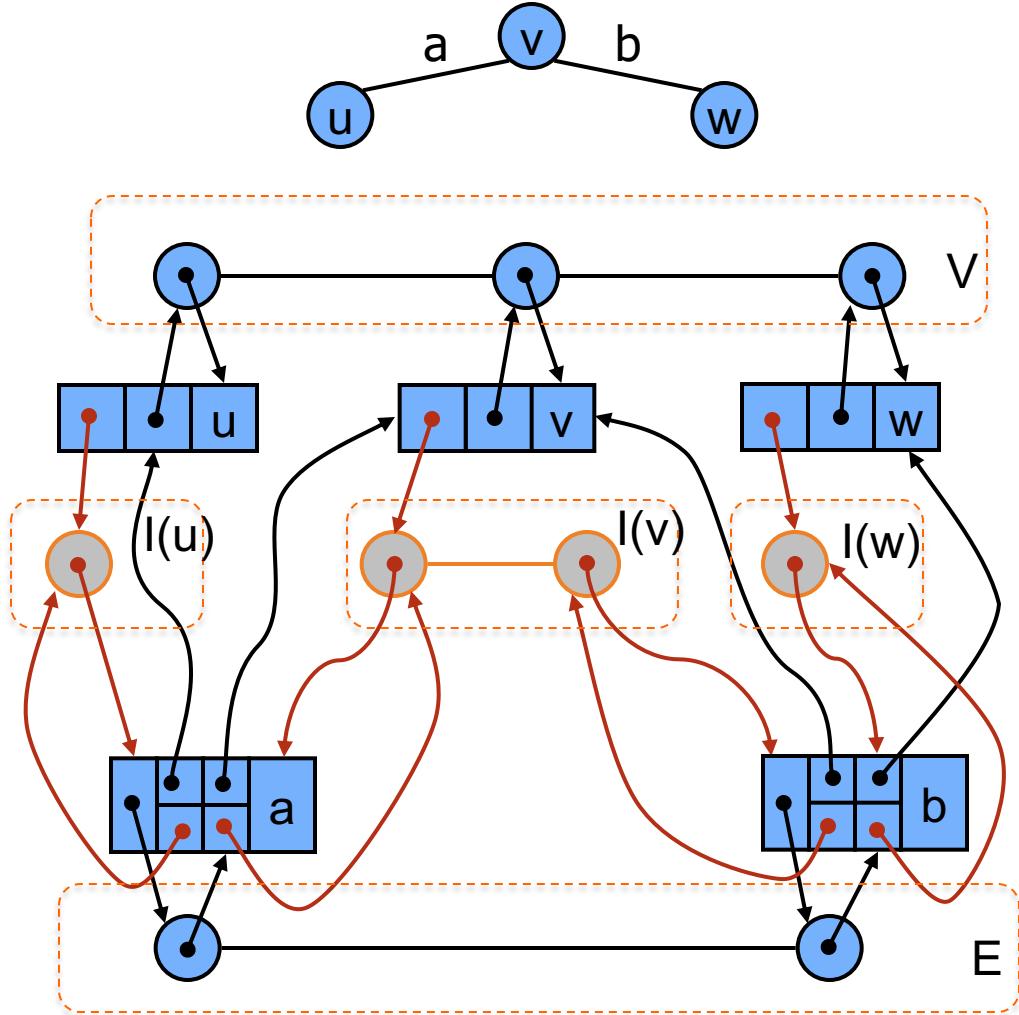
- sequence edges
- edge object keeps track of its position in the sequence
- Edge object points to the two vertices it connects



Adjacency List

Additionally each vertex keeps a sequence of edges incident on it

Edge objects keep reference to their position in the incidence sequence of its endpoints

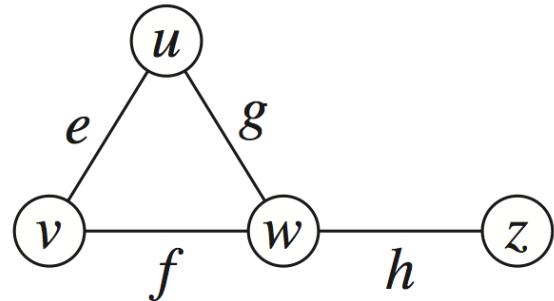


Adjacency Matrix Structure

Vertex array induces an index from 0 to n-1 for each vertex

2D-array adjacency matrix

- Reference to edge object for adjacent vertices
- Null for nonadjacent vertices



	0	1	2	3
$u \rightarrow$		e	g	
$v \rightarrow$	e		f	
$w \rightarrow$	g	f		h
$z \rightarrow$			h	

Asymptotic performance

<ul style="list-style-type: none"> ■ n vertices, m edges ■ no parallel edges ■ no self-loops 	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n + m)$	$O(n + m)$	$O(n^2)$
<code>incidentEdges(v)</code>	$O(m)$	$O(\deg(v))$	$O(n)$
<code>getEdge(u, v)</code>	$O(m)$	$O(\min(\deg(u), \deg(v)))$	$O(1)$
<code>insertVertex(x)</code>	$O(1)$	$O(1)$	$O(n^2)$
<code>insertEdge(u, v, x)</code>	$O(1)$	$O(1)$	$O(1)$
<code>removeVertex(v)</code>	$O(m)$	$O(\deg(v))$	$O(n^2)$
<code>removeEdge(e)</code>	$O(1)$	$O(1)$	$O(1)$

Graph traversals

A fundamental kind of algorithmic operation that we might wish to perform on a graph is **traversing the edges and the vertices** of that graph.

A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges.

For example, a **web crawler**, which is the data collecting part of a search engine, must explore a graph of hypertext documents by examining its vertices, which are the documents, and its edges, which are the hyperlinks between documents.

A traversal is efficient if it visits all the vertices and edges in linear time: **$O(n+m)$** where **n**=number of vertices, **m**=number of edges.

