

Graphs-1
Data Structure
Divyashikha Sethia
divyashikha@dtu.ac.in

What is a Graph?

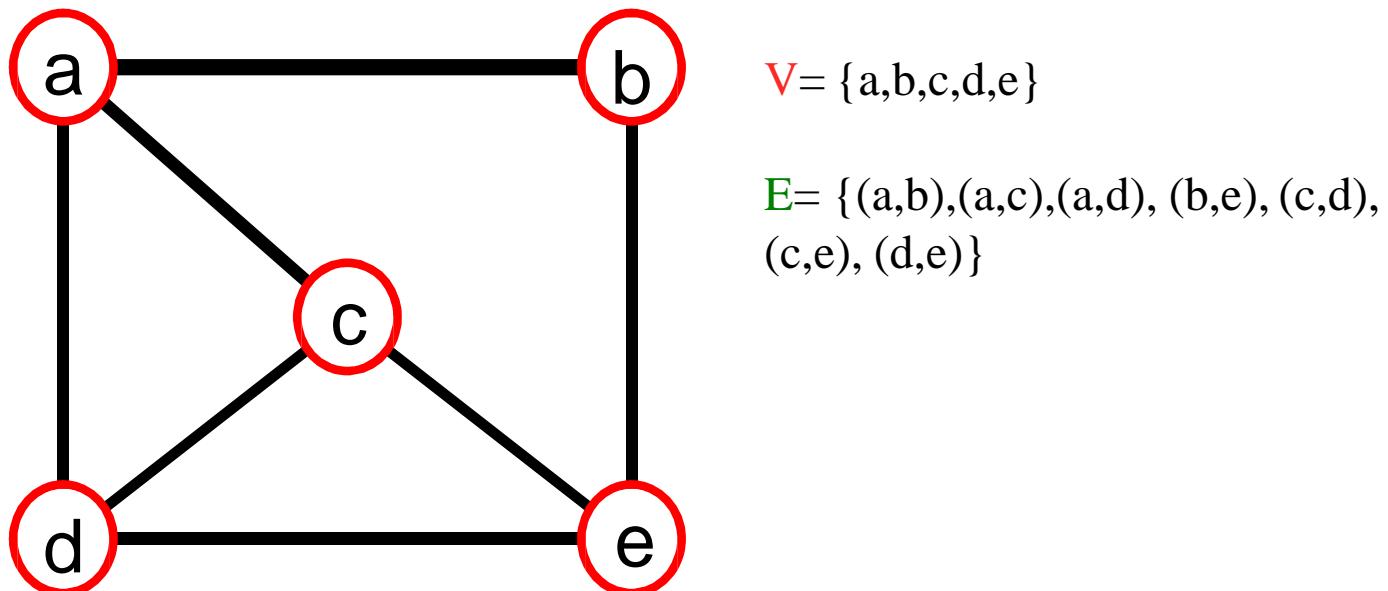
A graph $G = (V, E)$ is composed of:

V : set of **vertices**

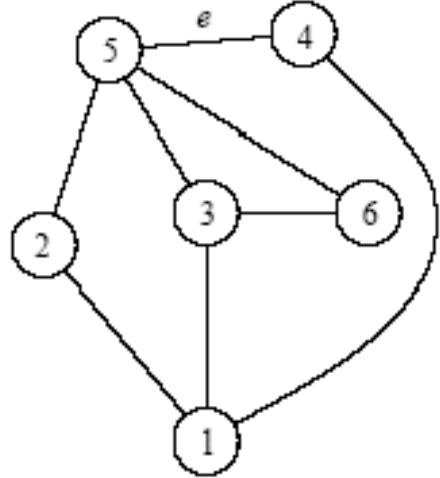
E : set of **edges** connecting the **vertices** in V

An **edge** $e = (u, v)$ is a pair of **vertices**

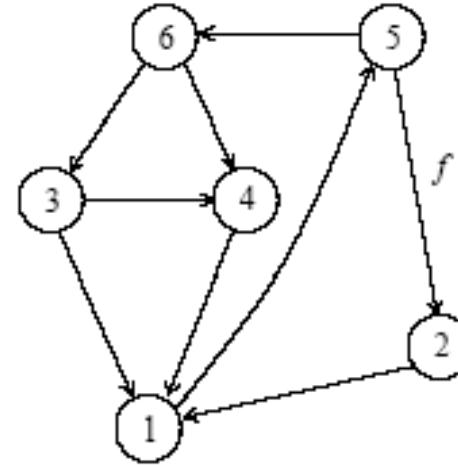
Example:



Directed v/s Undirected graphs



(a)



(b)

(a) An **undirected graph** is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$

(b) A **directed graph** is one in which each edge is a directed pair of vertices, $\langle v_0, v_1 \rangle \neq \langle v_1, v_0 \rangle$

