

CS 590: Algorithms

Lecture 10 & 11: Graphs & Graph's algorithm

Graphs



- Trees are limited in that a data structure can only have one parent
- Graphs overcome this limitation
- Graphs were being studied long before computers were invented
- Graphs algorithms run
 - large communication networks
 - the software that makes the Internet function
 - programs to determine the optimal placement of components on a silicon chip
- Graphs describe
 - roads maps
 - airline routes
 - course prerequisites

Graph Terminology

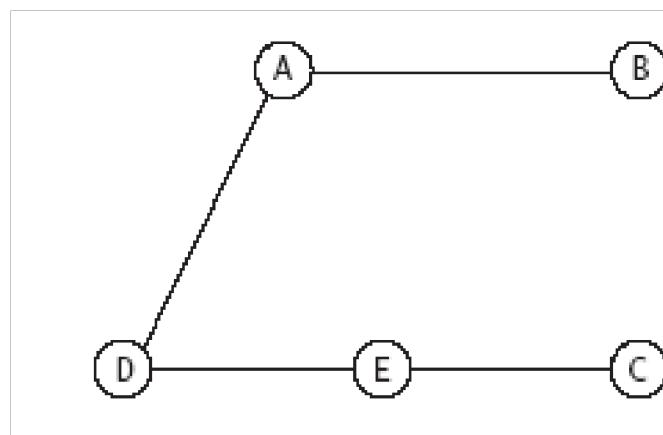


- A graph is a data structure that consists of a set of vertices (or nodes) and a set of edges (relations) between pairs of vertices
- Edges represent paths or connections between vertices
- Both the set of vertices and the set of edges must be finite
- Either set may be empty (if the set of vertices is empty, the set of edges also must be empty)
- We restrict our discussion to simple graphs in which there is at least one edge from a given vertex to another vertex

Visual Representation of Graphs



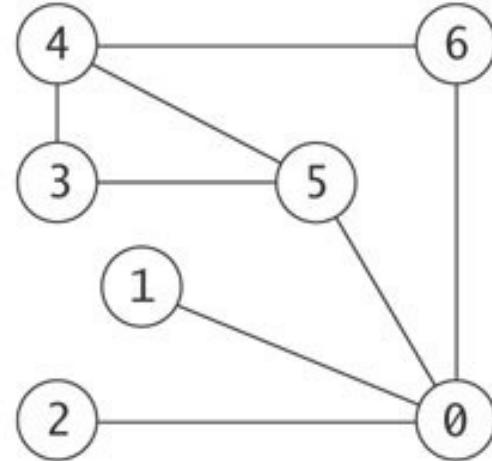
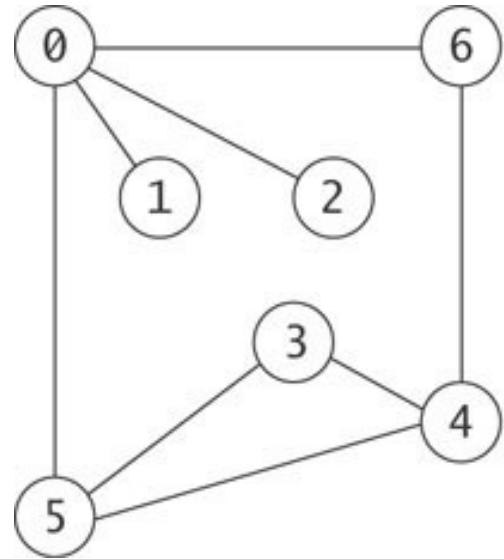
- Vertices are represented as points or labeled circles, and edges are represented as line segments joining the vertices
- Each edge is represented by the two vertices it connects
- If there is an edge between vertices x and y , there is a path from x to y and vice versa



- $V = \{A, B, C, D, E\}$
- $E = \{\{A, B\}, \{A, D\}, \{C, E\}, \{D, E\}\}$

Visual Representation of Graphs

- The physical layout of the vertices and their labeling is not relevant

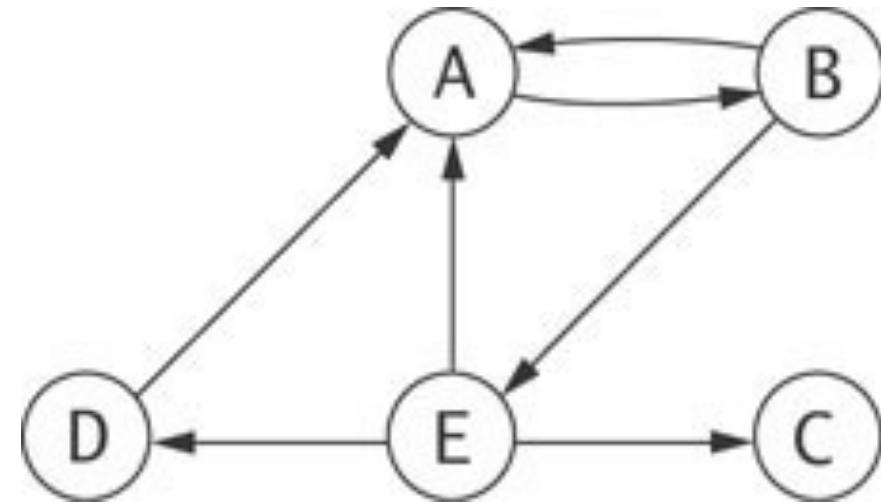


- $V = \{0, 1, 2, 3, 4, 5, 6\}$
- $E = \{\{0, 1\}, \{0, 2\}, \{0, 5\}, \{0, 6\}, \{3, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\}$



Directed and Undirected Graphs

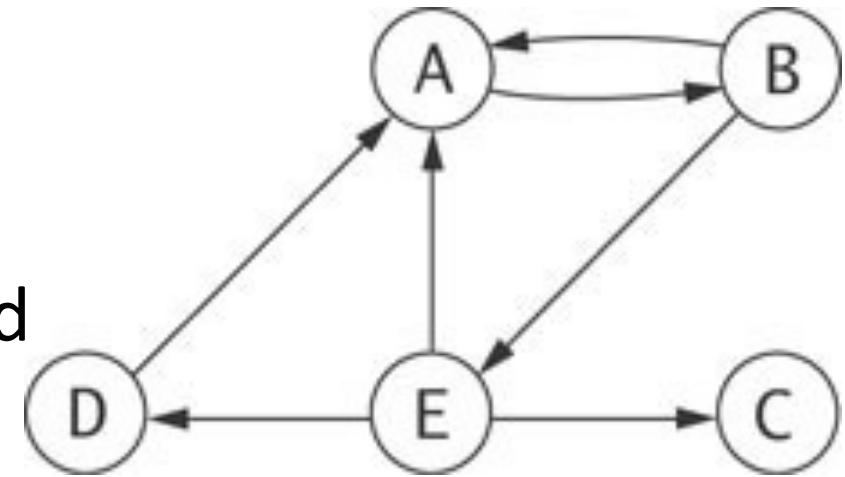
- The edges of a graph are directed if the existence of an edge from A to B does not necessarily guarantee that there is a path in both directions
- A graph with directed edges is called a directed graph or digraph
- A graph with undirected edges is an undirected graph, or simply a graph





Directed and Undirected Graphs (cont.)

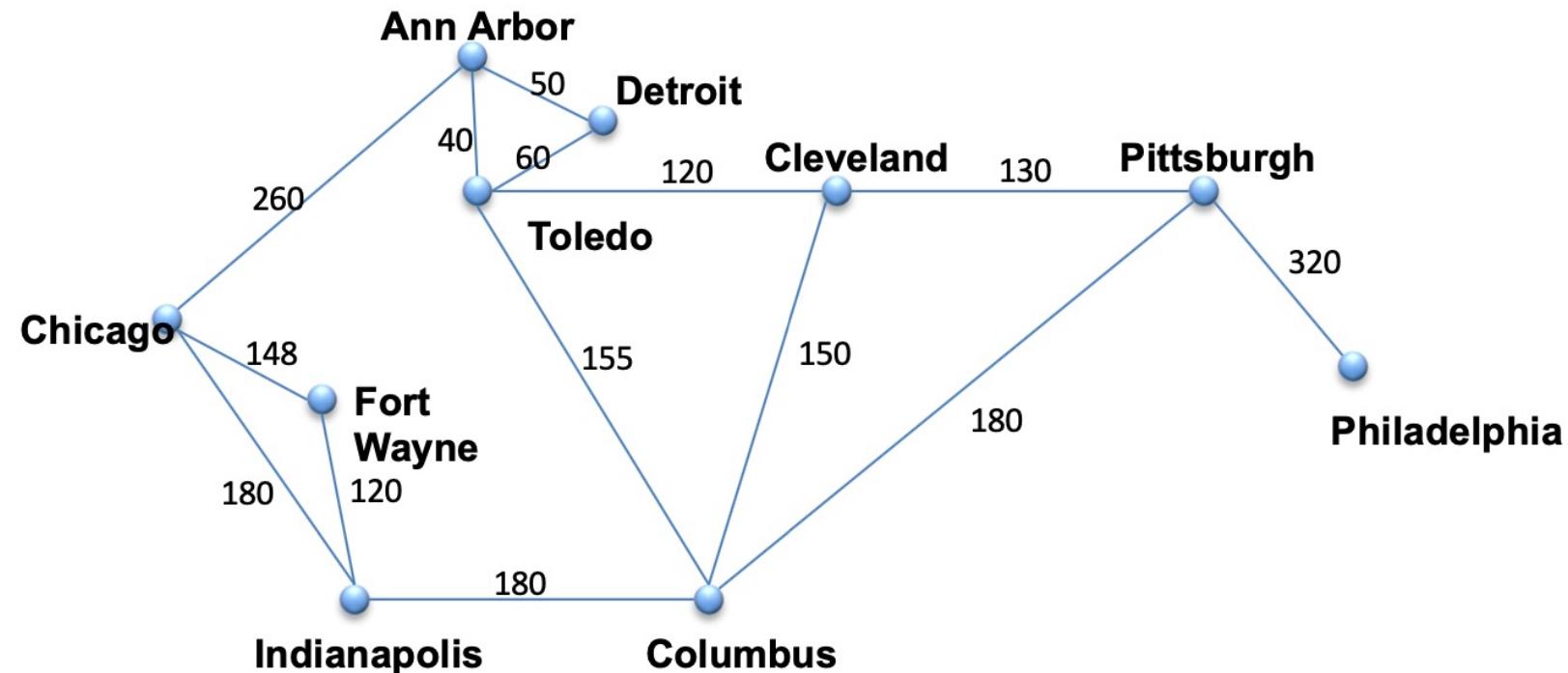
- A directed edge is like a one way street; you can travel in only one direction
- Directed edges are represented as lines with an arrowhead on one end (undirected edges do not have an arrowhead at either end)
- Directed edges are represented by ordered pairs of vertices {source, destination}; the edges for the digraph on this slide are:
 - $\{\{A, B\}, \{B, A\}, \{B, E\}, \{D, A\}, \{E, A\}, \{E, C\}, \{E, D\}\}$





Directed and Undirected Graphs (cont.)

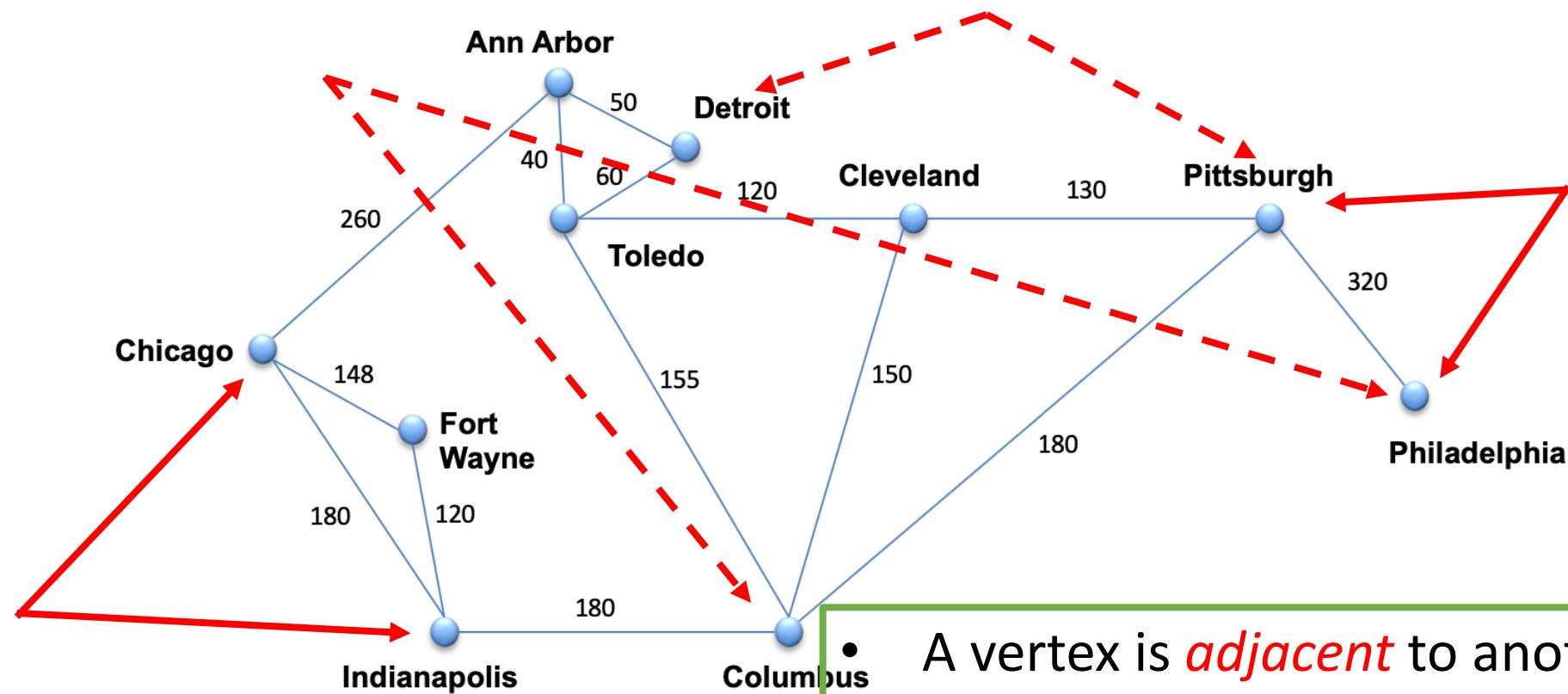
- The edges in a graph may have associated values known as their weights
- A graph with weighted edges is known as a weighted graph





Paths and Cycles

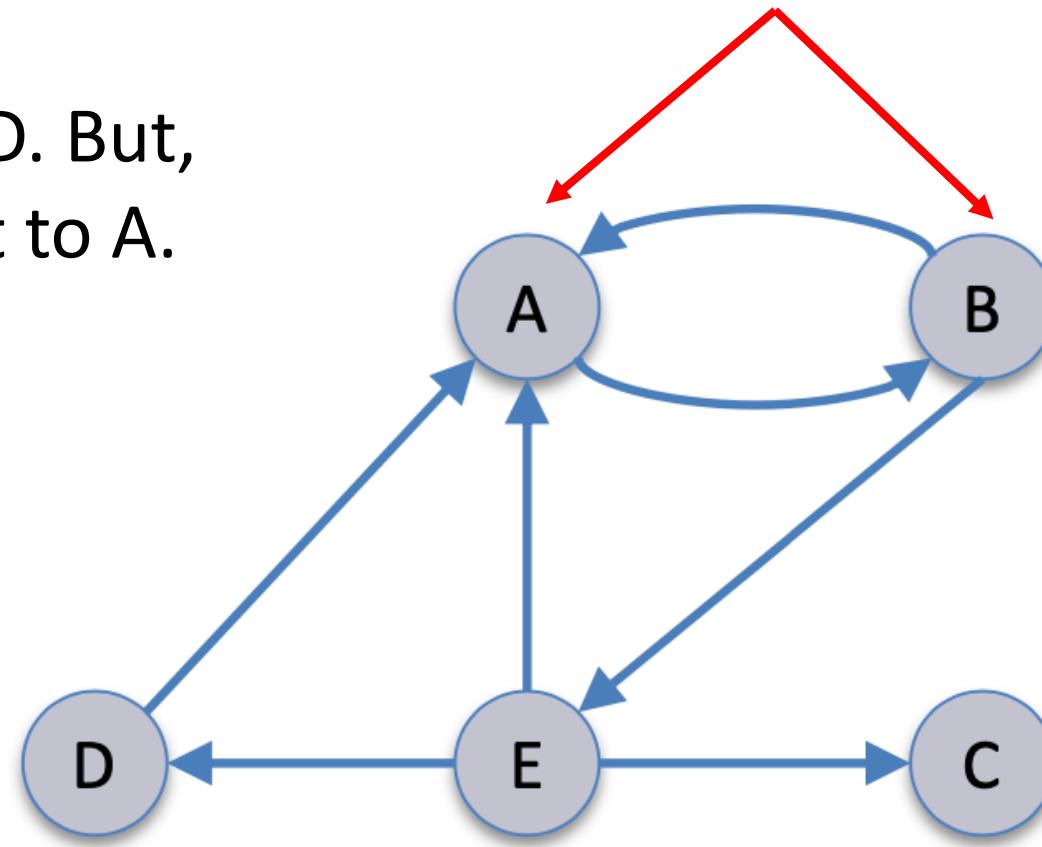
- The following definitions describe pathways between vertices:



- A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex.
- “Destination” is adjacent to “Source”

Paths and Cycles (cont.)