Graph traversal techniques

A systematic and structured way of visiting all the vertices and all the edges of a graph

Two main strategies:

- Depth first search
- Breadth first search

Given adjacency list representation of the graph with n vertices and m edges both traversal run in $O(n + m)$ time

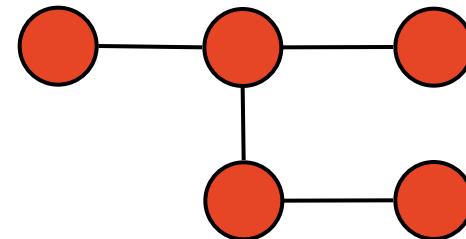
Reminder: Trees and Forests

An unrooted tree T is a graph such that

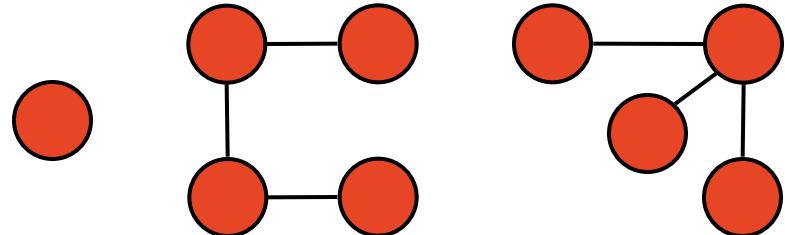
- T is connected
- T has no cycles

A forest is a graph without cycles. In other words, its connected components are trees

Fact: Every tree on n vertices has $n-1$ edges



Tree

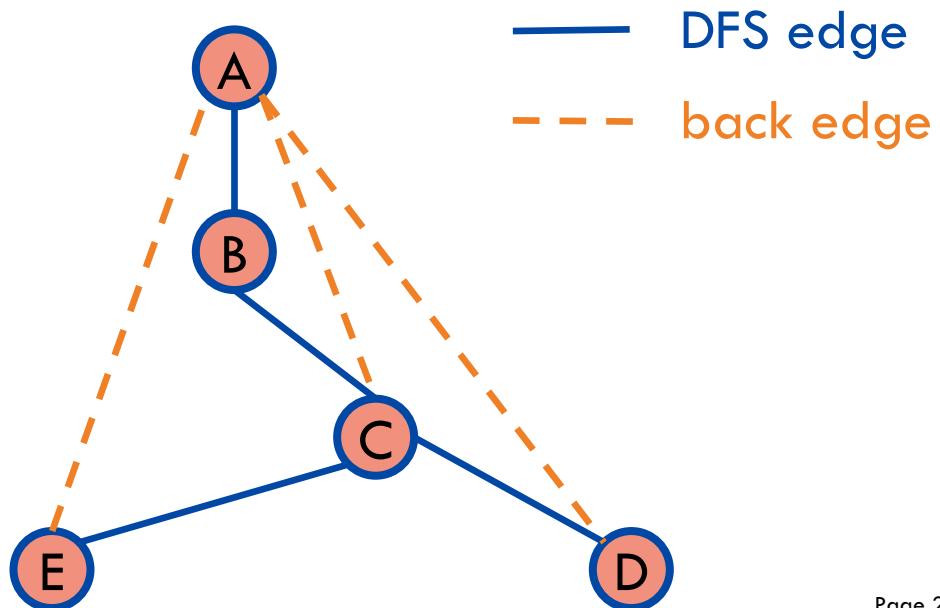
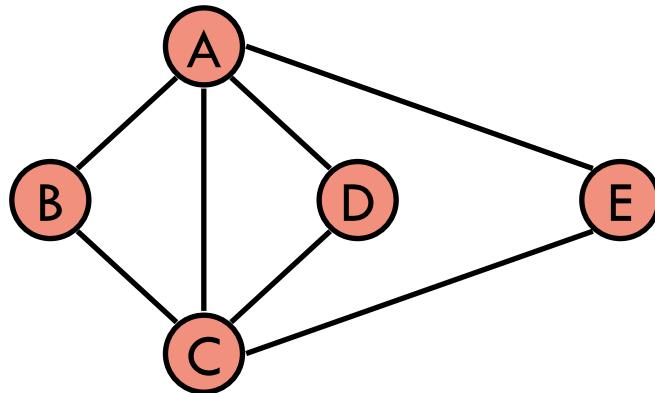


Forest

Depth-First Search (DFS)

This strategy tries to follow outgoing edges leading to yet unvisited vertices whenever possible, and backtrack if “stuck”

If an edge is used to discover a new vertex, we call it a DFS edge, otherwise we call it a back edge



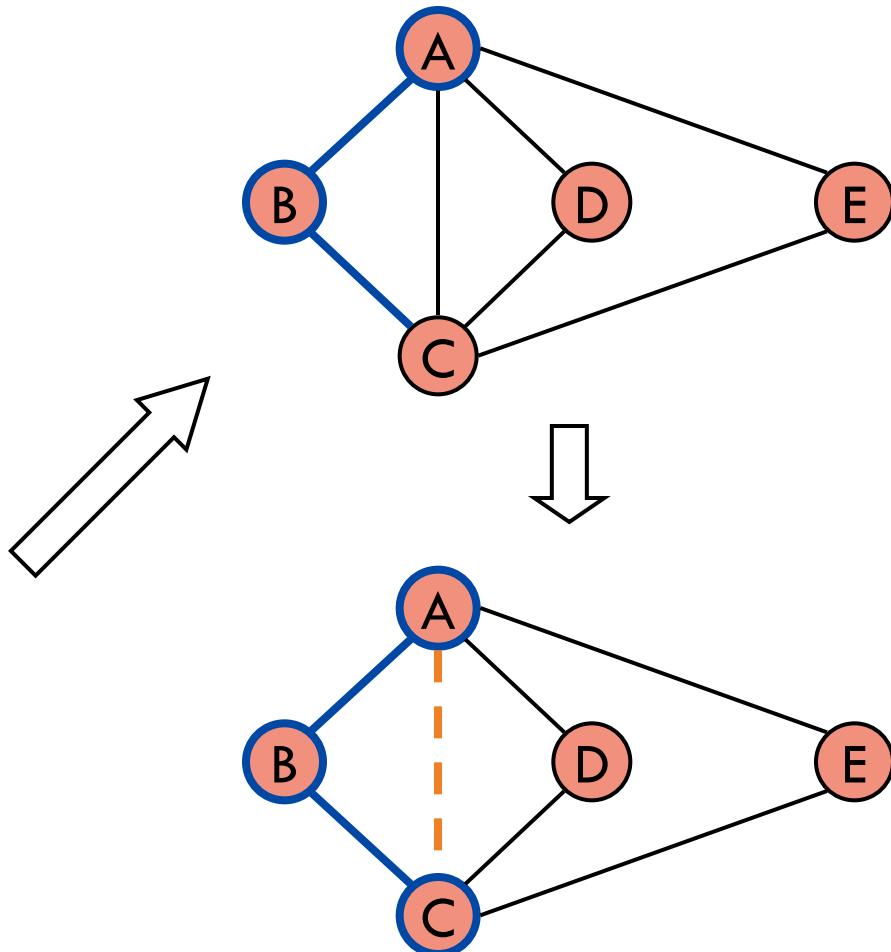
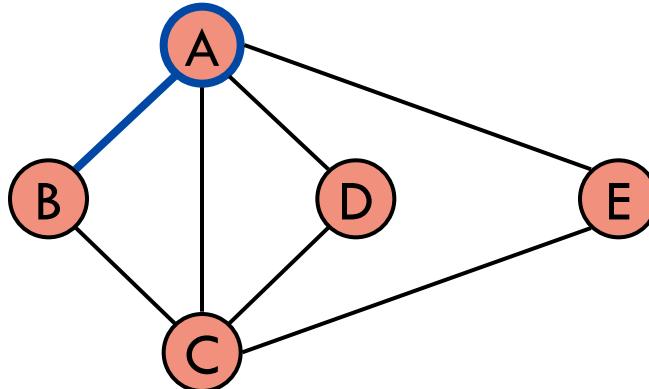
DFS pseudocode

```
def DFS(G):  
  
    # set things up for DFS  
    for u in G.vertices(): do  
        visited[u] ← False  
        parent[u] ← None  
  
    # visit vertices  
    for u in G.vertices(): do  
        if not visited[u] then  
            DFS_visit(u)  
  
    return parent
```