Graph Terminology

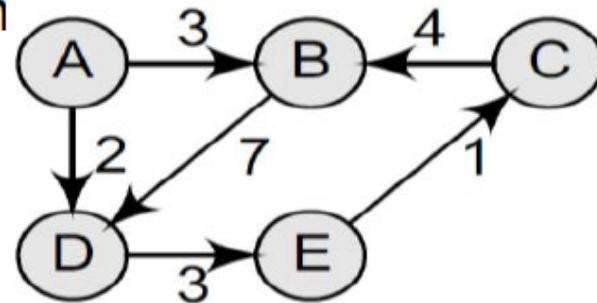
- **Adjacent nodes or neighbours** For every edge, $e = (u, v)$ that connects nodes u and v , the nodes u and v are the end-points and are said to be the adjacent nodes or neighbours.
- **Degree of a node** Degree of a node u , $\deg(u)$, is the total number of edges containing the node u .
- If $\deg(u) = 0$, it means that u does not belong to any edge and such a node is known as an **isolated node**.

Graph Terminology ...

- **Path** A path $P = \{v_0, v_1, v_2, \dots, v_n\}$, of length n from a node u to v is defined as a sequence of $(n+1)$ nodes. Here, $u = v_0$, $v = v_n$ and v_{i-1} is adjacent to v_i for $i = 1, 2, 3, \dots, n$.
- **Closed path** A path P is known as a closed path if the edge has the same end-points. That is, if $v_0 = v_n$.
- **Simple path** A simple path if all the nodes in the path are distinct with an exception that v_0 may be equal to v_n . If $v_0 = v_n$, then the path is called a *closed simple path*.
- **Cycle** A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices). Same as **closed path/circuit**

Graph Terminology...

- **Clique**
 - undirected graph $G = (V, E)$, clique is a subset of the vertex set $C \subseteq V$, such that for every two vertices in C , there is an edge that connects two vertices.
- **Labelled graph or weighted graph** A graph is said to be labelled if every edge in the graph is assigned some data. The weight of an edge denoted by $w(e)$ is a positive value which indicates the cost of traversing th



(c) Weighted graph

Directed Graphs

- A directed graph G , also known as a *digraph*, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G . For an edge (u, v) ,
- The edge begins at u and terminates at v .
- u is the predecessor of v . Correspondingly, v is the successor of u .
- Nodes u and v are adjacent to each other.

Directed Graph...

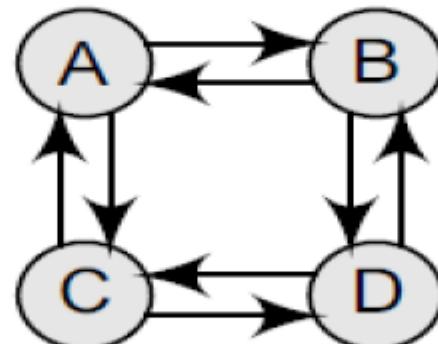
- **Out-degree of a node** The out-degree of a node u , written as $\text{outdeg}(u)$, is the number of edges that originate at u .
- **In-degree of a node** The in-degree of a node u , written as $\text{indeg}(u)$, is the number of edges that terminate at u .
- **Degree of a node** The degree of a node, written as $\text{deg}(u)$, is equal to the sum of in-degree and out-degree of that node.
$$\text{deg}(u) = \text{indeg}(u) + \text{outdeg}(u).$$
- **Isolated vertex** A vertex with degree zero. Such a vertex is not an end-point of any edge.
- **Pendant vertex** (also known as leaf vertex) A vertex with degree one.

Directed Graph...

- **Cut vertex** A vertex which when deleted would disconnect the remaining graph.
- **Source** A node u is known as a source if it has a positive out-degree but a zero in-degree.
- **Sink** A node u is known as a sink if it has a positive in-degree but a zero out-degree.
- **Reachability** A node v is said to be reachable from node u , if and only if there exists a (directed) path from node u to node v .

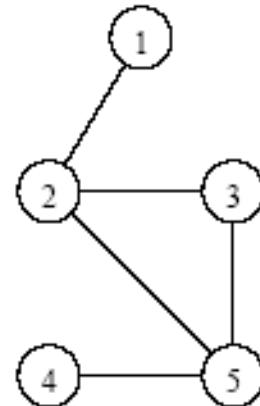
Directed Graph...

- **Strongly connected directed graph** there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.