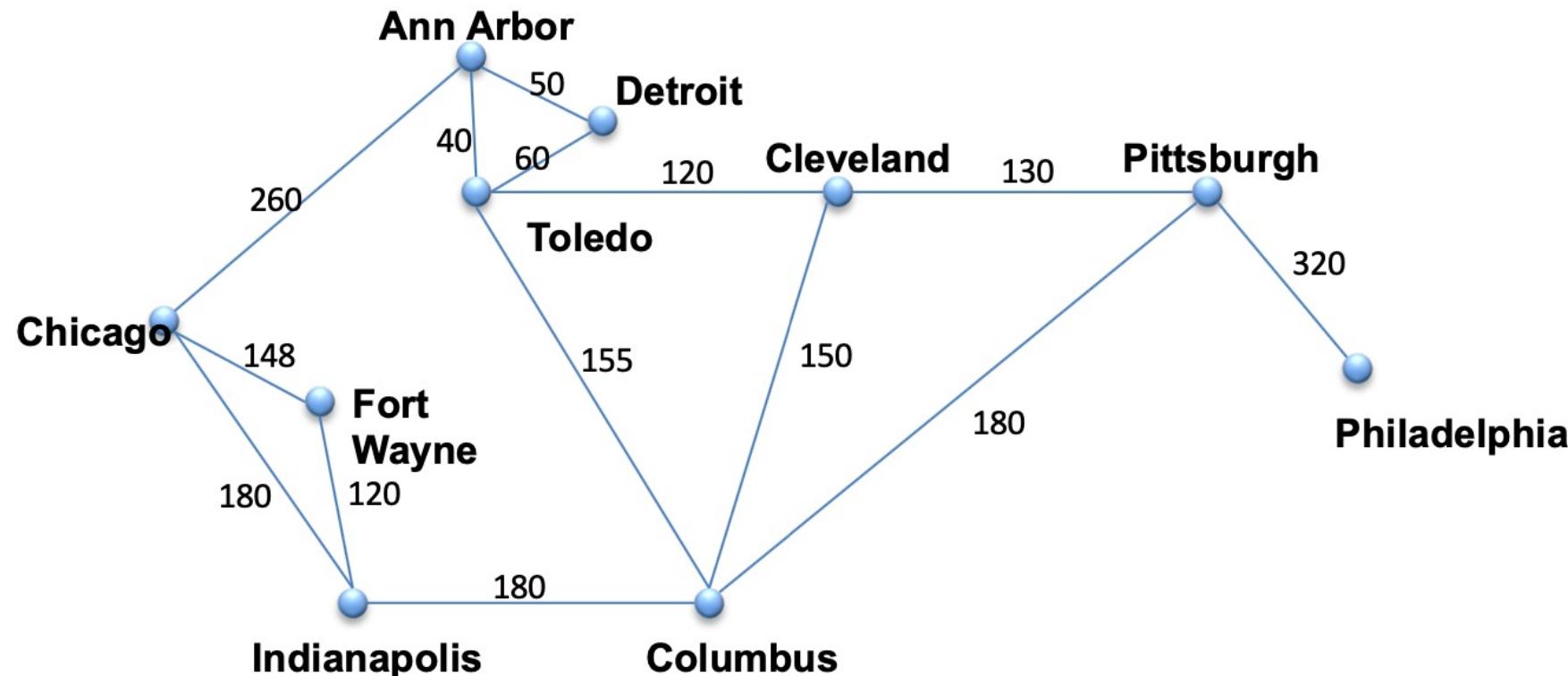
- A is adjacent to D. But,
- D is not adjacent to A.





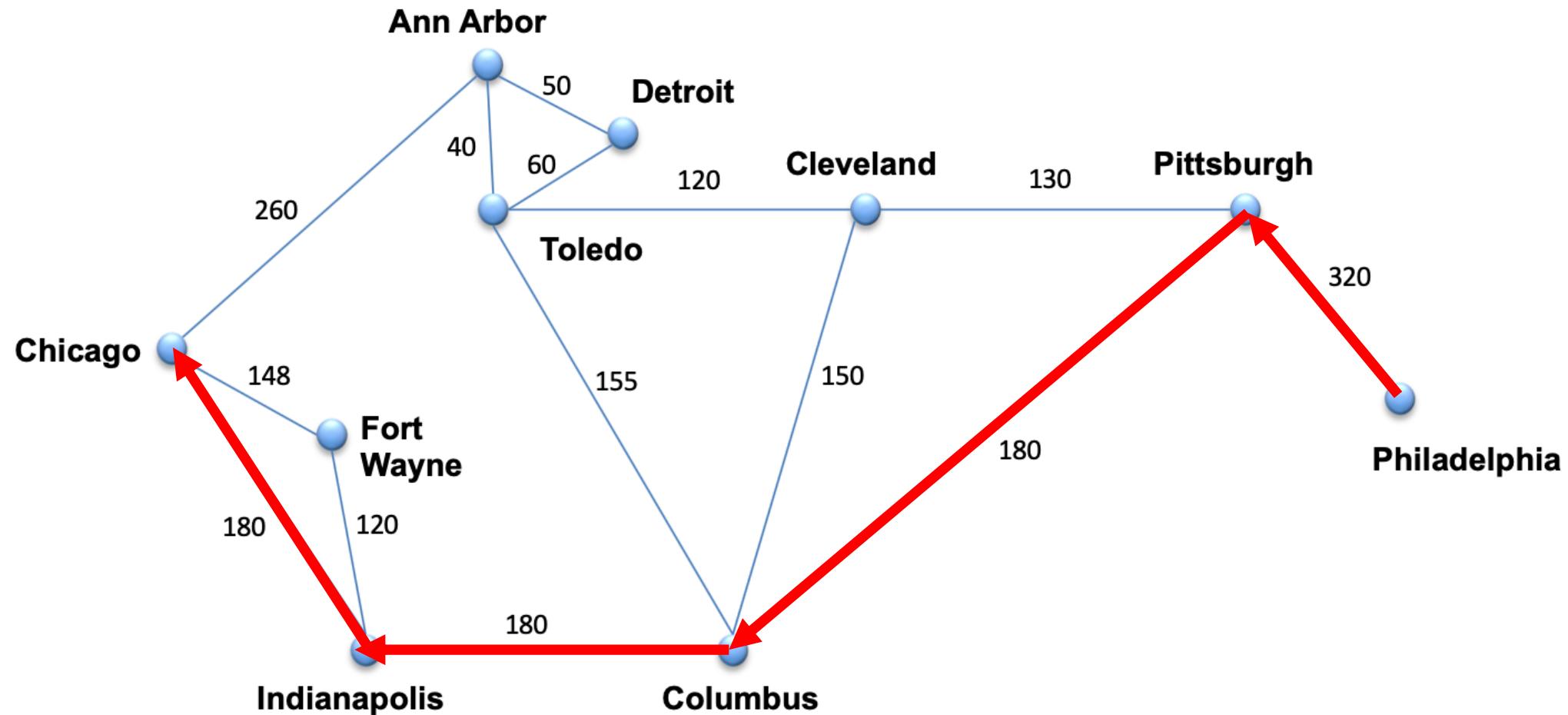
Paths and Cycles (cont.)

- A **path** is a sequence of vertices in which each successive vertex is **adjacent** to its predecessor.
- In a **simple path**, the vertices and edges are **distinct**.
 - The first and last vertex may be the same – **cycle**.



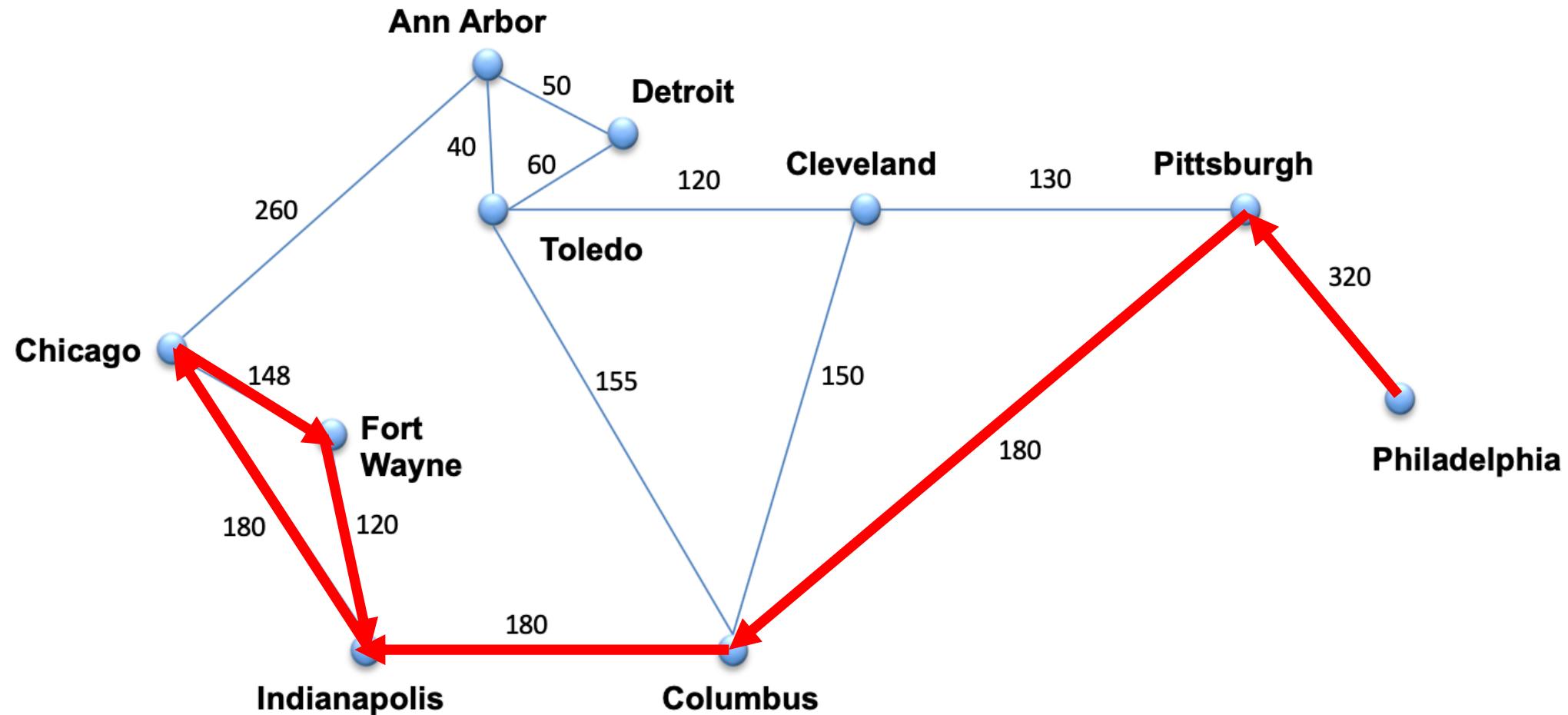


Paths and Cycles (cont.)



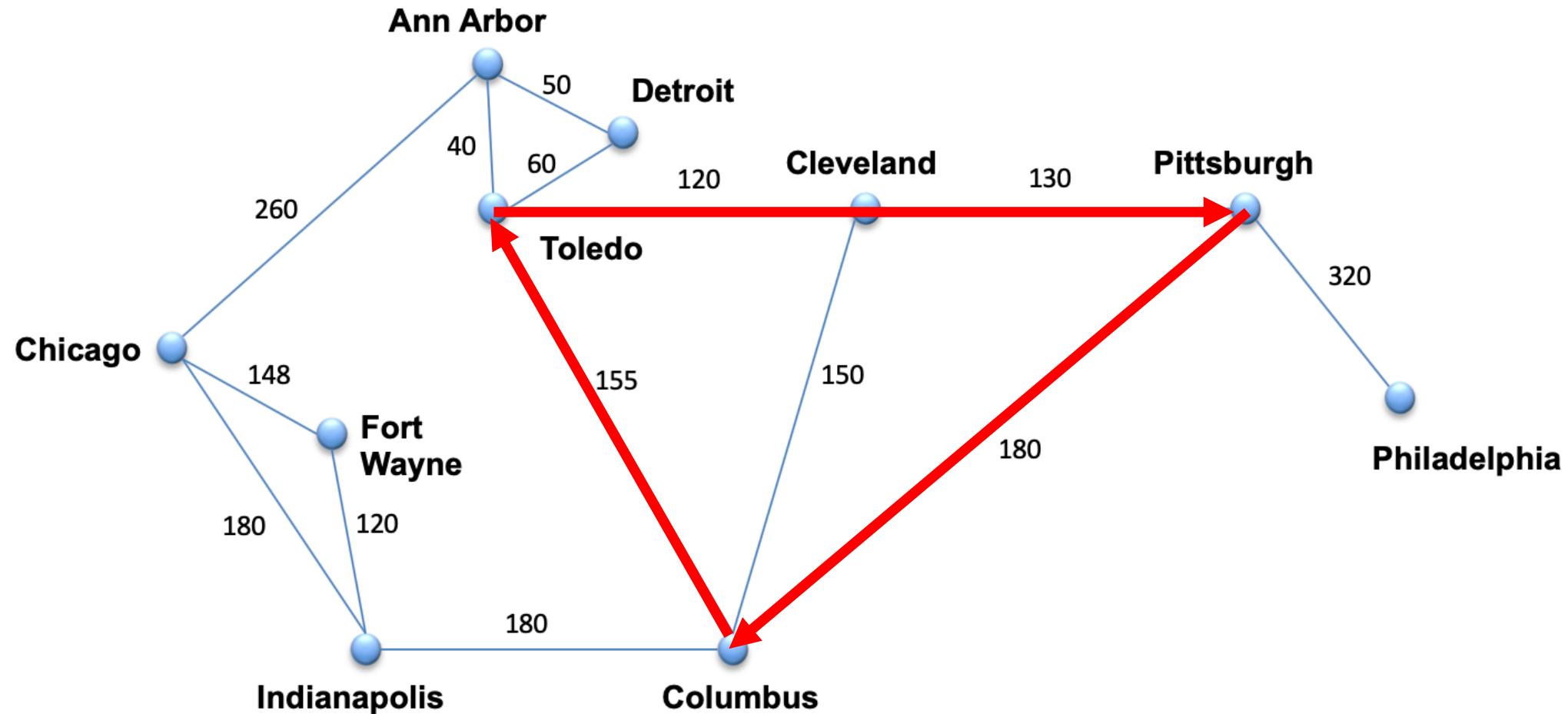


Paths and Cycles (cont.)