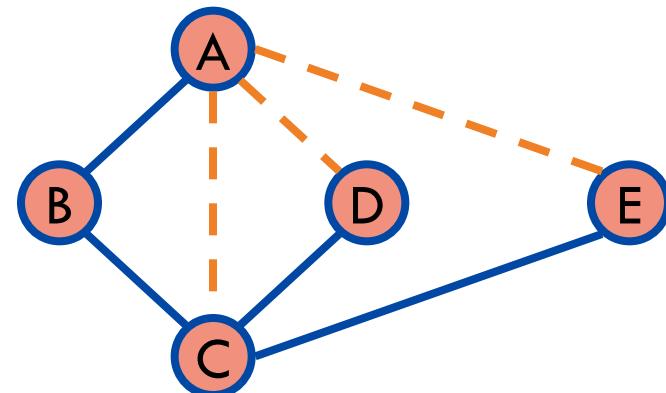
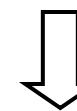
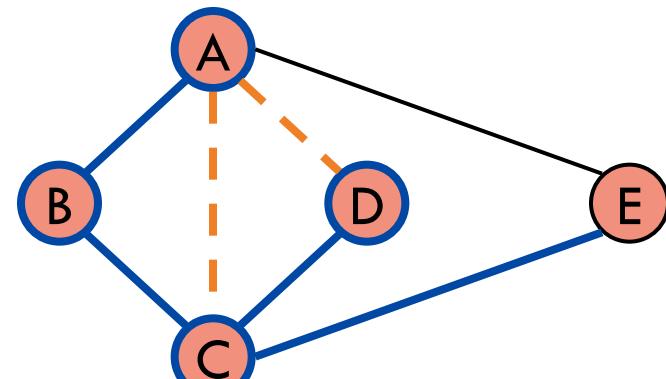
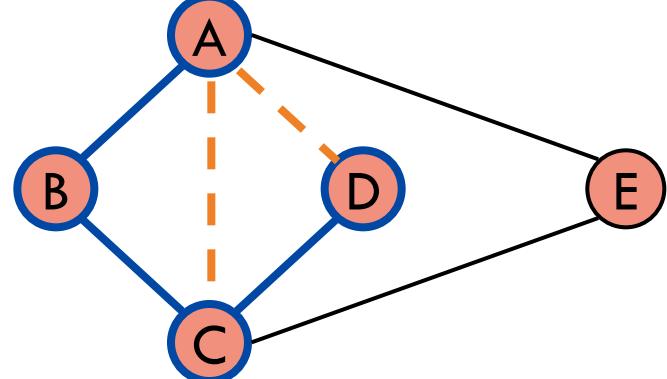
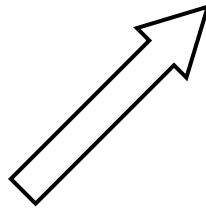
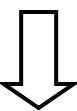
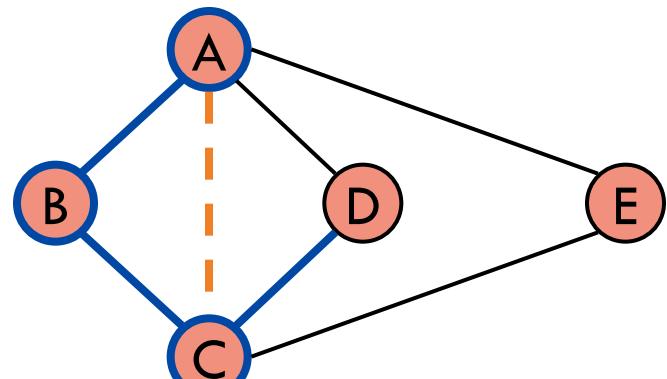
```
def DFS_visit(u):  
  
    visited[u] ← True  
  
    # visit neighbors of u  
    for v in G.incident(u): do  
        if not visited[v] then  
            parent[v] ← u  
            DFS_visit(v)
```

Example

- unexplored vertex
- visited vertex
- unexplored edge
- DFS edge
- - - back edge



Example (cont.)



DFS main function performance

```
def DFS(G):  
  
    # set things up for DFS  
    for u in G.vertices():  
        visited[u] ← False  
        parent[u] ← None  
  
    # visit vertices  
    for u in G.vertices():  
        if not visited[u]:  
            DFS_visit(u)  
  
    return parent
```

Assuming adjacency list representation

$O(n)$ time

$O(n)$ time not counting work done in `DFS_visit`

DFS_visit performance

Assuming adjacency list representation

$O(\deg(u))$ time not counting work done in recursive calls to DFS_visit

Thus, overall time is

$$O(\sum_u \deg(u)) = O(m)$$

```
def DFS_visit(u):
    visited[u] ← True
    # visit neighbors of u
    for v in G.incident(u) do
        if not visited[v] then
            parent[v] ← u
            DFS_visit(v)
```

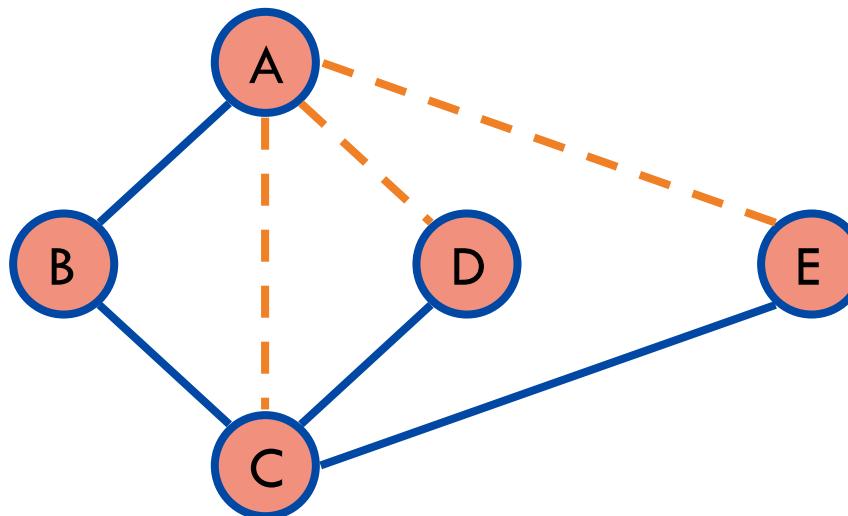
Properties of DFS

Let C_v be the connected component of v in our graph G

Fact: $\text{DFS_visit}(v)$ visits all vertices in C_v

Fact: Edges $\{ (u, \text{parent}[u]): u \in C_v \}$ form a spanning tree of C_v

Fact: Edges $\{ (u, \text{parent}[u]): u \in V \}$ form a spanning forest of G



DFS Applications

DFS can be used to solve other graph problems in $O(n + m)$ time:

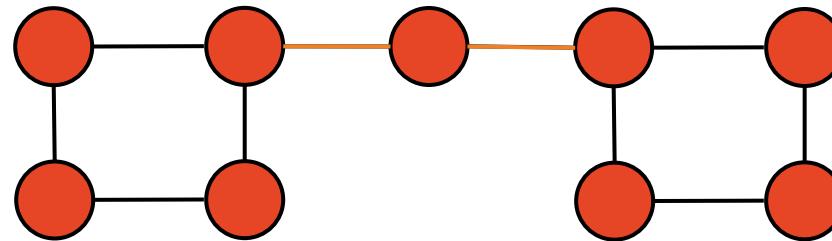
- Find a path between two given vertices, if any
- Find a cycle in the graph
- Test whether a graph is connected
- Compute connected components of a graph
- Compute spanning tree of a graph (if connected)

And is the building block of more sophisticated algorithms:

- testing bi-connectivity
- finding cut edges
- finding cut vertices

Identifying cut edges

In a connected graph $G=(V, E)$, we say that an edge (u, v) in E is a cut edge if $(V, E \setminus \{(u, v)\})$ is not connected



Identifying cut edges

In a connected graph $G=(V, E)$, we say that an edge (u, v) in E is a cut edge if $(V, E \setminus \{(u, v)\})$ is not connected

The cut edge problem is to identify all cut edges

Trivial $O(m^2)$ time algorithm: For each edge (u, v) in E , remove (u, v) and check using DFS if G is still connected, put back (u, v)

Better $O(nm)$ time algorithm: Only test edges in a DFS tree of G

Identifying cut edges in $O(n+m)$ time

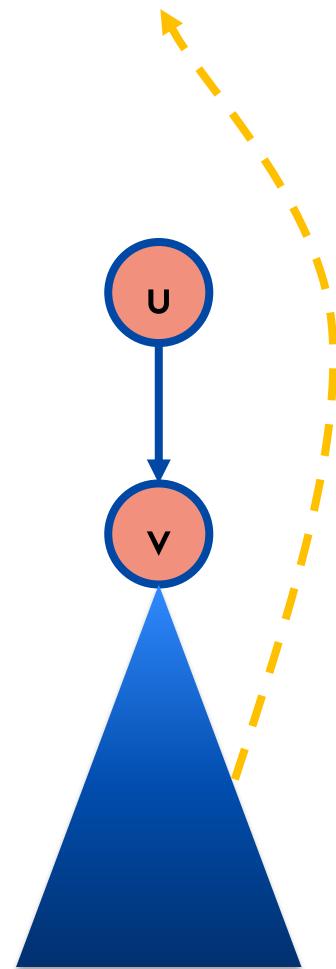
Compute a DFS tree of the input graph $G=(V, E)$

For every u in V , compute $\text{level}[u]$, its level in the DFS tree

For every vertex v compute the highest level that we can reach by taking DFS edges down the tree and then one back edge up. Call this $\text{down_and_up}[v]$

Fact: A DFS edge (u, v) where $u = \text{parent}[v]$ is not a cut edge if and only if $\text{down_and_up}[v] \leq \text{level}[u]$

Basis of an $O(n+m)$ time algorithm for finding cut edges

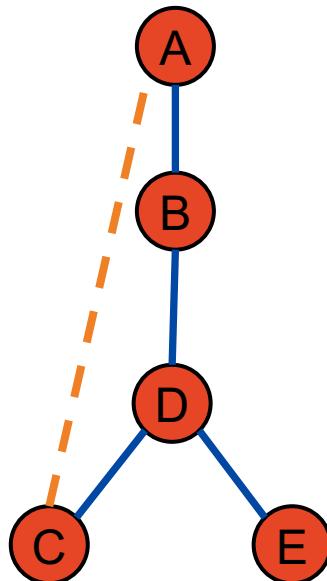
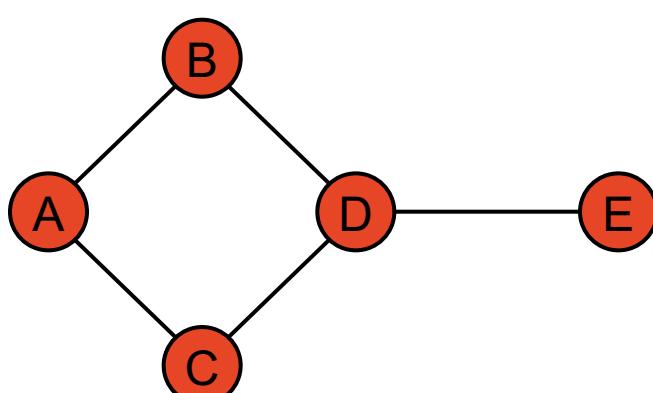


Identifying cut edges in $O(n+m)$ time

Compute a DFS tree of the input graph $G=(V, E)$

For every u in V , compute $\text{level}[u]$, its level in the DFS tree

For every vertex v compute the highest level that we can reach by taking DFS edges down the tree and then one back edge up. Call this $\text{down_and_up}[v]$

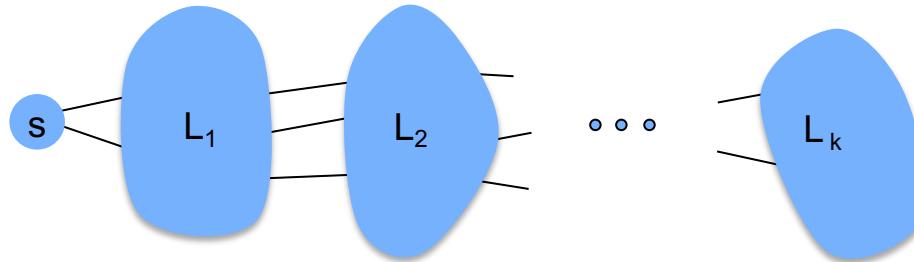


	level	d&u
A	0	0
B	1	0
C	2	0
D	2	0
E	3	3

Breadth-First Search (BFS)