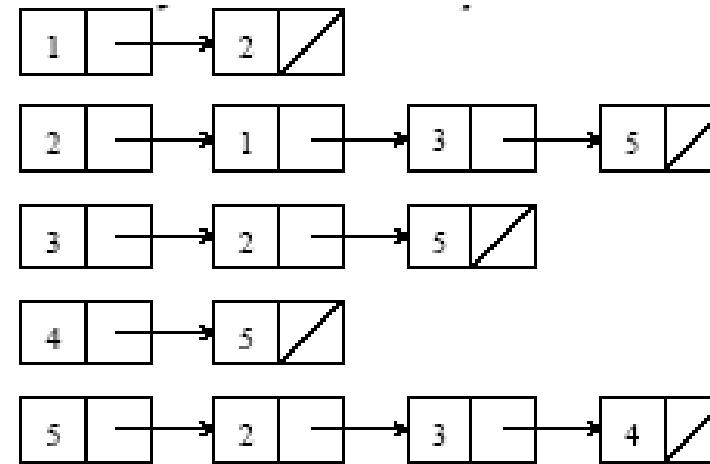
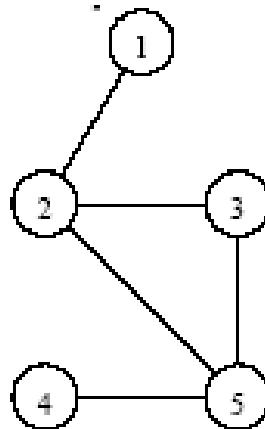
- **Unilaterally connected graph** there exists a path between any pair of nodes u, v in G such that there is a path from u to v or a path from v to u, but not both.

Graph Representation



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

An undirected graph and its adjacency matrix representation.



An undirected graph and its adjacency list representation.

Definitions

An undirected graph is *connected* if every pair of vertices is connected by a path.

A *forest* is an acyclic graph, and a *tree* is a connected acyclic graph.

A graph that has weights associated with each edge is called a *weighted graph*.

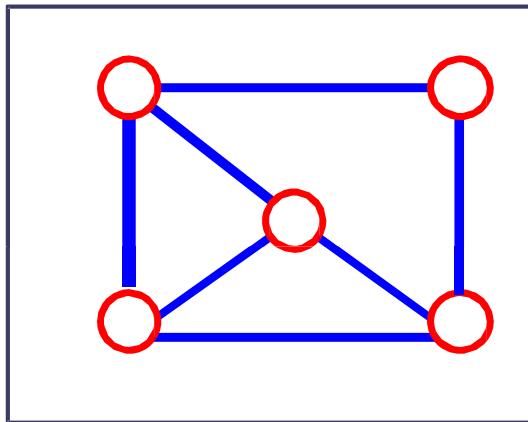
adjacent vertices: connected by an edge

degree (of a vertex): # of adjacent vertices.

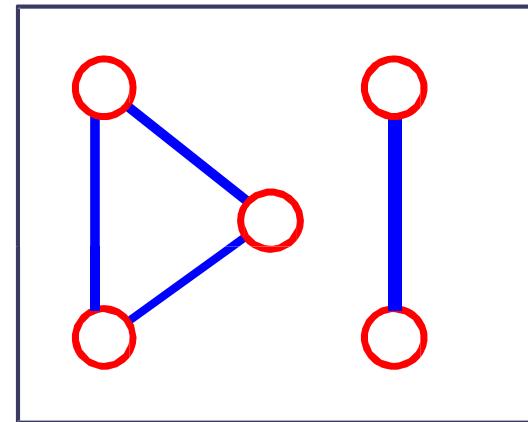
path: sequence of vertices v_1, v_2, \dots, v_k such that consecutive vertices v_i and v_{i+1} are adjacent.

Connected graph

connected graph: any two vertices are connected by some path



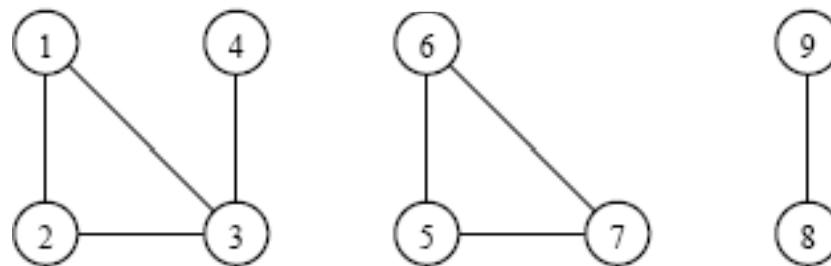
Connected



not connected

Connected Components

- The connected components of an undirected graph are the equivalence classes of vertices under the “is reachable from” relation.



- A graph with three connected components: {1, 2, 3, 4}; {5, 6, 7}; and {8, 9}.