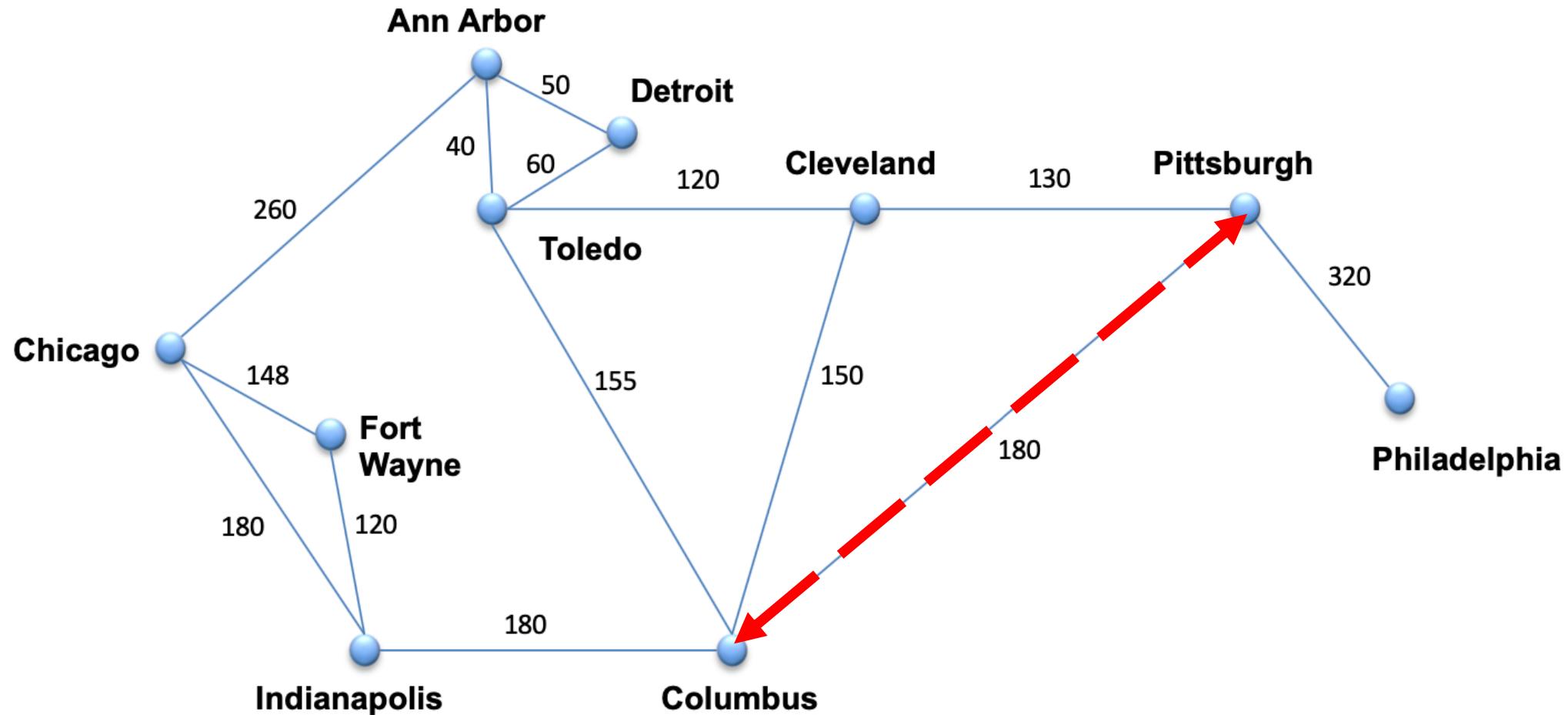


Paths and Cycles (cont.)



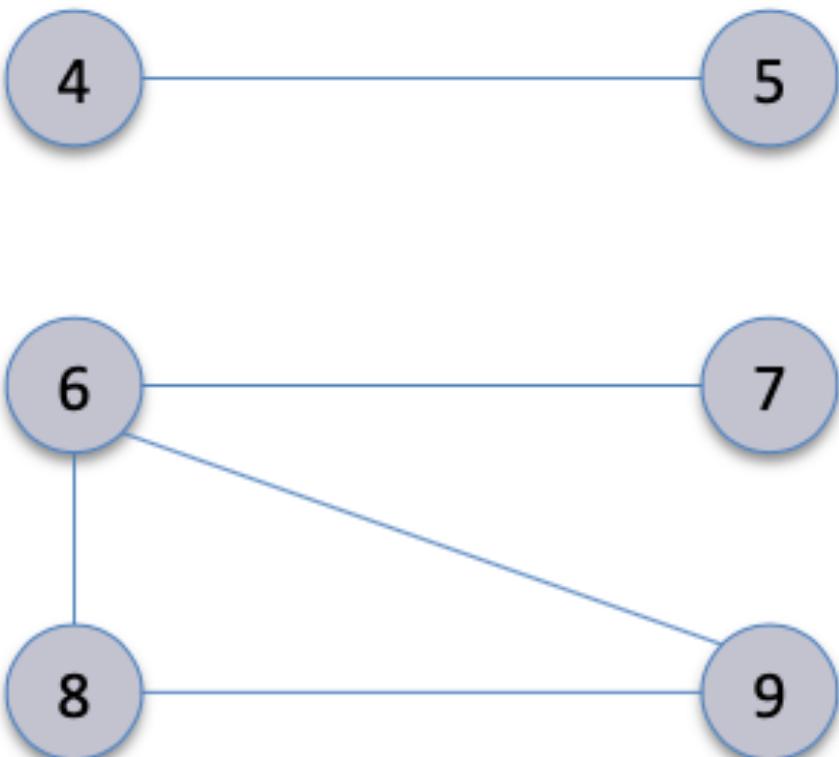
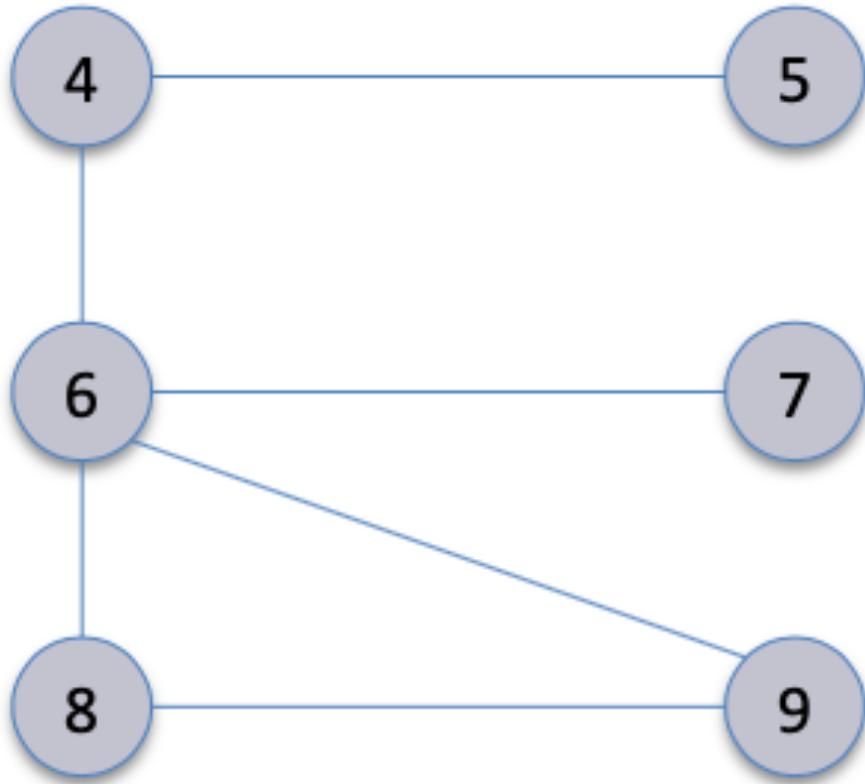


Paths and Cycles (cont.)



Paths and Cycles (cont.)

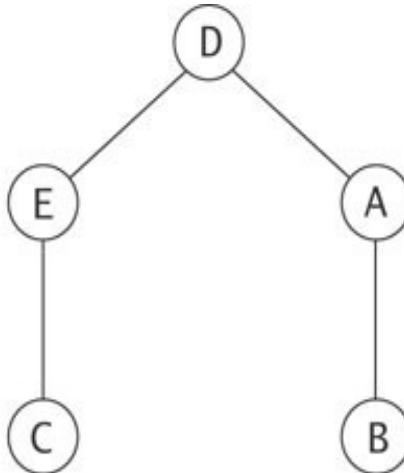
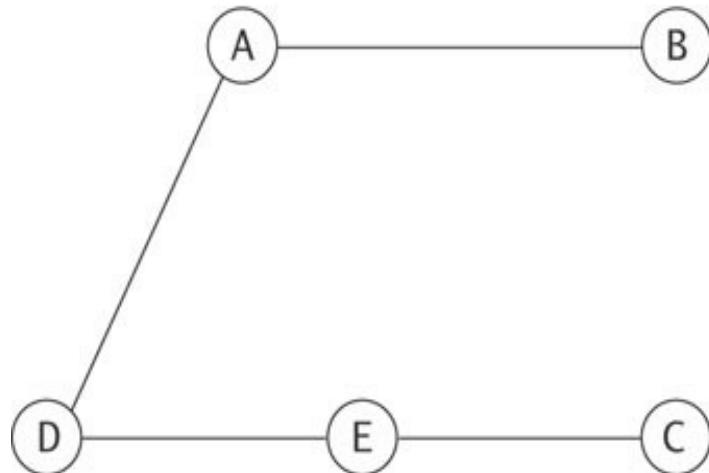
- If a graph is not connected, it is considered ***unconnected***, but will still consist of connected components.



Relationship between Graphs and Trees



- A tree is a special case of a graph
- Any graph that is
 - Connected
 - contains no cycles can be viewed as a tree by making one of the vertices the root



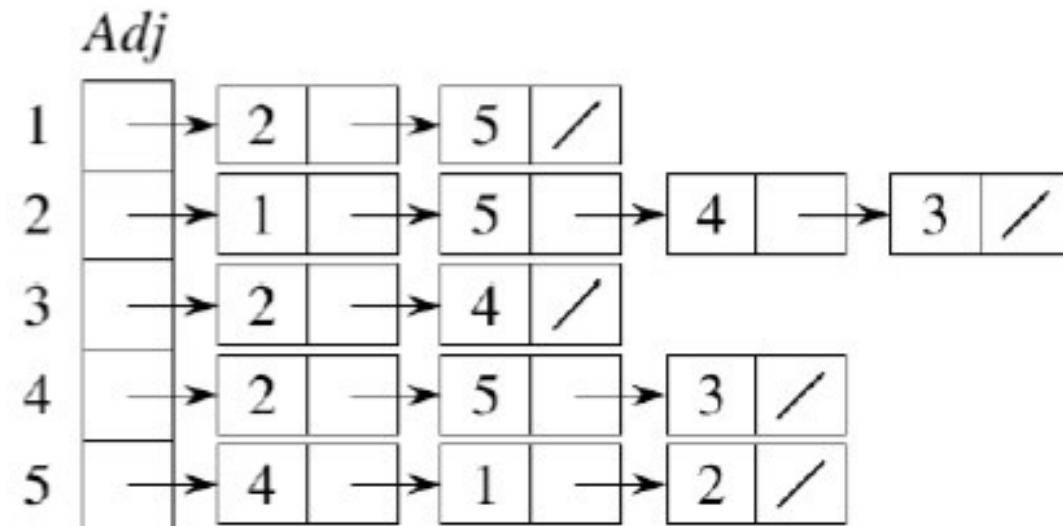
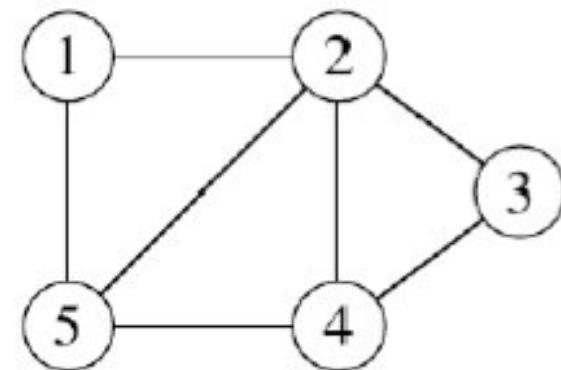


Graph Applications

- Graphs can be used to:
 - determine if one node in a network is connected to all the others
 - map out multiple course prerequisites (a solution exists if the graph is a directed graph with no cycles)
 - find the shortest route from one city to another (least cost or shortest path in a weighted graph)

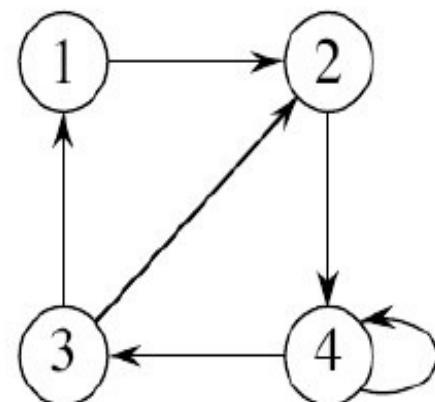
Adjacency Lists

- Array Adj of $|V|$ lists per vertex.
- List of vertex u has all vertices v such that $(u,v) \in E \Rightarrow$ works both for directed and undirected graphs.
- We denote the array as attribute $G.\text{adj} \Rightarrow$ we use the notation $G.\text{Adj}[u]$.



Adjacency Lists

- We can put weights in the list in case the edges have weights.
Weight $w:E \rightarrow \mathbb{R}$.
- Weights are later used for spanning trees and shortest path.
- Space: $\Theta(V + E)$
- Time: to list all vertices adjacent to u : $\Theta(\text{degree}(u))$.
- Time: to determine whether $(u, v) \in E$: $\Theta(\text{degree}(u))$.

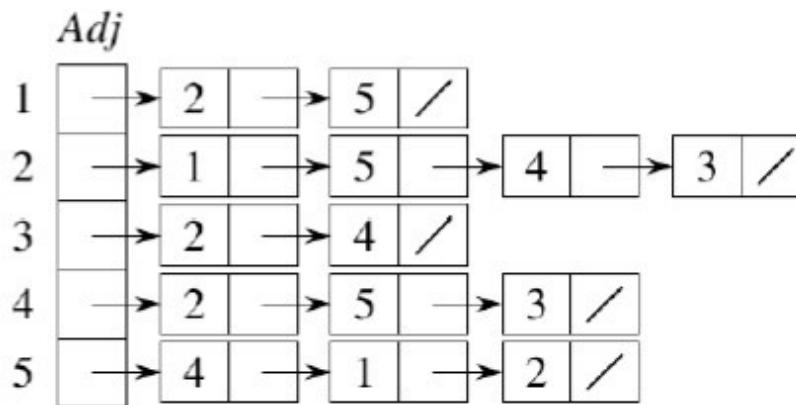


<i>Adj</i>	
1	→ 2 /
2	→ 4 /
3	→ 1 → 2 /
4	→ 4 → 3 /

Adjacency Matrix

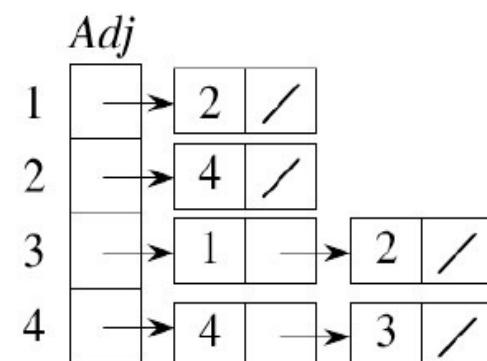
$|V| \times |V|$ matrix $A = (a_{ij})$ with

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



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

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



- **Space:** $\Theta(V^2)$
 - **Time:** to list all vertices adjacent to u : $\Theta(V)$.
 - **Time:** to determine whether $(u, v) \in E$: $\Theta(1)$.
 - We can store weights instead of bits for weighted graphs.
- ⇒ We will use both representations.



Graph attributes

Representing graph attributes:

- Graph algorithms usually maintain attributes for vertices and/or edges.
- We use $v.d$ to denote attribute d of vertex v .
- We use $(u, v).f$ to denote attribute f of edge (u, v) .

Implementing graph attributes:

- Depends on the programming language, the algorithm, and on the interaction of the rest of the program with the graph \Rightarrow no best way to implement.
- Example: Could use array $d[1\dots|V|]$ parallel to Adj in order to store vertex attributes. Vertices adjacent to u are in $\text{Adj}[u]$, store $u.d$ in array $d[u]$.
- Example: Represent vertex attributes as instance variables within a subclass of a Vertex class.

Elementary Graph Algorithms



Breadth-first search (BSF):

- Input: Graph $G = (V, E)$, either directed or undirected and a source vertex $s \in V$.
- Output: $v.d = \text{distance}$ (smallest # of edges) from s to v , for all $v \in V$.
- The textbook uses $v.\pi$ such that (u, v) is the last edge on the shortest path $s \rightarrow v$.
 - u is v 's predecessor.
 - Set of edges $\{(v.\pi, v) : v \neq s\}$ forms a tree.
- We do a generalization of the BFS, with edge weights, later. Keep it simple for now.
 - We compute only the $v.d$, not the $v.\pi$.
 - Omit colors of vertices.