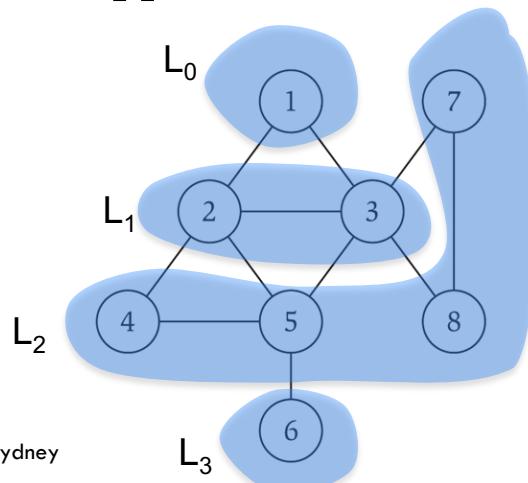
This strategy tries to visit all vertices at distance k from a start vertex s before visiting vertices at distance $k + 1$:

- $L_0 = \{s\}$
- $L_1 = \text{vertices one hop away from } s$
- $L_2 = \text{vertices two hops away from } s \text{ but no closer}$
- ⋮
- $L_k = \text{vertices } k \text{ hops away from } s \text{ but no closer}$

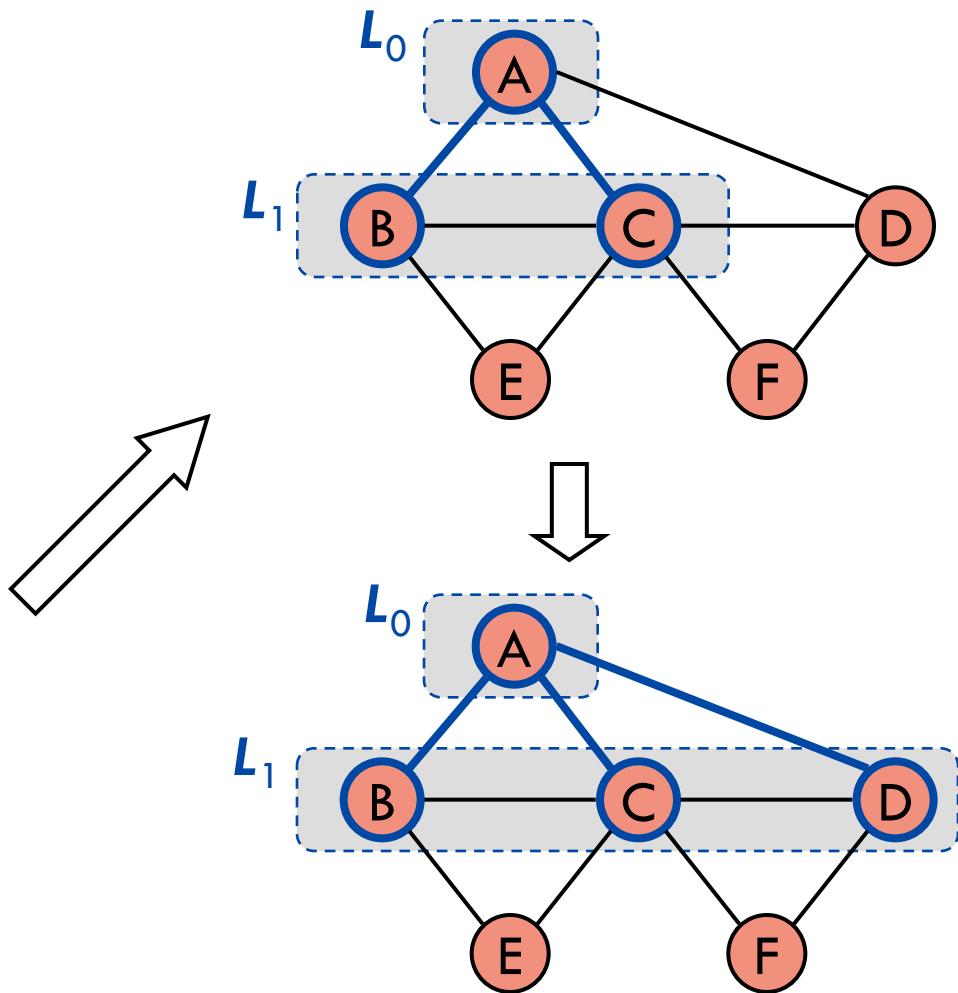
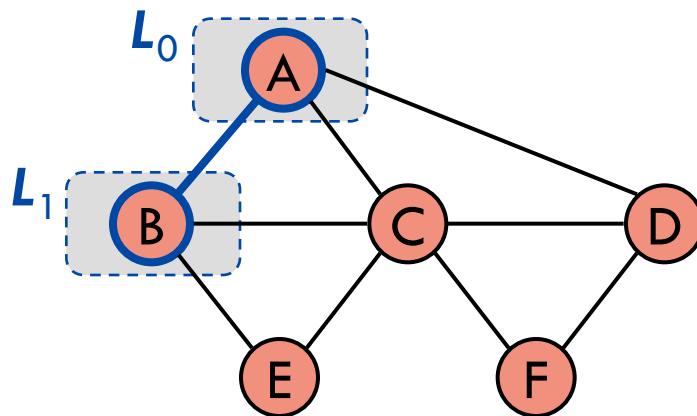
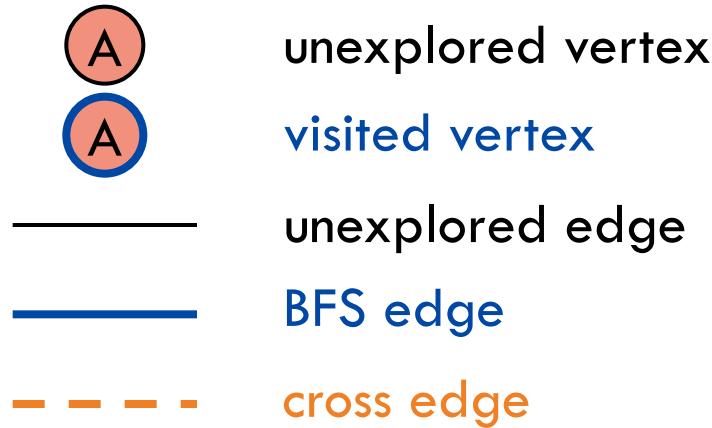