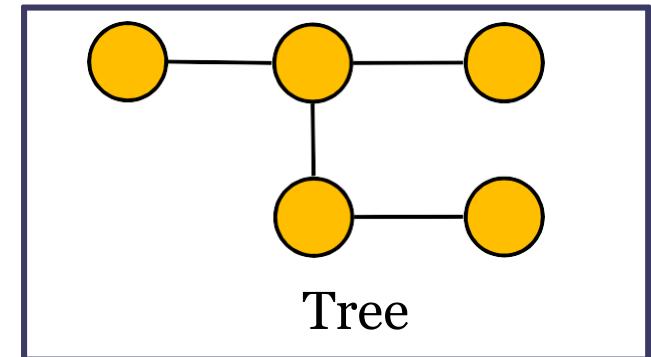
Trees and Forests

A tree is an undirected graph T such that

T is connected

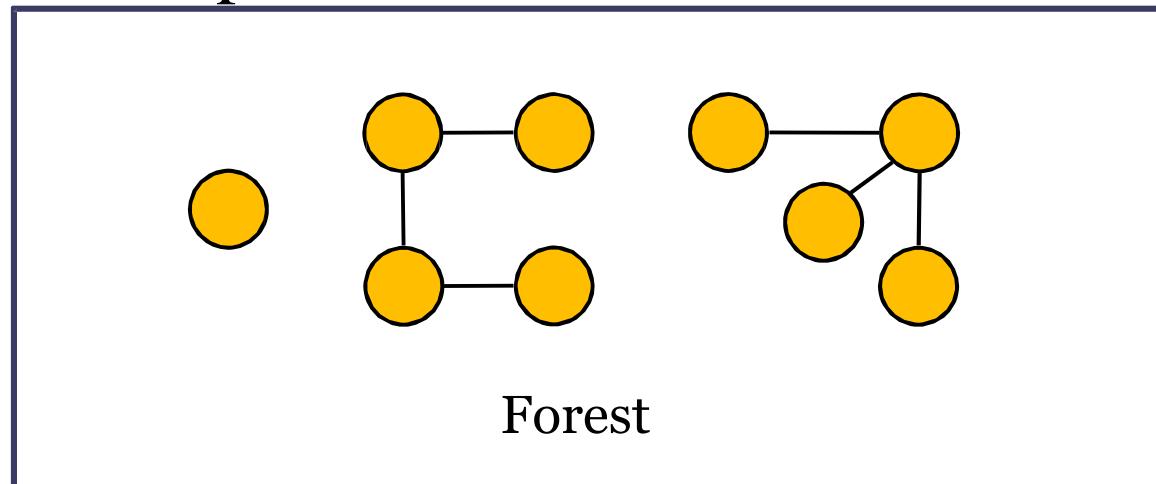
T has no cycles

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



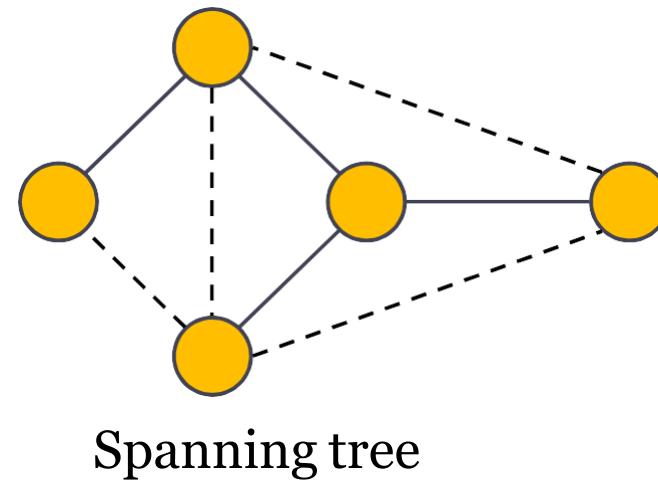
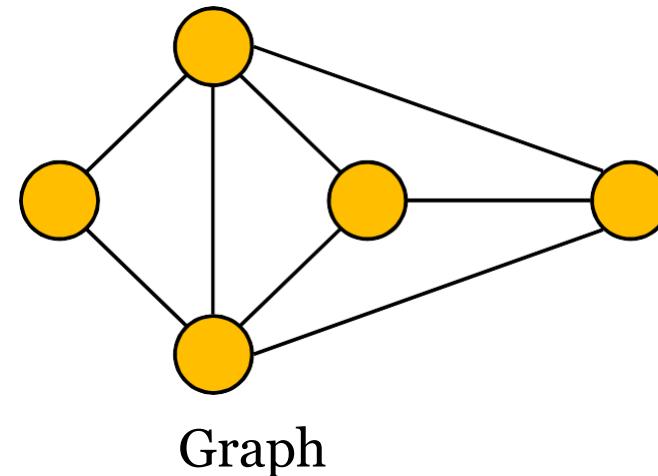
Spanning Trees and Forests