Elementary Graph Algorithms



Breadth-first search (BSF):

- We send a wave out from our source s .
 - The wave first hits all vertices 1 edge from s .
 - From there, the wave hits all vertices 2 edges from s ,...
- We will use a FIFO queue Q to maintain the waveform.
 - $v \in Q$ if and only if wave hit v but has not come out of v set.

Elementary Graph Algorithms

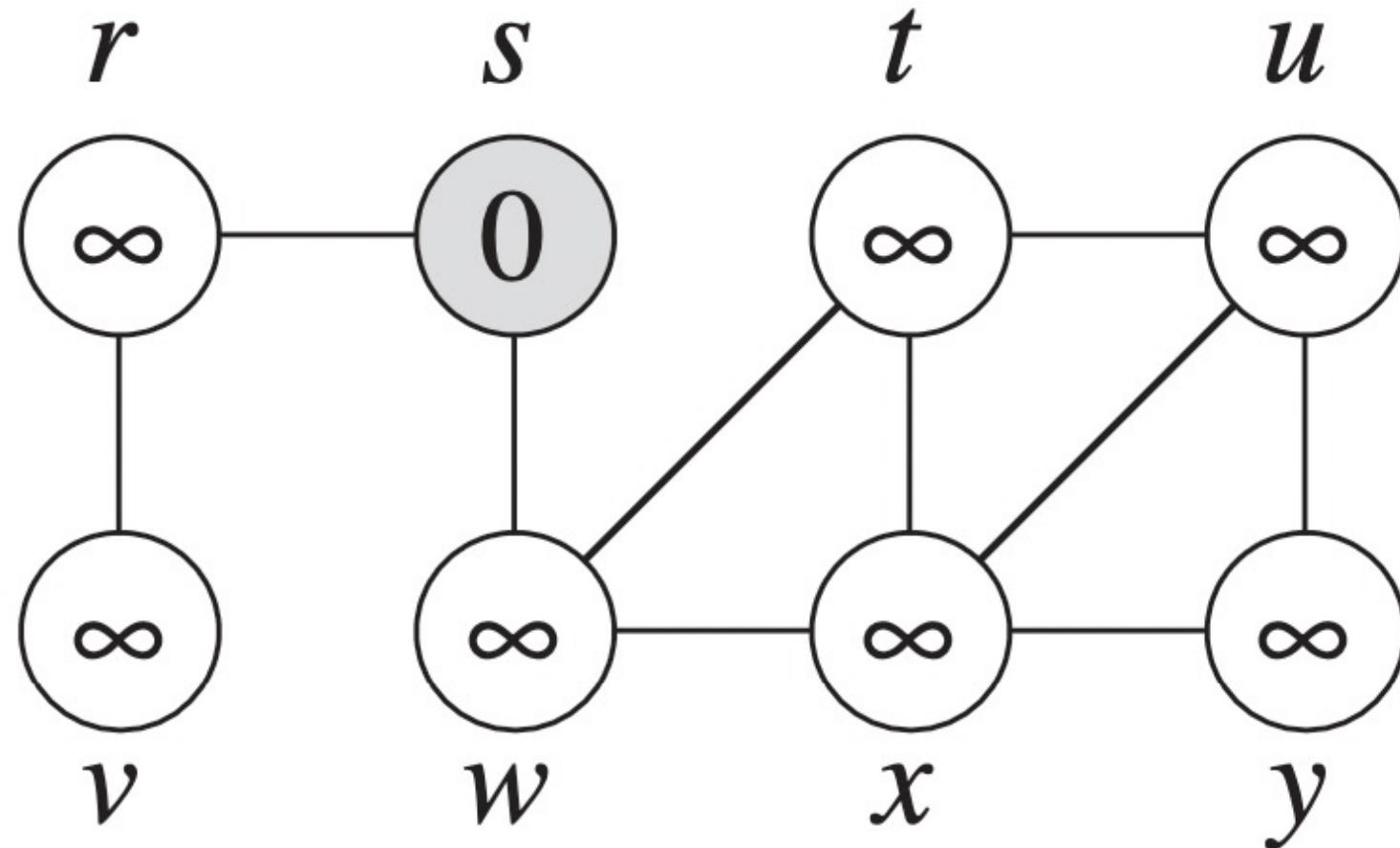


Breadth-first search (BSF):

Algorithm (BFS(V,E,s))

```
1 foreach (u ∈ (V – {s})) do
2     u.d = ∞
3 s.d = 0
4 Q = ∅
5 ENQUEUE(Q, s)
6 while (Q ≠ ∅) do
7     u = DEQUEUE(Q)
8     foreach (v ∈ G.Adj[u]) do
9         if (v.d = ∞) then
10             v.d = u.d + 1
11             ENQUEUE(Q, v)
```

Elementary Graph Algorithms



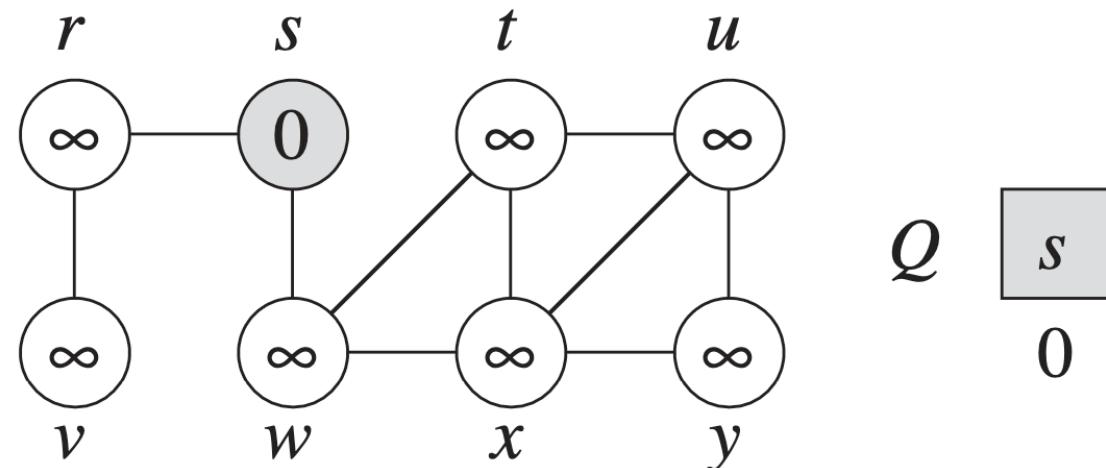
Elementary Graph Algorithms

Algorithm (BFS(V,E,s))

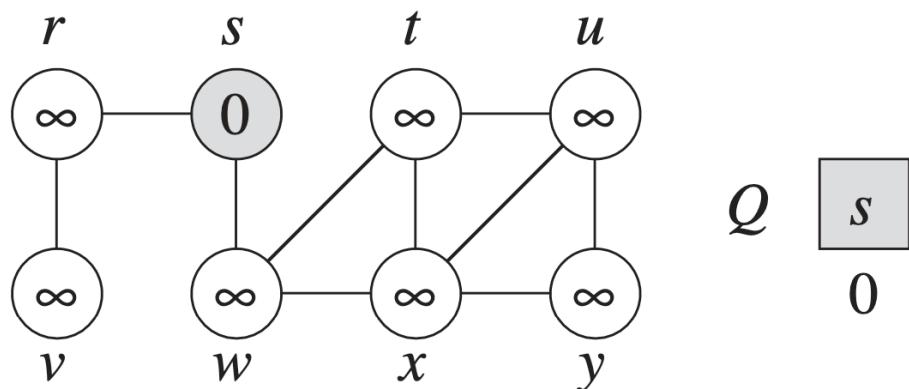
```

1 foreach (u ∈ (V – {s})) do
2   u.d = ∞
3 s.d = 0
4 Q = ∅
5 ENQUEUE(Q, s)
6 while (Q ≠ ∅) do
7   u = DEQUEUE(Q)
8   foreach (v ∈ G.Adj[u]) do
9     if (v.d = ∞) then
10       v.d = u.d + 1
11       ENQUEUE(Q, v)

```



Elementary Graph Algorithms



Algorithm (BFS(V, E, s))

```

1 foreach ( $u \in (V - \{s\})$ ) do
2    $u.d = \infty$ 
3  $s.d = 0$ 
4  $Q = \emptyset$ 
5 ENQUEUE( $Q, s$ )
6 while ( $Q \neq \emptyset$ ) do
7    $u = \text{DEQUEUE}(Q)$ 
8   foreach ( $v \in G.\text{Adj}[u]$ ) do
9     if ( $v.d = \infty$ ) then
10        $v.d = u.d + 1$ 
11       ENQUEUE( $Q, v$ )

```

1	1
w	r

1	2	2
r	t	x

2	2	2
t	x	v

2	2	3
x	v	u

2	3	3
v	u	y

s	w	r
0	1	1

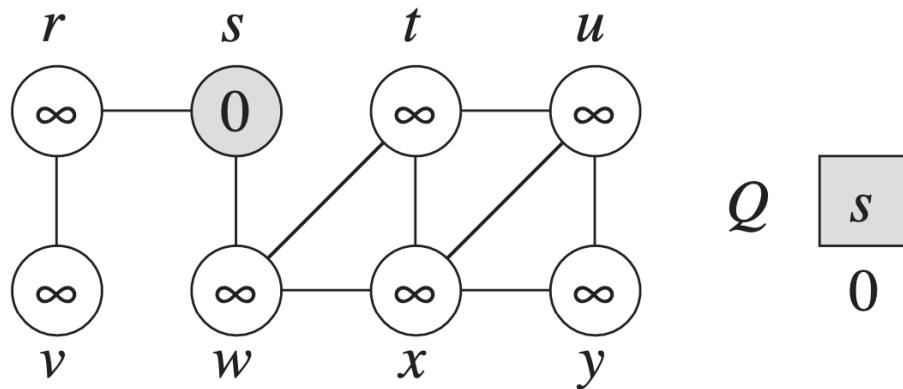
s	w	r	r	x
0	1	1	2	2

s	w	r	t	x	v
0	1	1	2	2	2

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

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

Elementary Graph Algorithms

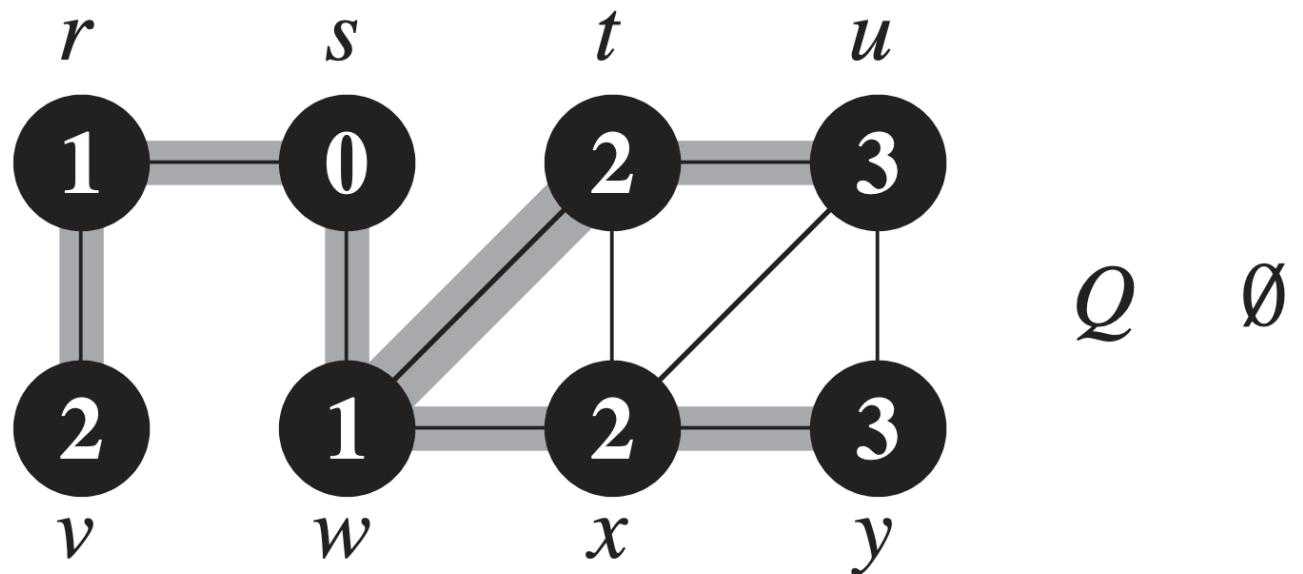


Algorithm (BFS(V, E, s))

```

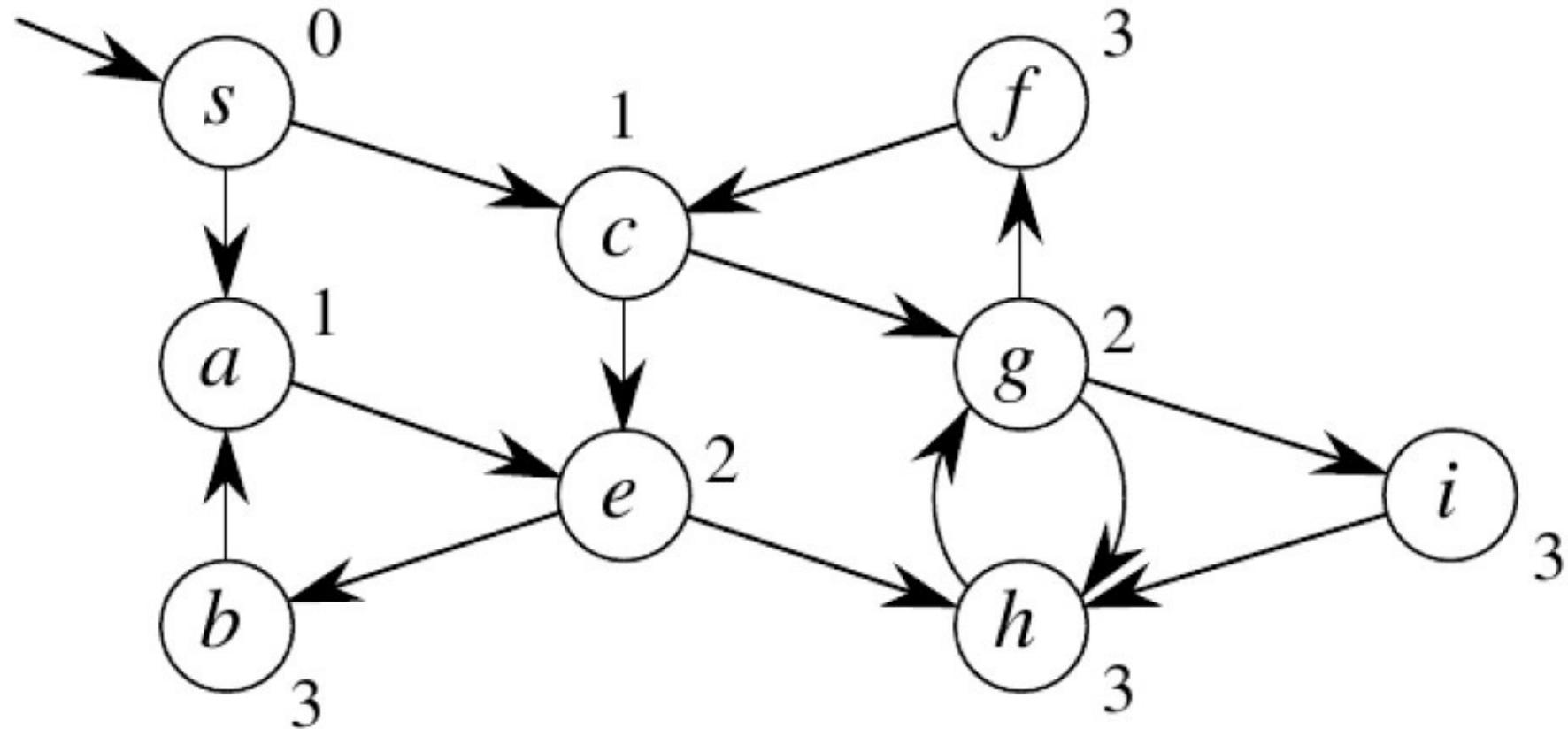
1 foreach ( $u \in (V - \{s\})$ ) do
2    $u.d = \infty$ 
3  $s.d = 0$ 
4  $Q = \emptyset$ 
5 ENQUEUE( $Q, s$ )
6 while ( $Q \neq \emptyset$ ) do
7    $u = \text{DEQUEUE}(Q)$ 
8   foreach ( $v \in G.\text{Adj}[u]$ ) do
9     if ( $v.d = \infty$ ) then
10        $v.d = u.d + 1$ 
11       ENQUEUE( $Q, v$ )

```



Elementary Graph Algorithms

Example: directed graph



Elementary Graph Algorithms



Breadth-first search (continued):

- We can show that Q consists of vertices with d values:
 $i \ i \ i \dots i \ i+1 \ i+1 \dots i+1$.
- The values assigned to vertices are monotonically increasing over time, since each vertex gets a finite d value at most once.
- See the book for the actual (a bit trickier) proof.
- BFS may not reach all vertices.
- **Time:** $O(V + E)$
 - $O(V)$ because every vertex is enqueued at most once.
 - $O(E)$ because every vertex is dequeued at most once and we then examine (u, v) only when u is dequeued \Rightarrow every edge examined at most once if directed, at most twice if undirected.