BFS

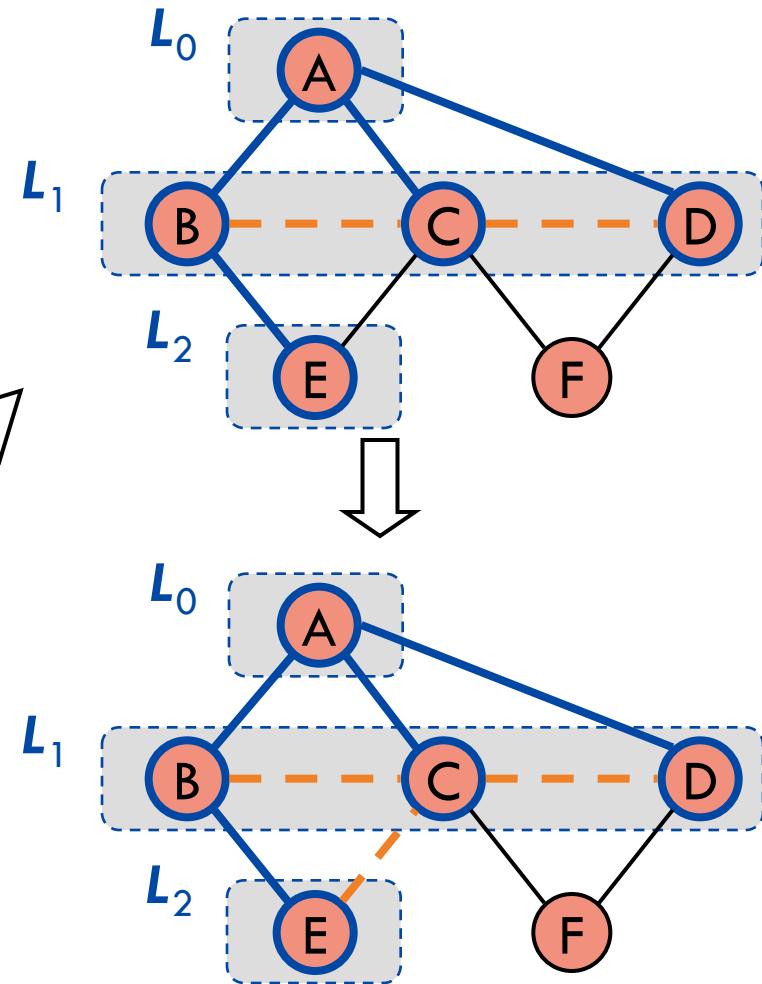
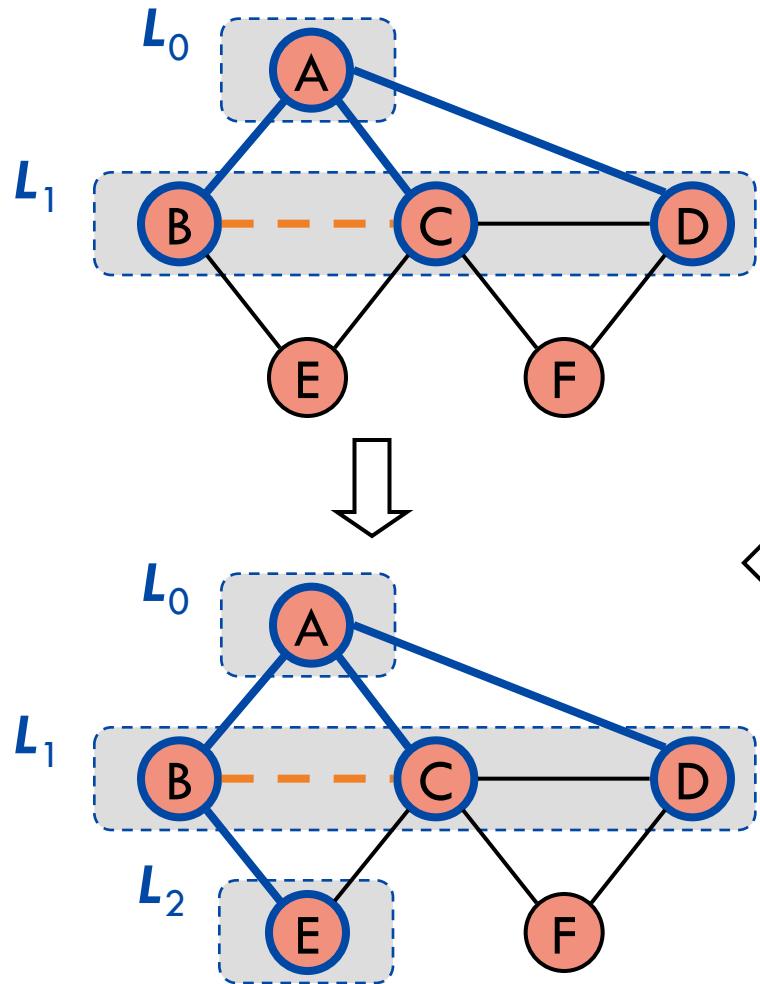
```
def BFS(G,s):  
  
    # set things up for BFS  
    for u in G.vertices(): do  
        seen[u] ← False  
        parent[u] ← None  
  
    seen[s] ← True  
    layers ← []  
    current ← [s]  
    next ← []  
  
    # process current layer  
    while not current.is_empty() do  
        layers.append(current)  
        # iterate over current layer  
        for u in current do  
            for v in G.incident(u) do  
                if not seen[v] then  
                    next.append(v)  
                    seen[v] ← True  
                    parent[v] ← u  
  
        # update current & next layers  
        current ← next  
        next ← []  
  
    return layers, parent
```