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

A spanning forest of a graph is a spanning subgraph that is a forest



Connectivity

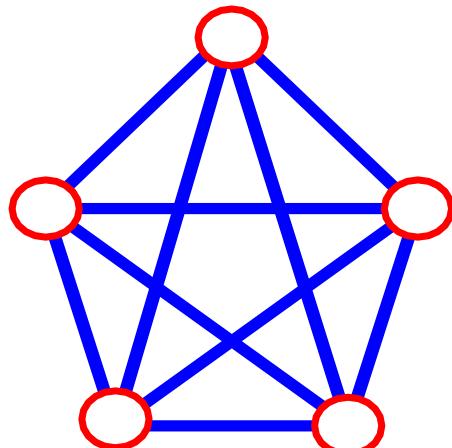
Let $n = \# \text{vertices}$, and $m = \# \text{edges}$

A complete graph: one in which all pairs of vertices are adjacent

How many total edges in a complete graph?

- Each of the n vertices is incident to $n-1$ edges, however, we would have counted each edge twice! Therefore, intuitively, $m = n(n-1)/2$.

Therefore, if a graph is not complete, $m < n(n-1)/2$



$$n = 5$$

$$m = (5 * 4) / 2 = 10$$

Connectivity

\mathbf{n} = #vertices

\mathbf{m} = #edges

For a tree $\mathbf{m} = \mathbf{n} - 1$

$\mathbf{n} = 5$

$\mathbf{m} = 4$

If $\mathbf{m} < \mathbf{n} - 1$, G is not connected

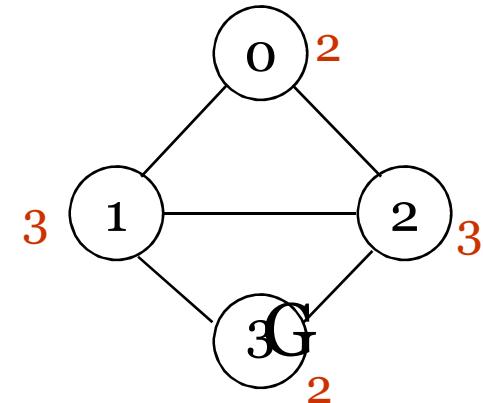
$\mathbf{n} = 5$

$\mathbf{m} = 3$

Degree of vertex

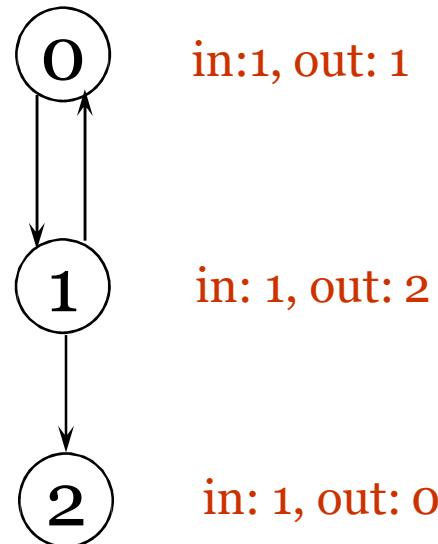
Undirected Graph:

Degree of vertex



Directed Graph:

in-degree and out-degree



Breadth First Search (BFS)

Input: Graph $G = (V, E)$, either directed or undirected, and *source vertex* $s \in V$.

Output:

$d[v] =$ distance (smallest # of edges, or shortest path) from s to v ,
for all $v \in V$. $d[v] = \infty$ if v is not reachable from s .

- $[v] = u$ such that (u, v) is last edge on shortest path $s \rightarrow v$.
 u is v 's predecessor.

Builds breadth-first tree with root s that contains all reachable vertices.

BFS: some points

A vertex is “discovered” the first time it is encountered during the search.

A vertex is “finished” if all vertices adjacent to it have been discovered.

Color the vertices to keep track of progress.

White – Undiscovered.

Gray – Discovered but not finished.

Black – Finished.

- Colors are required only to reason about the algorithm. Can be implemented without colors.