Elementary Graph Algorithms

Depth-first search:

Input: $G = (V, E)$, directed or undirected, No source vertex is given.

Output: 2 **timesteps** on each vertex:

- $v.d$ = **discovery time**.
- $v.f$ = **finishing time**.

These timestamps will be useful for other algorithms later on (see textbook for computation of $v.\pi$).

- We will methodically explore every edge.
 - We will start over from different vertices as necessary.
- We will explore a vertex, as soon as we discover it.
 - Unlike BFS, which puts a vertex on a queue so that we explore from it later.

Elementary Graph Algorithms



Depth-first search (continued):

- Every vertex has a **color** as DFS progresses.
 - *WHITE* = undiscovered
 - *GRAY* = discovered, not finished (not done exploring from it).
 - *BLACK* = finished (have found everything reachable from it).
- Discovery and finishing times:
 - Unique integers from 1 to $2|V|$.
 - For all v , we have $v.d < v.f$.
$$\Rightarrow 1 \leq v.d < v.f \leq 2|V|.$$

Algorithm (DFS):

- We use a global time stamp *time*.
- We use two routines $\text{DFS}(G)$ and $\text{DFS-VISIT}(G,u)$.
- $\text{DFS}(G)$ ensures that every vertex is visited.
- $\text{DFS-VISIT}(G,u)$ does the recursive depth-first exploration.

Elementary Graph Algorithms



Algorithm (DFS(G))

```
1 foreach (u ∈ G.V) do
2   u.color = WHITE
3 time = 0
4 foreach (u ∈ G.V) do
5   if(u.color = WHITE) then
6     DFS-VISIT (G,u)
```

Algorithm (DFS-VISIT(G,u))

```
1 time = time + 1
2 u.d = time
3 u.color = GRAY
4 foreach (v ∈ G.Adj[u]) do
5   if(v.color = WHITE) then
6     DFS-VISIT (v)
7 u.color = BLACK
8 time = time + 1
9 u.f = time
```

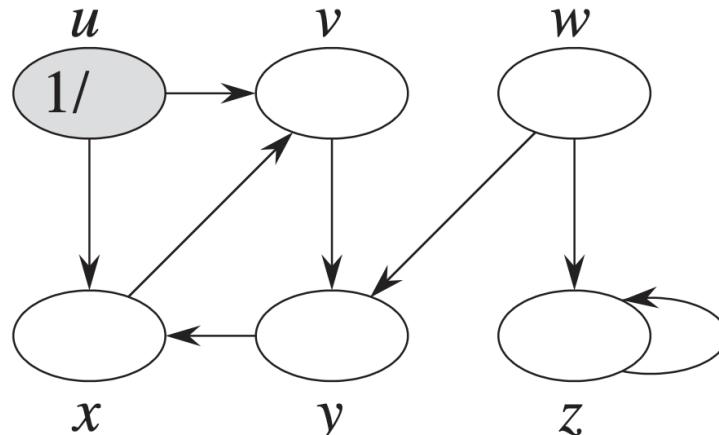
Elementary Graph Algorithms

Algorithm (DFS-VISIT(G, u))

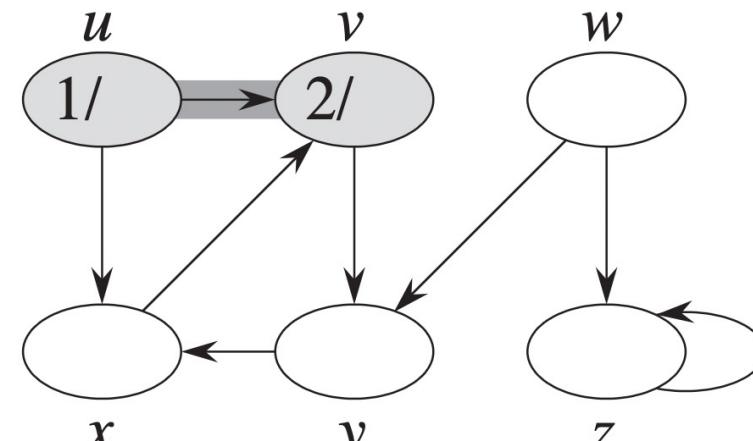
```

1 time = time + 1
2 u.d = time
3 u.color = GRAY
4 foreach (v ∈ G.Adj[u]) do
5   if(v.color = WHITE) then
6     DFS-VISIT(v)
7 u.color = BLACK
8 time = time + 1
9 u.f = time

```

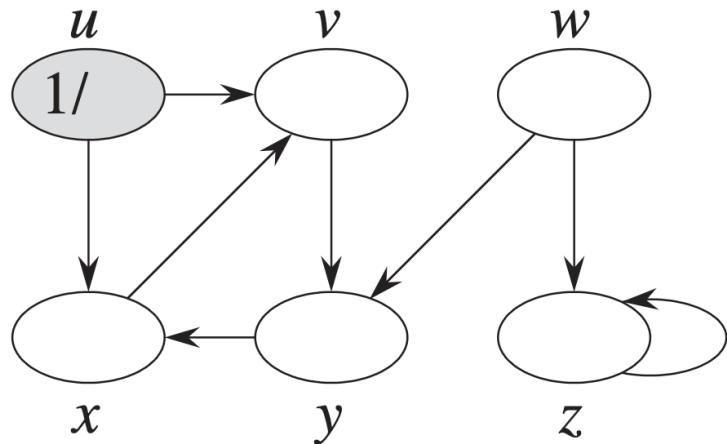


(a)

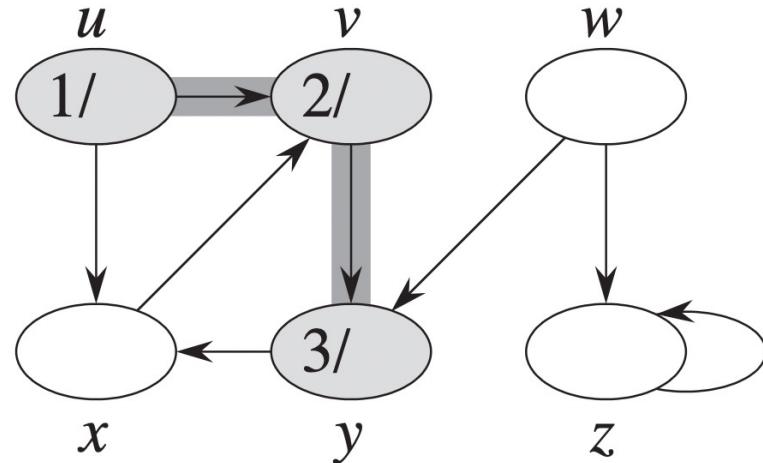


(b)

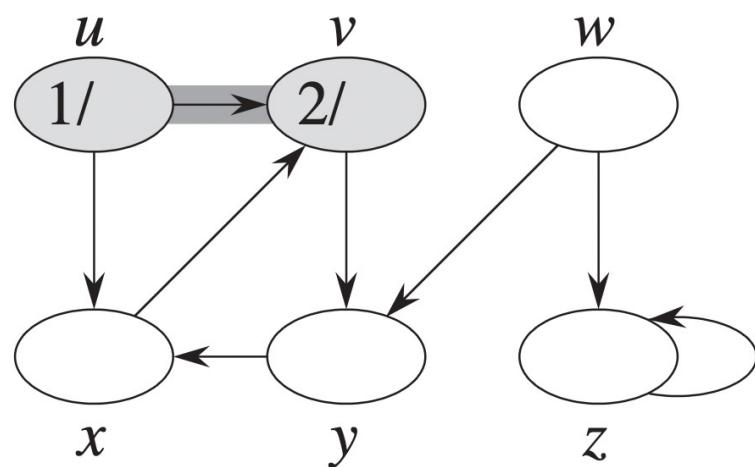
Elementary Graph Algorithms



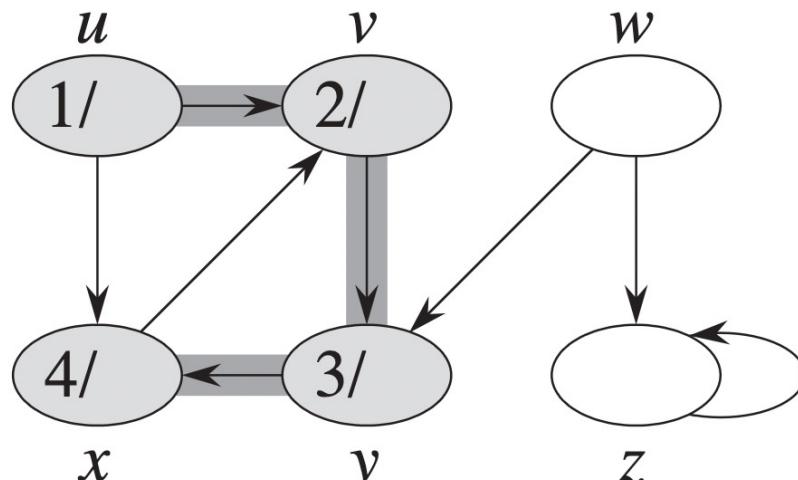
(a)



(c)

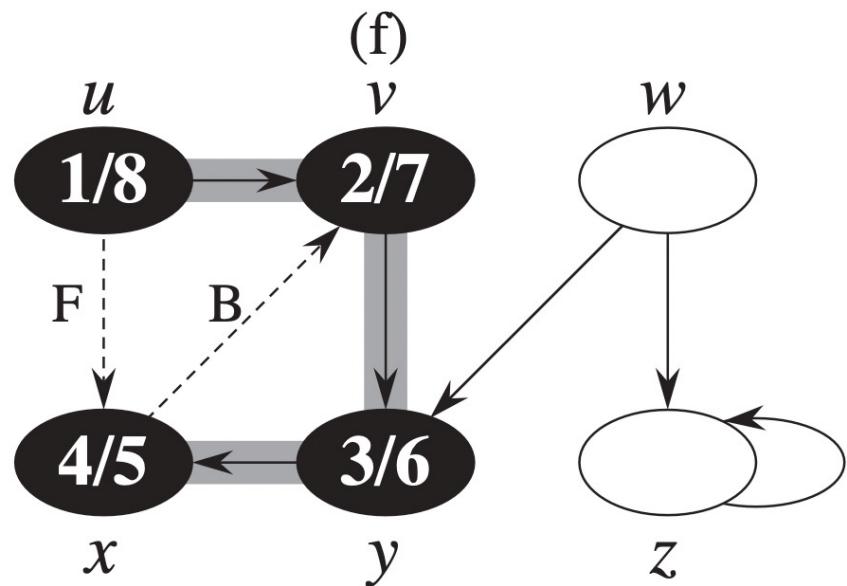
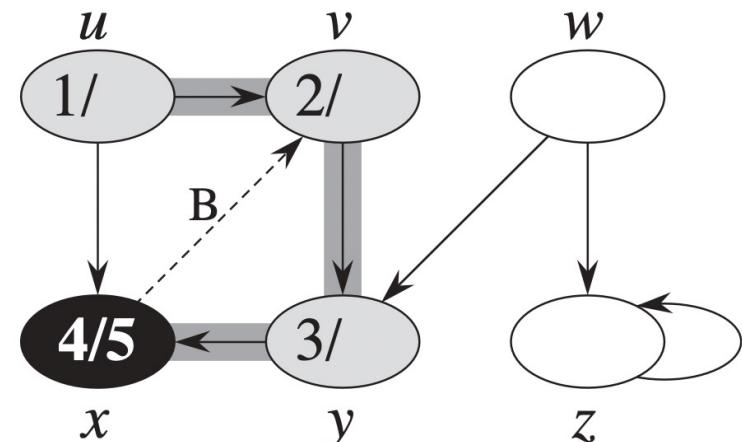


(b)

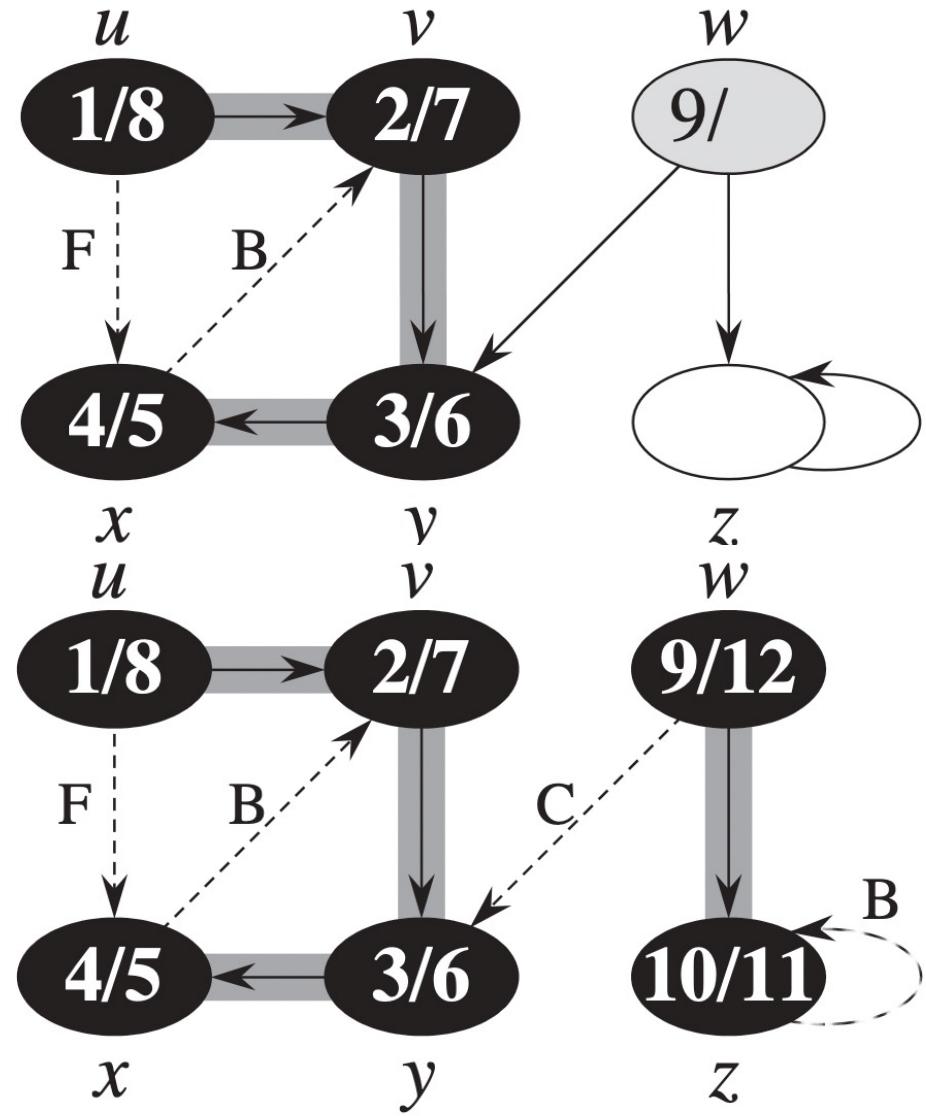


(d)

Elementary Graph Algorithms



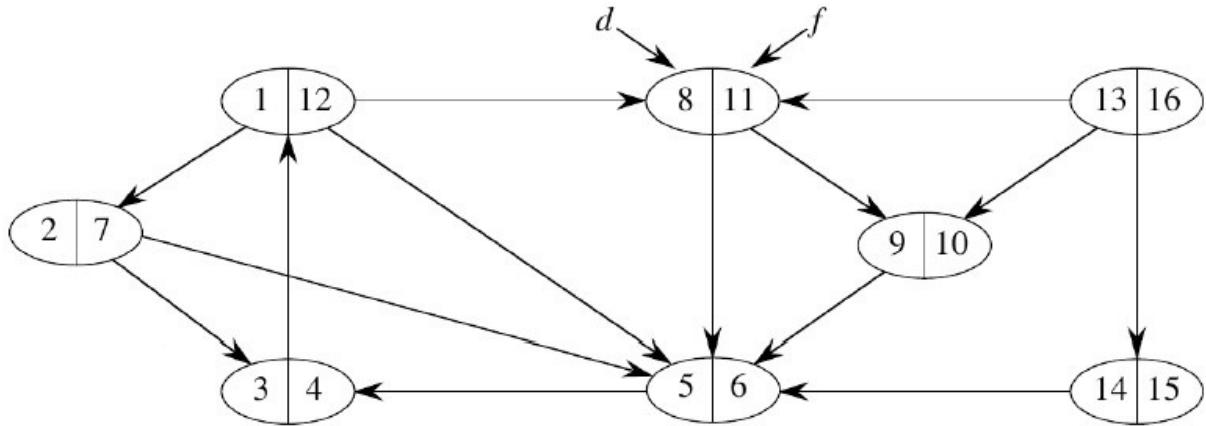
(j)



(p)

Elementary Graph Algorithms

Example: directed graph (2)



- **Time:** $\Theta(V + E)$.
 - Similar to the BFS analysis.
 - Θ , not just O , since it is guaranteed that every vertex and edge is examined.
- DFS does form a **depth-first forest** which is comprised of > 1 **depth-first trees**.
- Each tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.