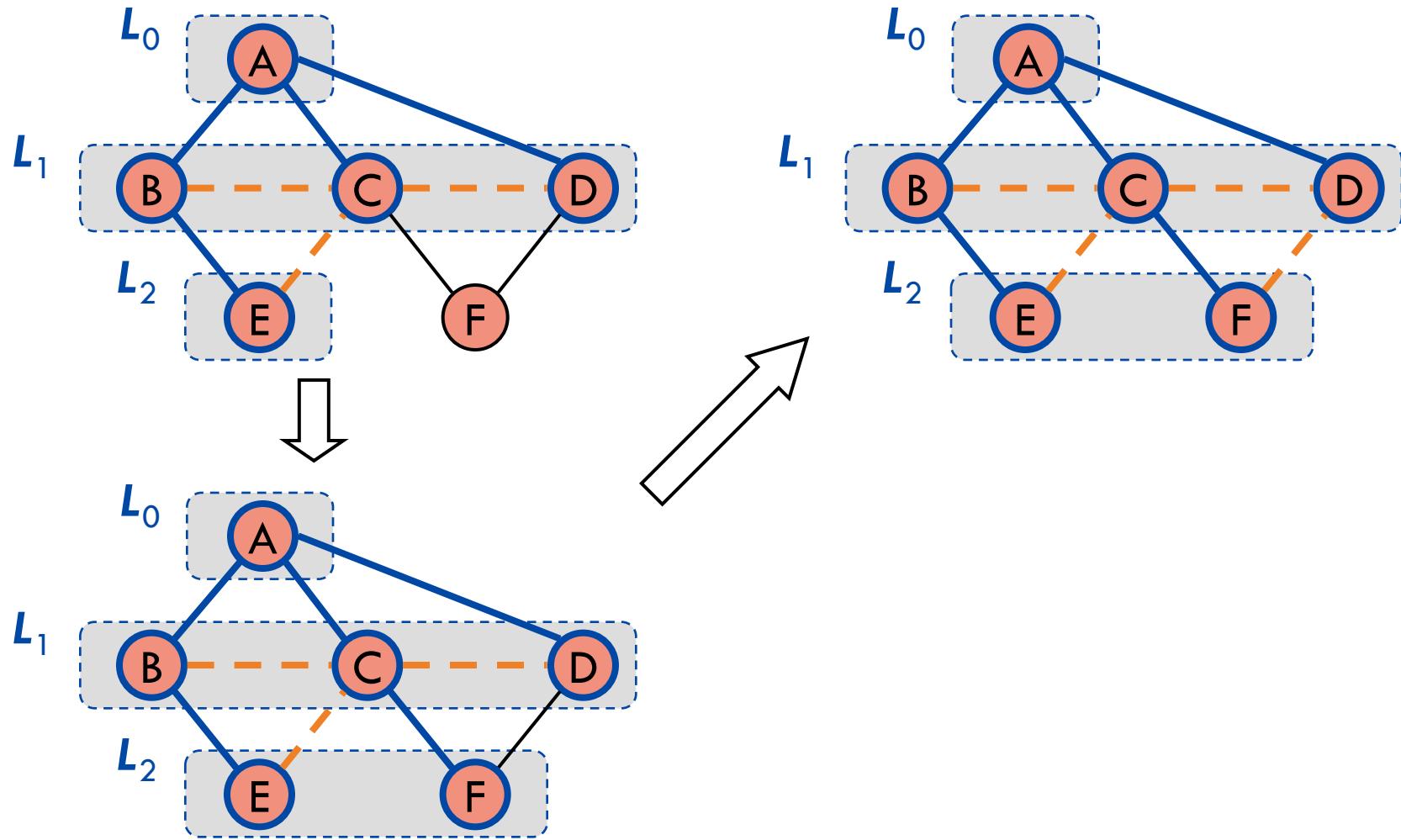
Example



Example (cont.)



Example (cont.)



Properties

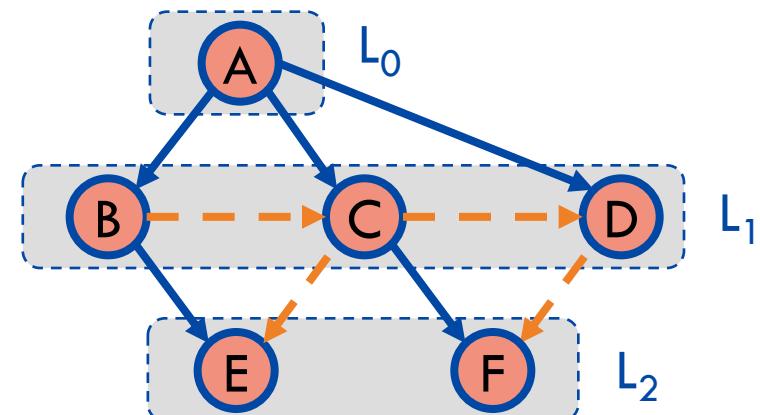
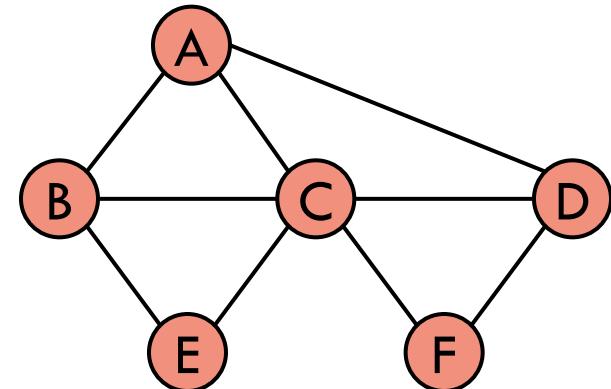
Let C_v be the connected component of v in our graph G

Fact: $\text{BFS}(G, s)$ visits all vertices in C_s

Fact: Edges $\{ (u, \text{parent}[u]): u \in C_s \}$ form a spanning tree T_s of C_s

Fact: For each v in L_i there is a path in T_s from s to v with i edges

Fact: For each v in L_i any path in G from s to v has at least i edges



BFS performance

```
def BFS(G,s):  
  
    # set things up for BFS  
    for u in G.vertices() do  
        seen[u] ← False  
        parent[u] ← None  
  
    seen[s] ← True  
    layers ← []  
    current ← [s]  
    next ← []  
  
    # process current layer  
    while not current.is_empty() do  
        layers.append(current)  
        # iterate over current layer  
        for u in current do  
            for v in G.incident(u) do  
                if not seen[v] then  
                    next.append(v)  
                    seen[v] ← True  
                    parent[v] ← u  
  
    # update curr and next layers  
    current ← next  
    next ← []  
  
    return layers
```

$\mathcal{O}(n)$ time

$\mathcal{O}(\sum_u \deg(u)) = \mathcal{O}(m)$ time

BFS performance

Fact: Assuming adjacency list representation we can perform a BFS traversal of a graph with n vertices and m edges in $O(n+m)$ time

Fact: Assuming adjacency matrix representation we can perform a BFS traversal of a graph with n vertices and m edges in $O(n^2)$ time

The additional attributes about the vertices (seen and parent) can be associated directly via Vertex class or we can use an external map data structure

BFS Applications

BFS can be used to solve other graph problems in $O(n + m)$ time:

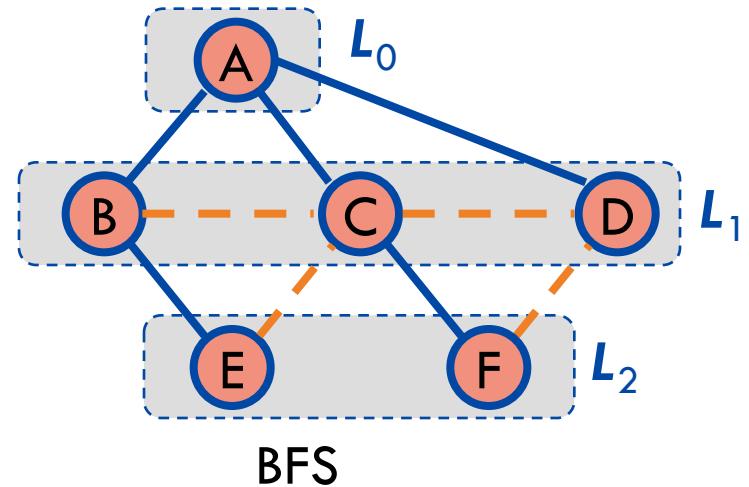
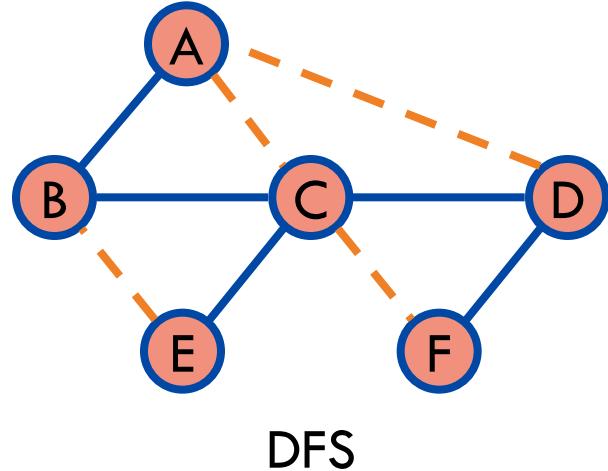
- Find a shortest path between two given vertices
- Find a cycle in the graph
- Test whether a graph is connected
- Compute a spanning tree of a graph (if connected)

And is the building block of more sophisticated algorithms:

- Testing if graph is bipartite

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓
Cut edges	✓	

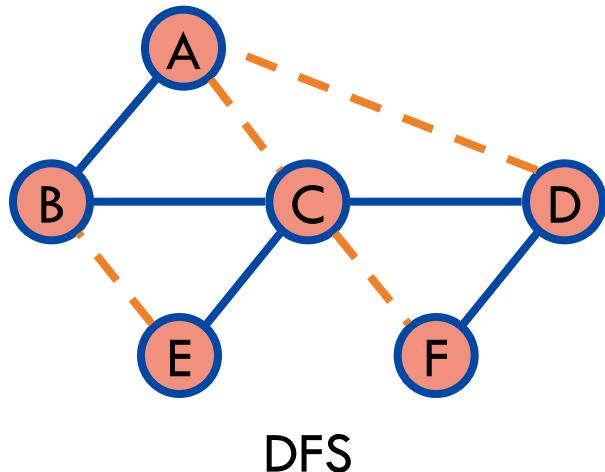


DFS vs. BFS (cont.)

Non-tree DFS edge (v, w)

w is an ancestor of v
in the DFS tree

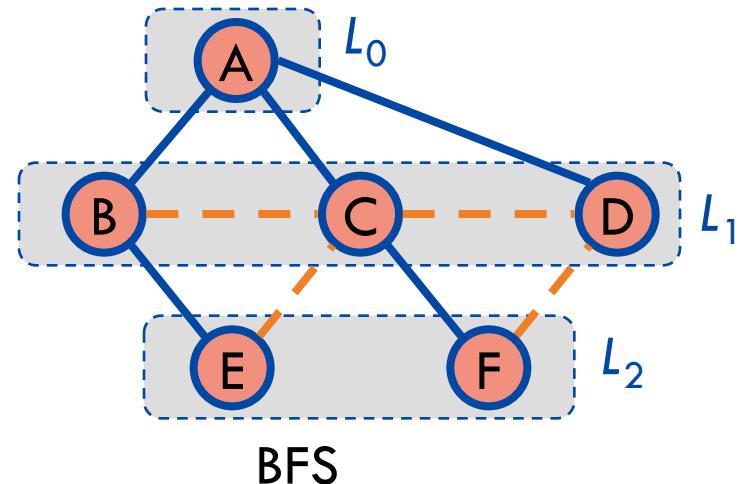
Called back edges



Non-tree BFS edge (v, w)

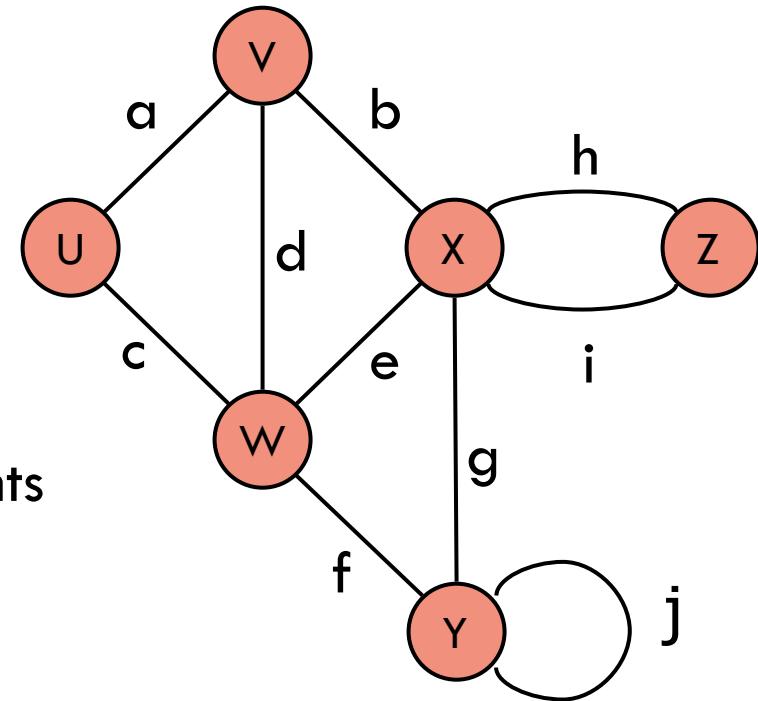
w is in the same level as v or
in the next level

Called cross edges



Terminology (Undirected graphs)

- Edges connect **endpoints**
e.g., W and Y for edge f
- Edges are **incident** on endpoints
e.g., a, d, and b are incident on V
- **Adjacent** vertices are connected
e.g., U and V are adjacent
- **Degree** is # of edges on a vertex
e.g., X has degree 5
- **Parallel edges** share same endpoints
e.g., h and i are parallel
- **Self-loop** have only one endpoint
e.g., j is a self-loop
- **Simple** graphs have no parallel or self-loops



Terminology (Directed graphs)

- Edges go from **tail** to **head**
e.g., W is the tail of c and U its head
- **Out-degree** is # of edges out of a vertex
e.g., W has out-degree 2
- **In-degree** is # of edges into a vertex
e.g., W has in-degree 1
- **Parallel edges** share tail and head
e.g., no parallel edge on the right
- **Self-loop** have same head and tail
e.g., X has a self-loop
- **Simple** directed graphs have no parallel or self-loops, but are allowed to have anti-parallel loops like f and a