BFS: Algorithm

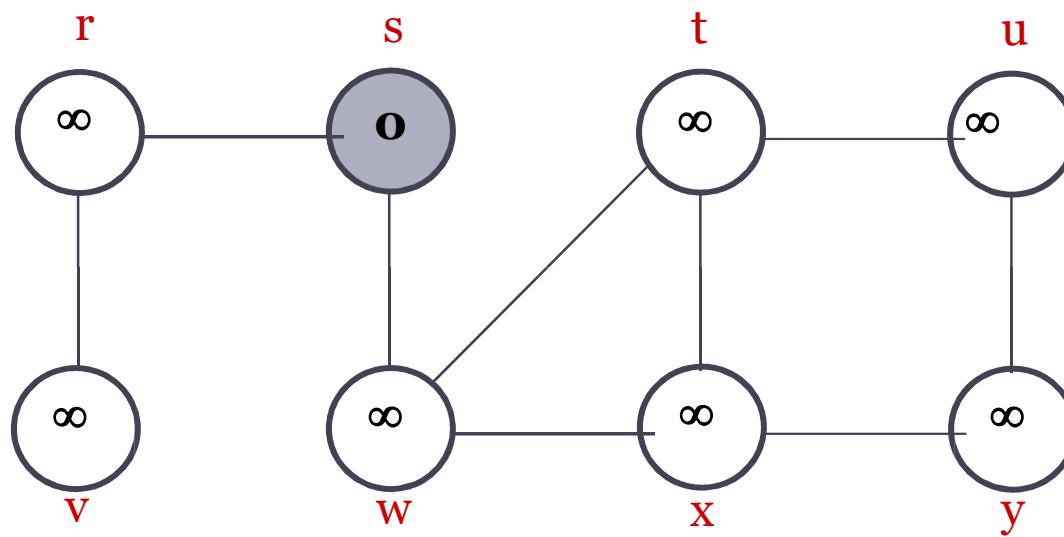
BFS(G,s)

```
1. for each vertex u in V[G] – {s}  
2      do  $\text{color}[u] \leftarrow \text{white}$   
3           $d[u] \leftarrow \infty$   
4           $\pi[u] \leftarrow \text{nil}$   
5   $\text{color}[s] \leftarrow \text{gray}$   
6   $d[s] \leftarrow 0$   
7   $\pi[s] \leftarrow \text{nil}$   
8   $Q \leftarrow \emptyset$   
9  enqueue( $Q, s$ )  
10 while  $Q \neq \emptyset$   
11      do  $u \leftarrow \text{dequeue}(Q)$   
12          for each  $v$  in  $\text{Adj}[u]$   
13              do if  $\text{color}[v] = \text{white}$   
14                  then  $\text{color}[v] \leftarrow \text{gray}$   
15                   $d[v] \leftarrow d[u] + 1$   
16                   $\pi[v] \leftarrow u$   
17                  enqueue( $Q, v$ )  
18       $\text{color}[u] \leftarrow \text{black}$ 
```

white: undiscovered
gray: discovered
black: finished

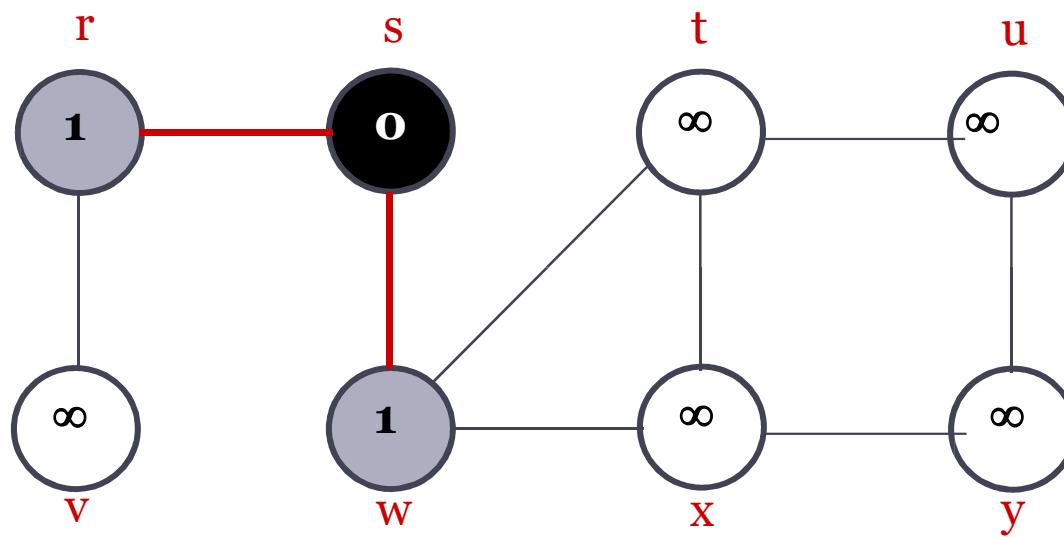
Q : a queue of discovered vertices
 $\text{color}[v]$: color of v
 $d[v]$: distance from s to v
 $\pi[u]$: predecessor of v