Elementary Graph Algorithms

Theorem (Parenthesis theorem)

For all u, v , exactly one of the following holds:

- ① $u.d < u.f < v.d < v.f$ or $v.d < v.f < u.d < u.f$ (i.e., the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint) and neither of u and v is a descendant of the other.
- ② $u.d < v.d < v.f < u.f$ and v is a descendant of u .
- ③ $v.d < u.d < u.f < v.f$ and u is a descendant of v .

So $u.d < v.d < u.f < v.f$ cannot happen.



Elementary Graph Algorithms

Corollary

v is a proper descendant of u if and only if $u.d < v.d < v.f < u.f$.

Theorem

v is a descendant of u if and only if at time $u.d$, there is a path $u \rightsquigarrow v$ consisting of only white vertices.

(Except for u , which was just colored gray.)

Classification of edges:

Tree edge: in depth-first forest. Found by exploring (u, v) .

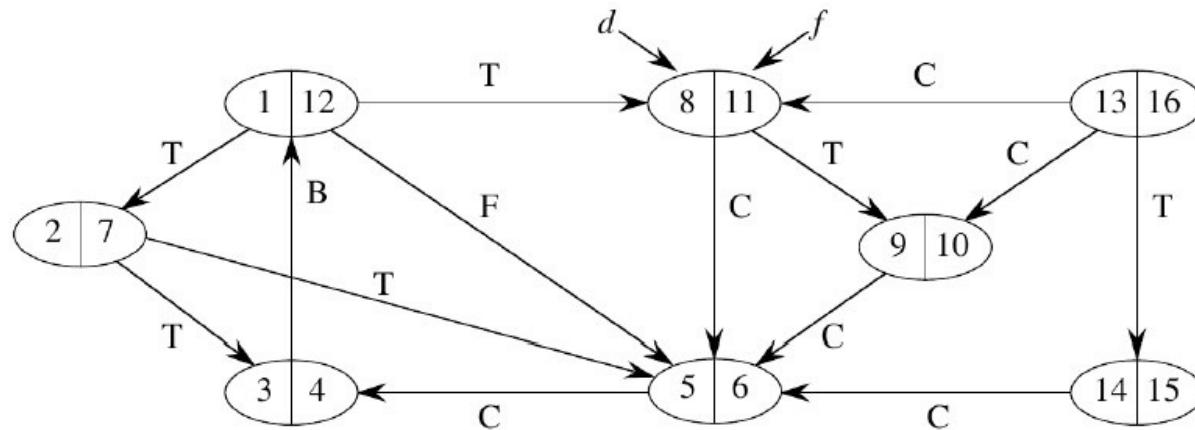
Back edge: (u, v) , where u is a descendant of v .

Forward edge: (u, v) , where v is a descendant of u , but not a tree edge.

Cross edge: any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

Elementary Graph Algorithms

Example: directed graph (2)

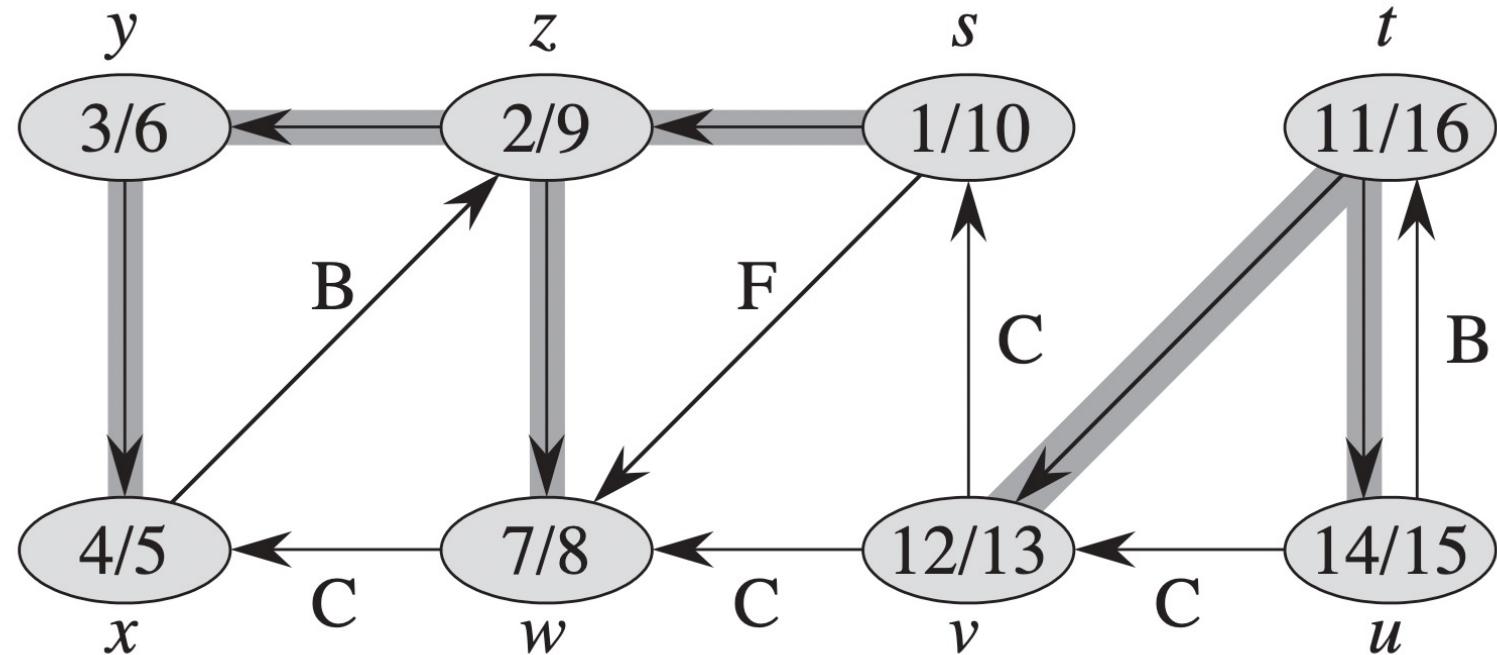


- In an undirected graph, there may be some ambiguity since (u, v) and (v, u) are the same edge.
- We classify by the first type above that matches.

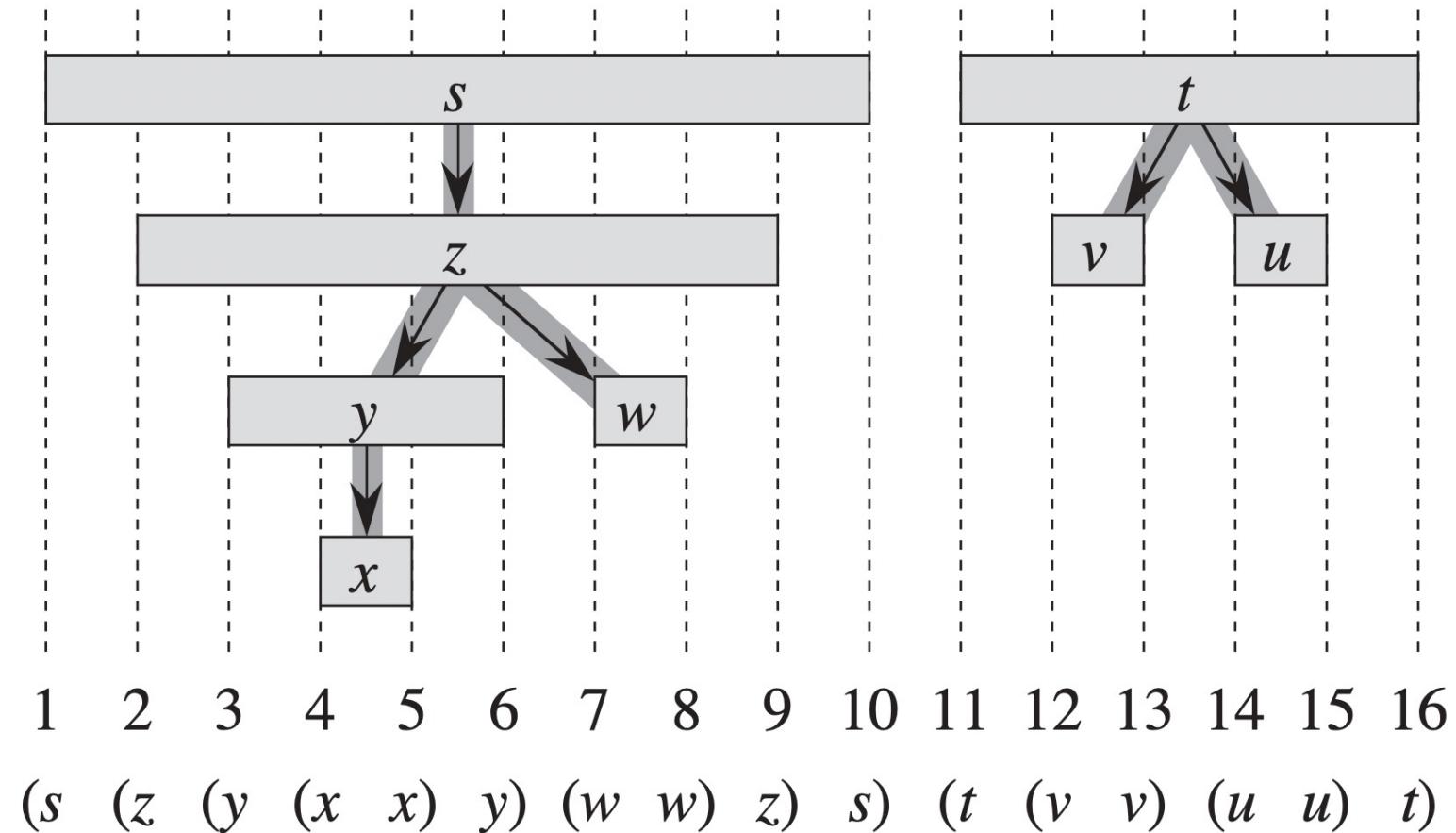
Theorem

In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges. (see textbook for proof)