BFS: Example



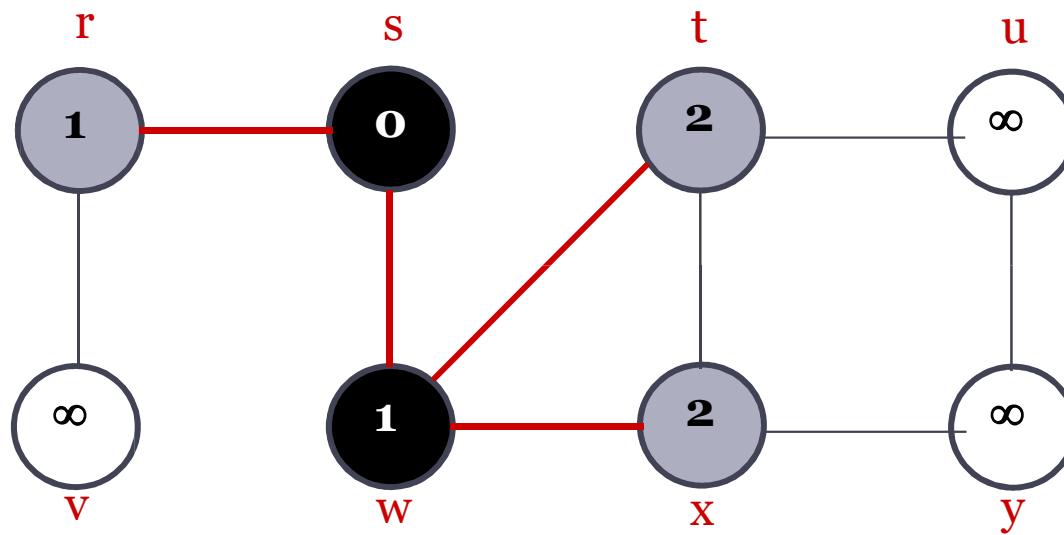
Q: s
o

BFS: Example



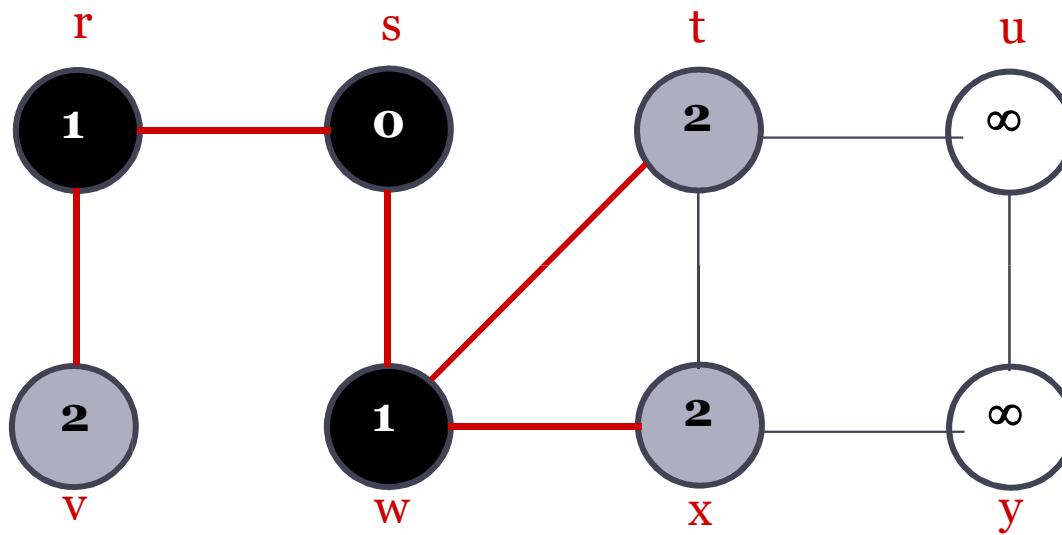
Q: w r
1 1

BFS: Example



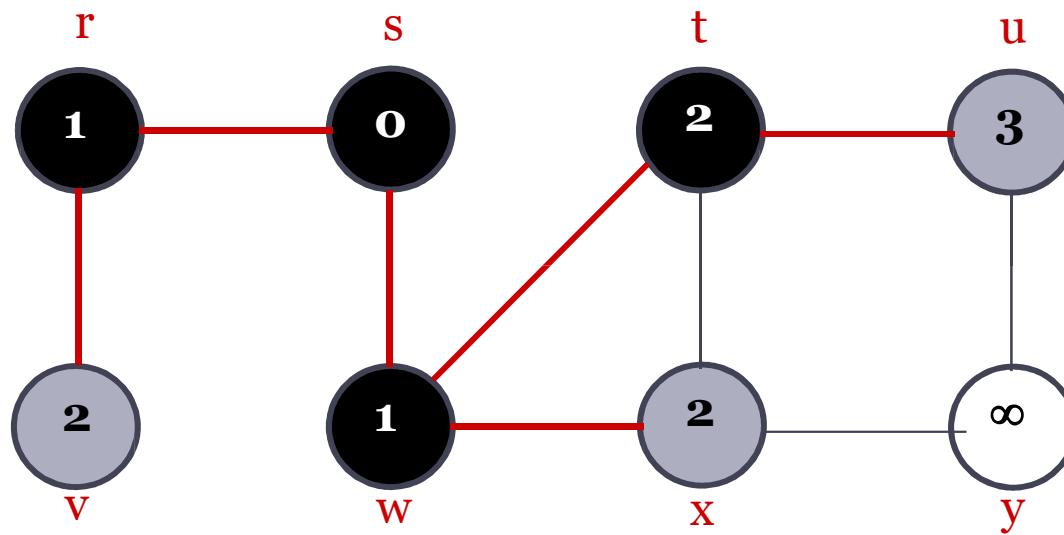