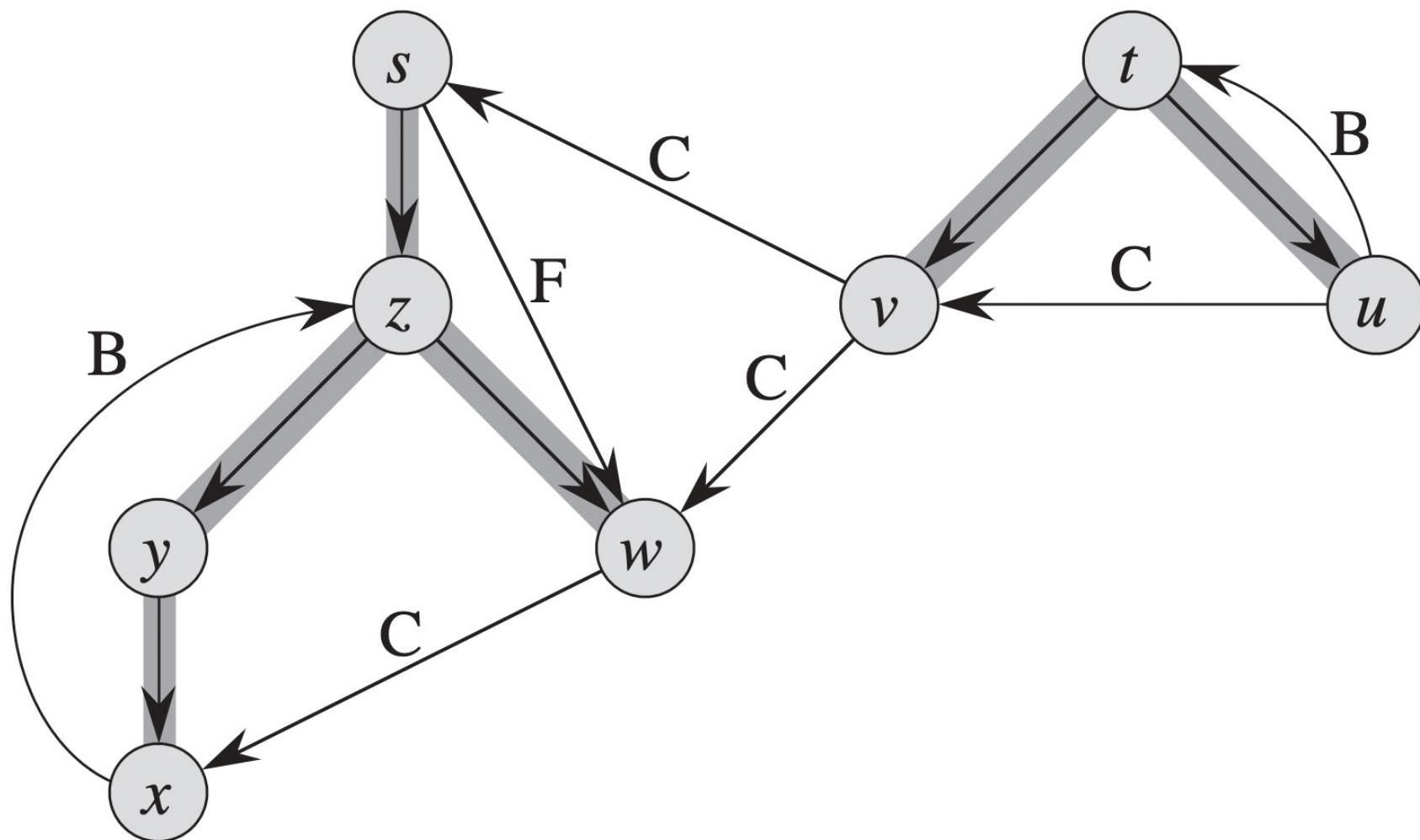
Elementary Graph Algorithms



Elementary Graph Algorithms



Elementary Graph Algorithms

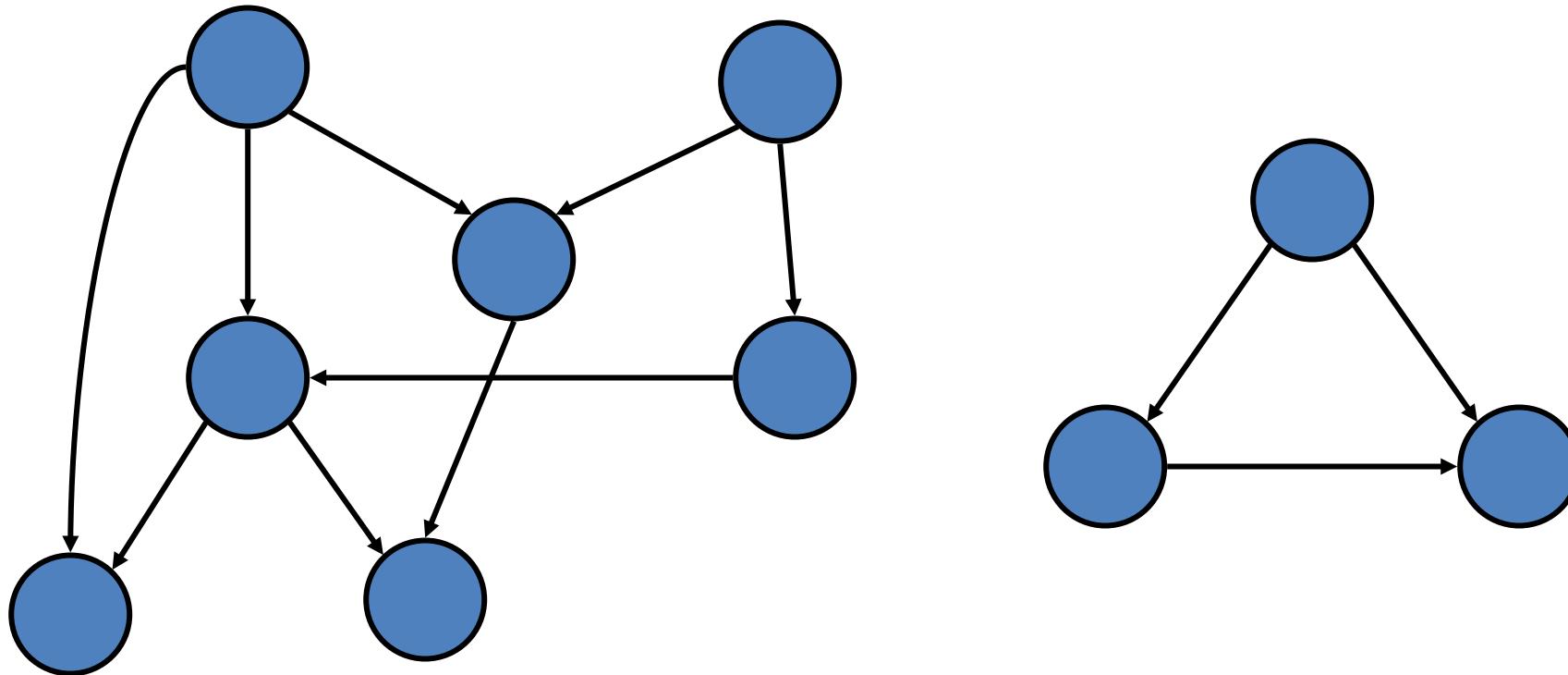




Graph's algorithm

Directed Acyclic Graphs

- A *directed acyclic graph* or **DAG** is a directed graph with no directed cycles:



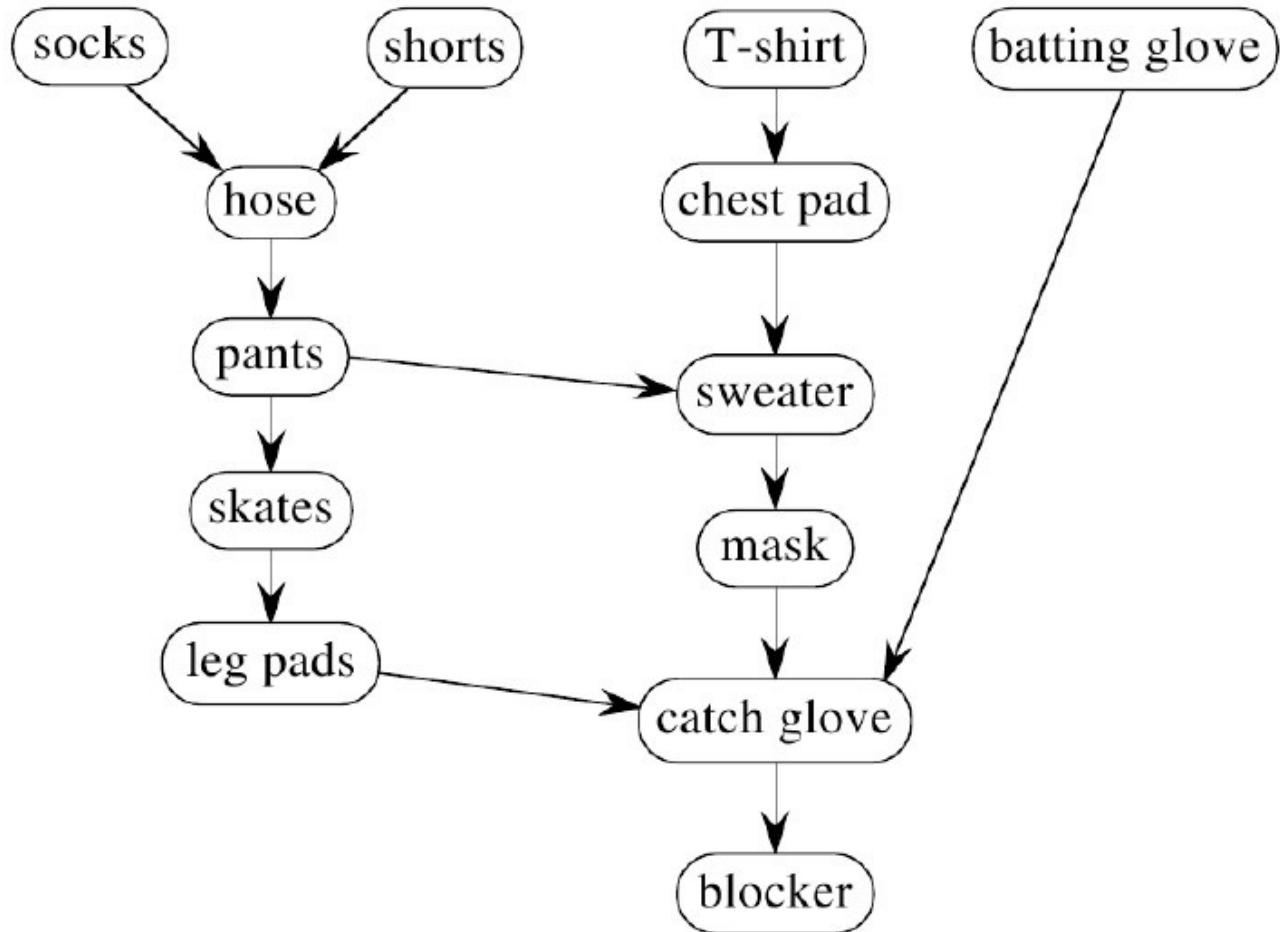
Topological Sort



- ***Topological sort*** of a DAG:
 - Linear ordering of all vertices in graph G such that vertex u comes before vertex v if edge $(u, v) \in G$.
 - Real-world example: getting dressed

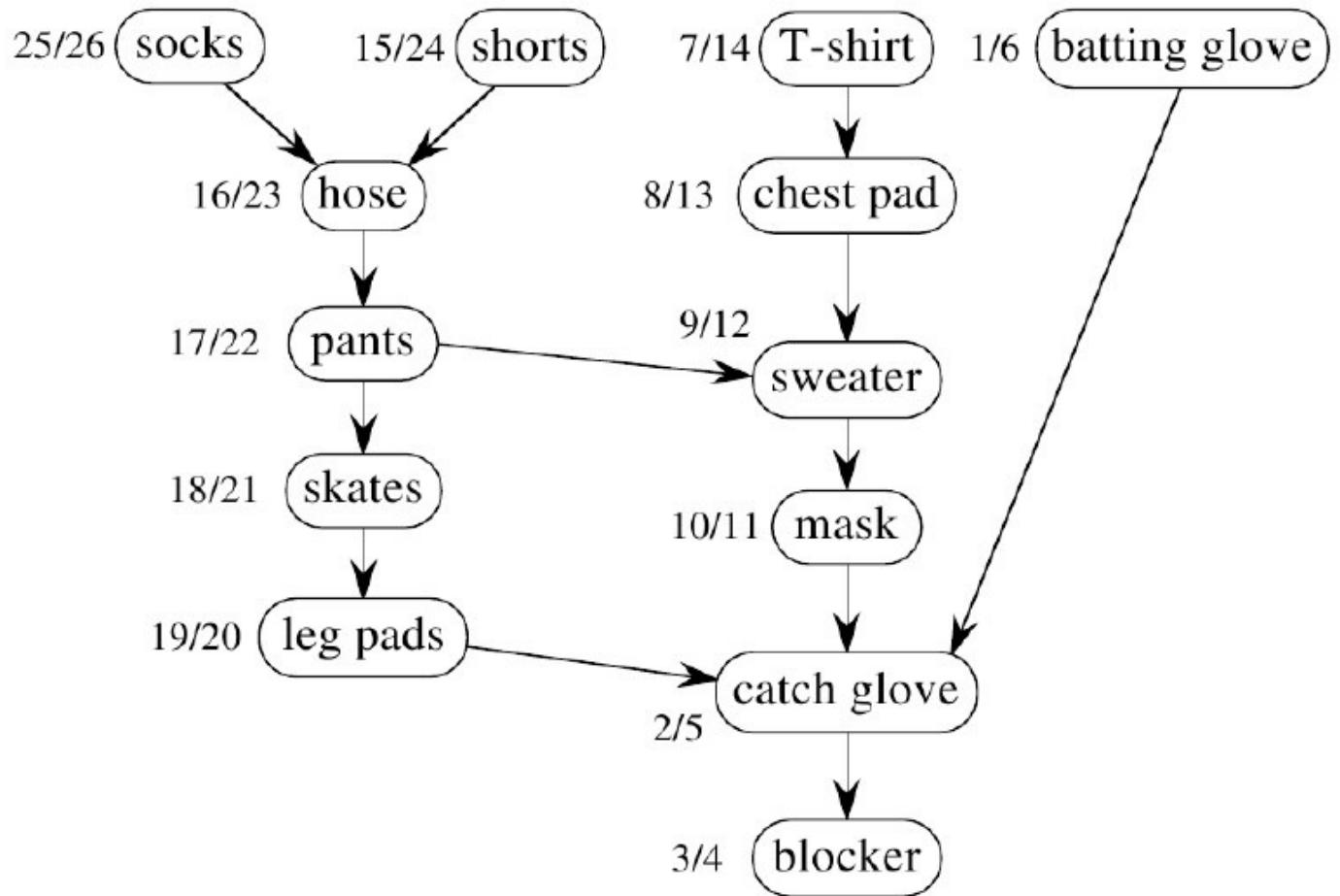
Graphs

Example:



Graphs

Example:

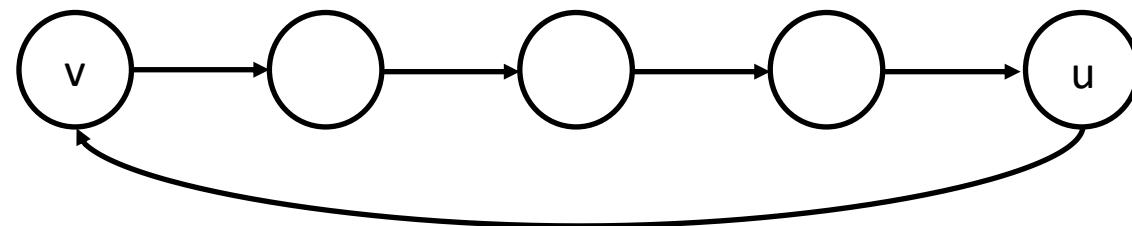


Graphs

- A directed graph G is acyclic if and only if a DFS of G yields no back edges.

- Proof:

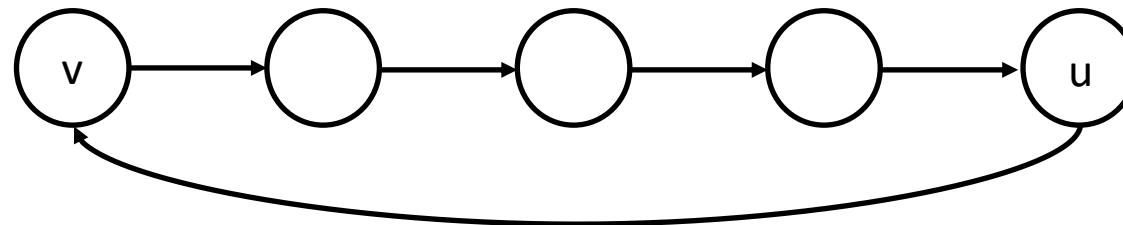
- We need to show that back edge is a cycle.
- Suppose there is a back edge (u, v) . Then v is an ancestor of u in the DFS.



- Therefore, there is a path $v \rightarrow u$, so $v \rightarrow v$ is a cycle.
- ✓ Suppose G contains a cycle c . Let v be the first vertex discovered in c , and let (u, v) be the preceding edge in c .

Graphs

- A directed graph G is acyclic if and only if a DFS of G yields no back edges.



- Therefore, there is a path $v \rightarrow u$, so $v \rightarrow v$ is a cycle.
 - ✓ Suppose G contains a cycle c . Let v be the first vertex discovered in c , and let (u, v) be the preceding edge in c .
 - ✓ At time $v.d$ vertices of c form a white path $v \rightarrow u$ (since v is the first vertex discovered in c).
 - ✓ By the white-path theorem, u is a descendant of v in the depth-first forest.
 - ✓ Therefore, (u, v) is a back edge.



Topological-sort

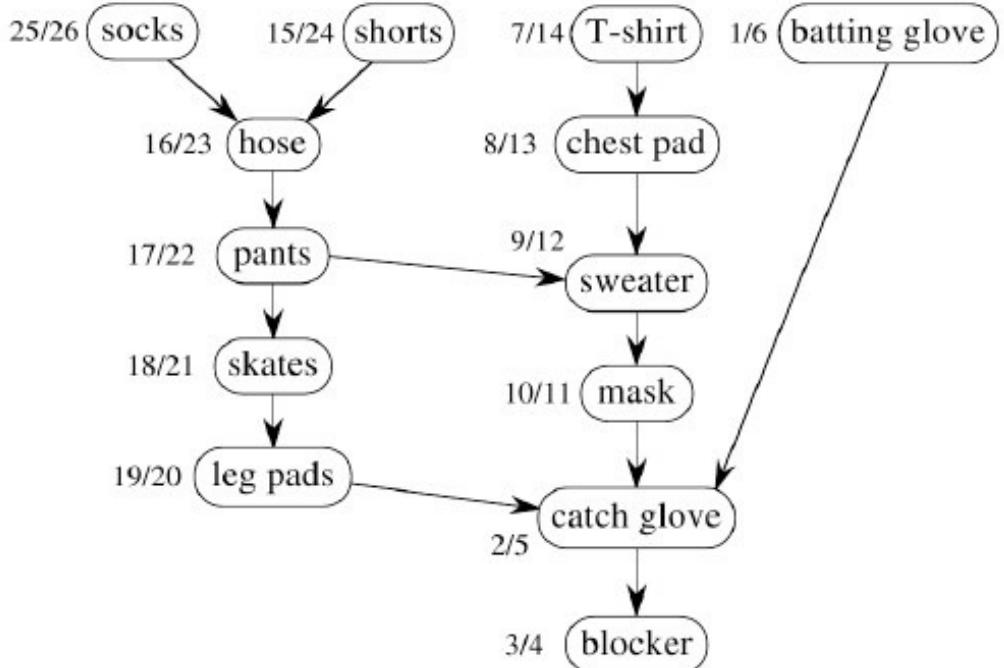
- Topological sort of a DAG: a linear ordering of vertices such that if $(u, v) \in E$, then u appear somewhere before v .
- We call $\text{DFS}(G)$ to compute the finishing time $v.f$ for all $v \in G.V$.
- We output all vertices in order of decreasing finishing times.
- We do not need to sort by finishing time.
 - We can just output the vertices as they are finished and understand that we want the reverse of this list.
 - Another option is to put them in front of a liked list as they are finished.
After the DFS, the list contains the vertices in topological sorted order.
- Finished referrers to the $\text{DFS-VISIT}(G, u)$ algorithm and the assignment of the finishing time $u.f$.
- Time: $\Theta(V + E)$

Graphs

Example (continued):

- Order:

- 26 socks
- 24 shorts
- 23 hose
- 22 pants
- 21 skates
- 20 leg pads
- 14 t-shirt
- 13 chest pad
- 12 sweater
- 11 mask
- 6 batting glove
- 5 catch glove
- 4 blocker



Graphs

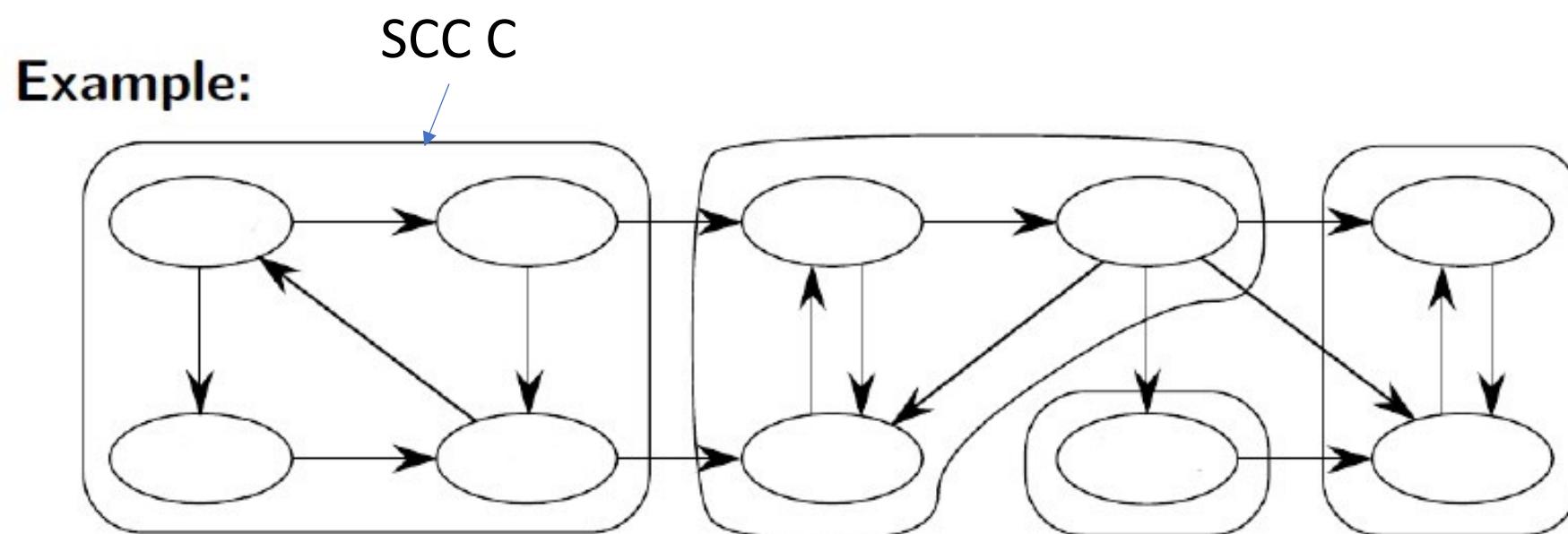


Correctness:

- We just need to show: if $(u, v) \in E$, then $v.f < u.f$.
- What are the colors of u and v when we explore (u, v) .
 - u is gray.
 - Is v also gray?
 - No, because then v would be an ancestor of u .
 - $\Rightarrow (u, v)$ is a back edge.
 - \Rightarrow contradiction of lemma (DAG has no back edges).
 - Is v white?
 - Then v becomes a descendent of u . By parenthesis theorem, $u.d < v.d < v.f < u.f$.
 - Is v black?
 - Then v is already finished. Since we exploring (u, v) , we have not yet finished u . Therefore, $v.f < u.f$.

Graphs – Strongly Connected Components (SCC)

- Given a directed graph $G = (V, E)$.
- A **strong connected component (SCC)** of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightarrow v$ and $v \rightarrow u$.





Graphs – Strongly Connected Components (SCC)

Algorithm for SCC:

- Algorithm uses G^T = transpose of G .
 - $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$.
 - G^T is G with all edges reversed.
- Can create G^T in $\Theta(V + E)$ time if using adjacency lists.

Observation:

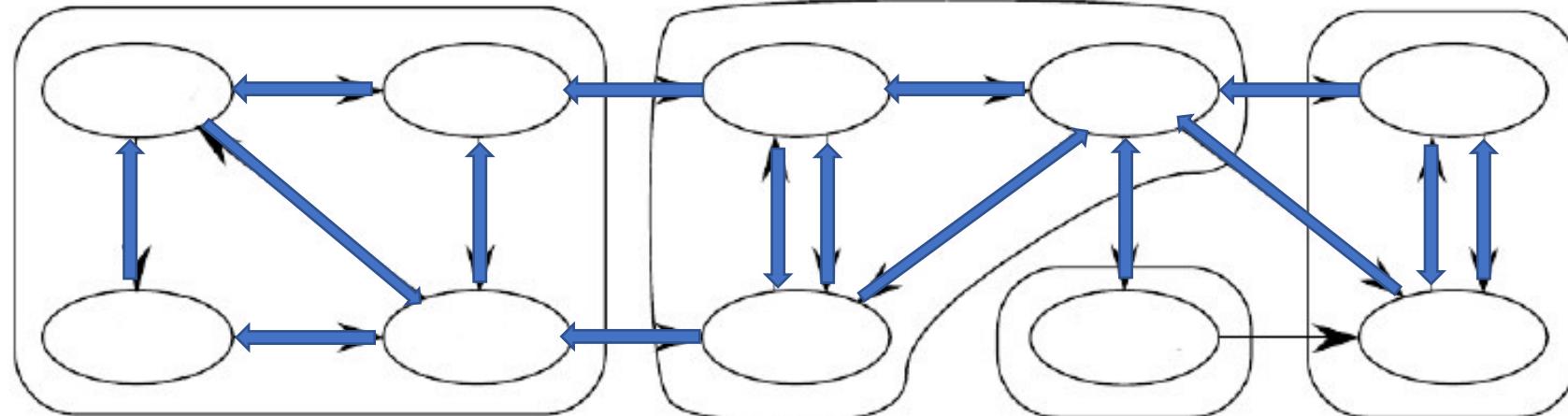
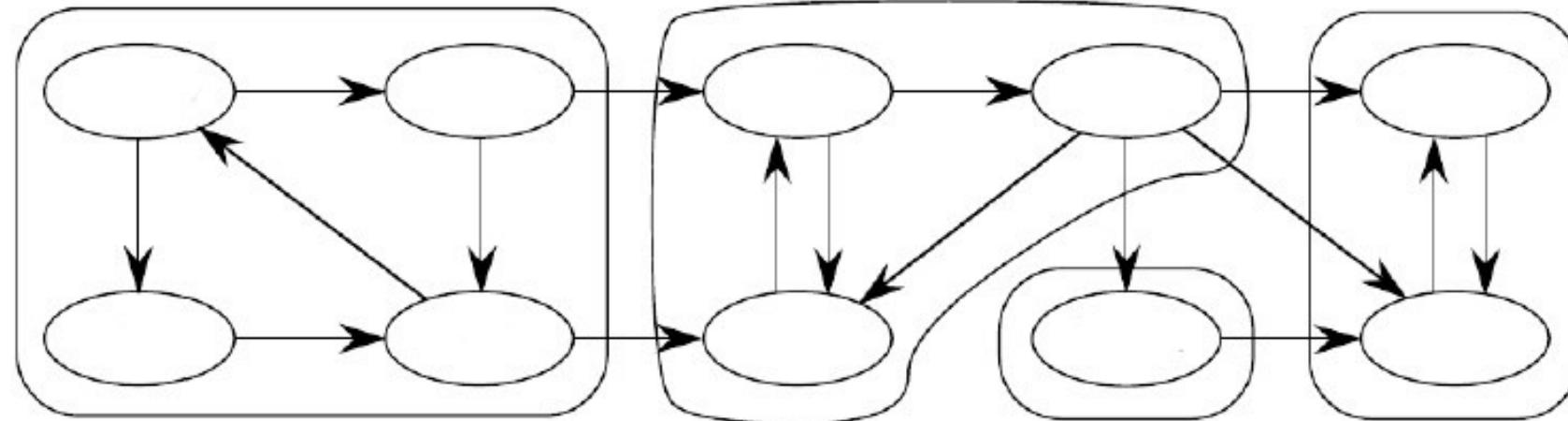
- G and G^T have the same SCC's.
- u and v are reachable from each other in G if and only if they are reachable from each other in G^T .

Component Graph:

- $G^{SCC} = (V^{SCC}, E^{SCC})$.
- V^{SCC} has one vertex from each SCC in G .
- E^{SCC} has an edge if there is an edge between the corresponding SCC's in G .

Graphs – Strongly Connected Components (SCC)

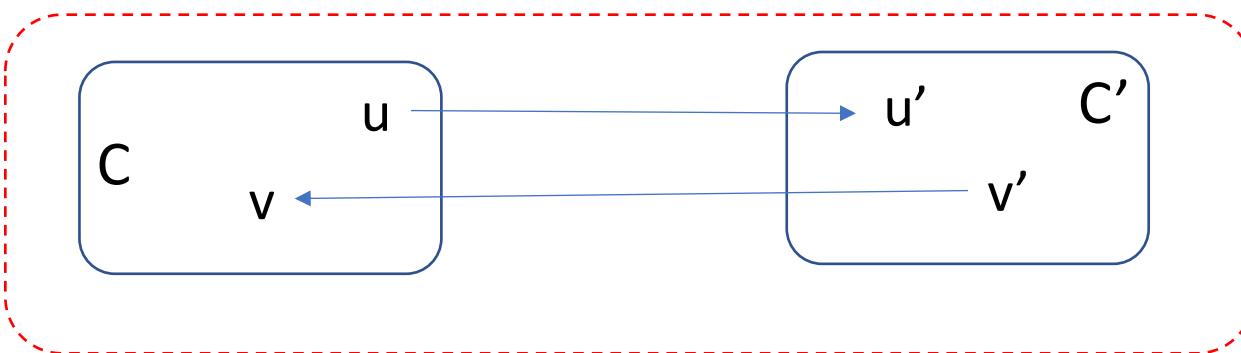
Example:





Graphs – Strongly Connected Components (SCC)

- G^{SCC} is a DAG.
- More formally, let C and C' be distinct SCC's in G , let $u, v \in C, u', v' \in C'$, and suppose there is a path $u \rightarrow u'$ in G . Then there cannot also be a path $v' \rightarrow v$ in G .
- Proof:
 - Suppose there is a path $v' \rightarrow v$ in G . Then there are path $u \rightarrow u' \rightarrow v'$ and $v' \rightarrow v \rightarrow u$ in G . Therefore, u and v' are reachable from each other, so they are not in separate SCC's.





Graphs – Strongly Connected Components (SCC)

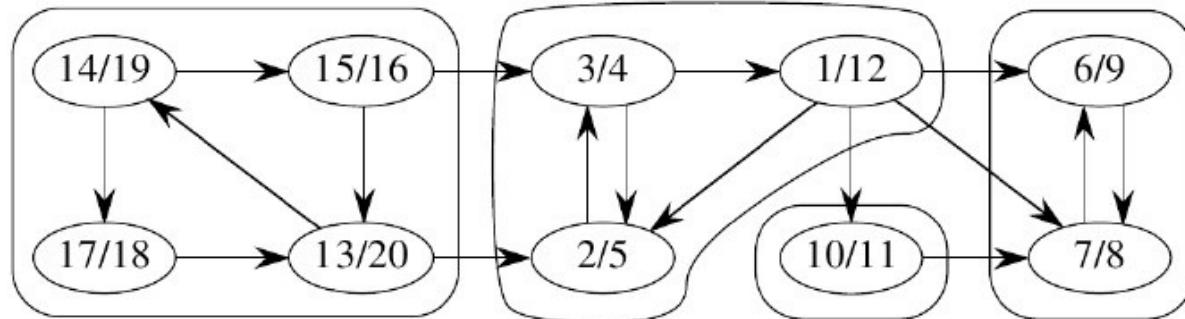
scc(G):

1. Call $\text{DFS}(G)$ to compute finishing times $u.f$ for all u .
2. Compute G^T .
3. Call $\text{DFS}(G^T)$, but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in the first DFS).
4. Output vertices in each tree of the depth-first forest formed in the second DFS as a separate SCC.

Graphs – Strongly Connected Components (SCC)

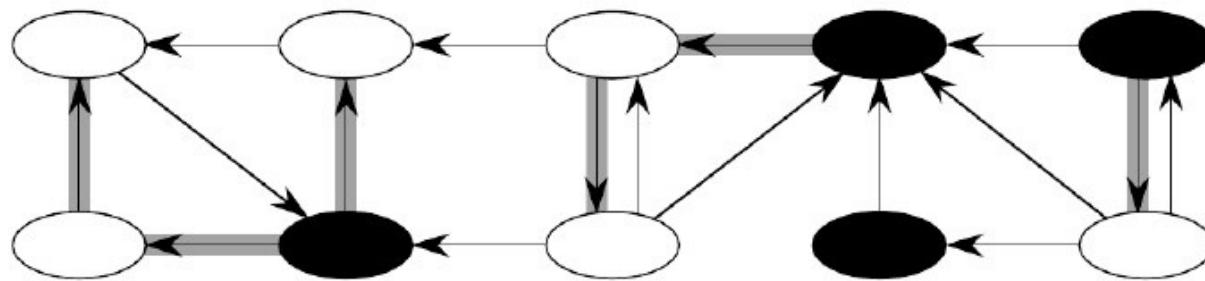
Example:

- ① Call DFS.



- ② Compute G^T .

- ③ Call DFS (roots blackened)



Time: $\Theta(V + E)$.

Graphs – Strongly Connected Components (SCC)



Idea:

- We are considering vertices in the second DFS in decreasing the order of finishing times from the first DFS.
- We are visiting the vertices of the component graph in topological sort order.
- To prove that it works, we must deal with two notational issues:
 - We will be discussing $u.d$ and $u.f$. These will always refer to the first DFS.
 - We extend the notation for d and f to sets of vertices $U \subseteq V$:
 - $d(U) = \min_{u \in U} \{u.d\}$
 - $f(U) = \max_{u \in U} \{u.f\}$



Graphs – Strongly Connected Components (SCC)

- Let C and C' are distinct SCC's in $G = (V, E)$. If $u \in C$ and $v \in C'$ and if there is an edge $(u, v) \in E$, SCC's are connected as $C \rightarrow C'$ but not $C' \rightarrow C$!
 - DFS will allow $V' \in C'$ to be fully explored before all $V \in C$.
$$\therefore f(C) > f(C')$$
 - If $d(C) < d(C')$, all vertices in C and C' are not all discovered except $x \in C$.
 - If there is a path from $C \rightarrow C'$, then all $V' \in C'$ will be explored before x because DFS runs on x .
$$\therefore x.f = f(C) > f(C')$$



Graphs – Strongly Connected Components (SCC)

- If $d(C) > d(C')$, all vertices in C and C' are not all discovered except $y \in C'$.
 - If y is the first discovered vertex, all vertices $V' - y \in C'$ are white $\Rightarrow y.f = f(C')$.
 - All vertices, $V \in C$, are white @ $y.d$ time.
 - If $C \rightarrow C'$ exists, then all $V \in C$ are white and they are not reachable from y .
 - Any $x \in C$ will be discovered after $f(C') = y.f$.
$$\therefore f(C) > f(C')$$
- Let C and C' are distinct SCC's in $G = (V, E)$. If $u \in C$ and $v \in C'$ and if there is an edge $(u, v) \in E^T$, there is an edge $(v, u) \in E$.
$$\therefore f(C) < f(C')$$



Graphs – Strongly Connected Components (SCC)

- There are no edges from C to C' in G^T , since $f(C) > f(C')$ for all $C' \neq C$.
 - Therefore, DFS will visit only vertices in C .
 - This means that the depth-first tree rooted at x contains exactly the vertices of C .
 - The next root chosen in the second DFS is in SCC C' such that $f(C')$ is maximum over all SCC's other than C . The DFS visits all vertices in C' , but the only edges out of C' go to C , which we have already visited.
- Therefore, the only tree edges will be to vertices in C' .
- We can continue the process.
- Each time we choose a root for second DFS, it can reach only
 - Vertices in its SCC – get tree edges to these,
 - Vertices in SCC's already visited in second DFS – get no tree edges to these.
- We are visiting vertices of $(G^T)^{SCC}$ in reverse of topologically sorted order.