Q:	r	t	x
	1	2	2

BFS: Example



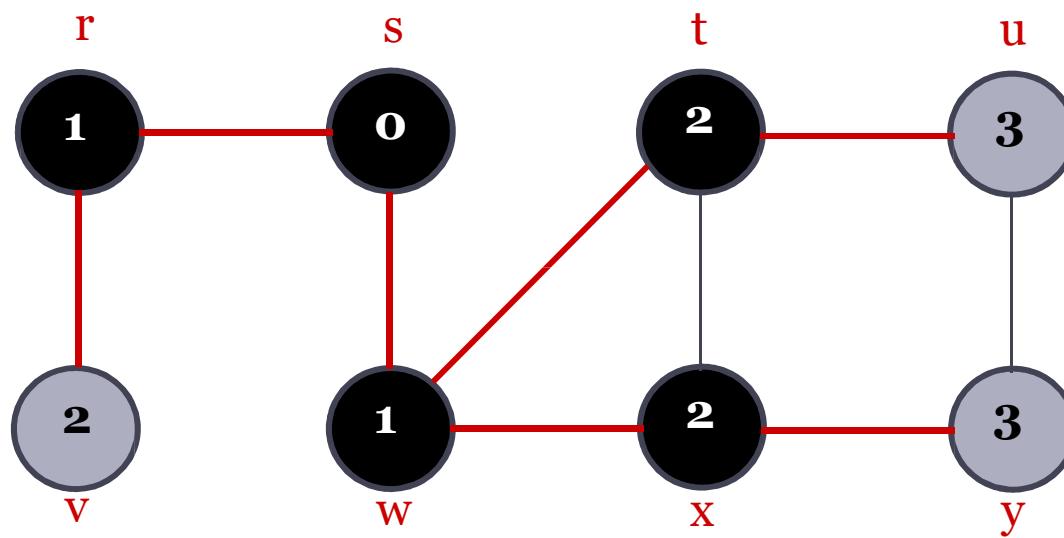
Q:	t	x	v
	2	2	2

BFS: Example



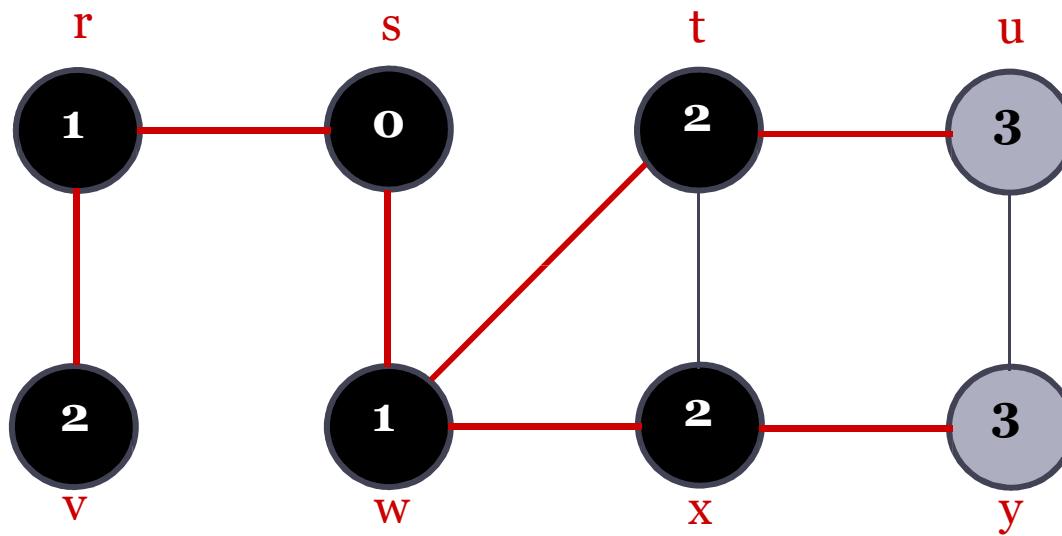
Q: x v u
2 2 3

BFS: Example



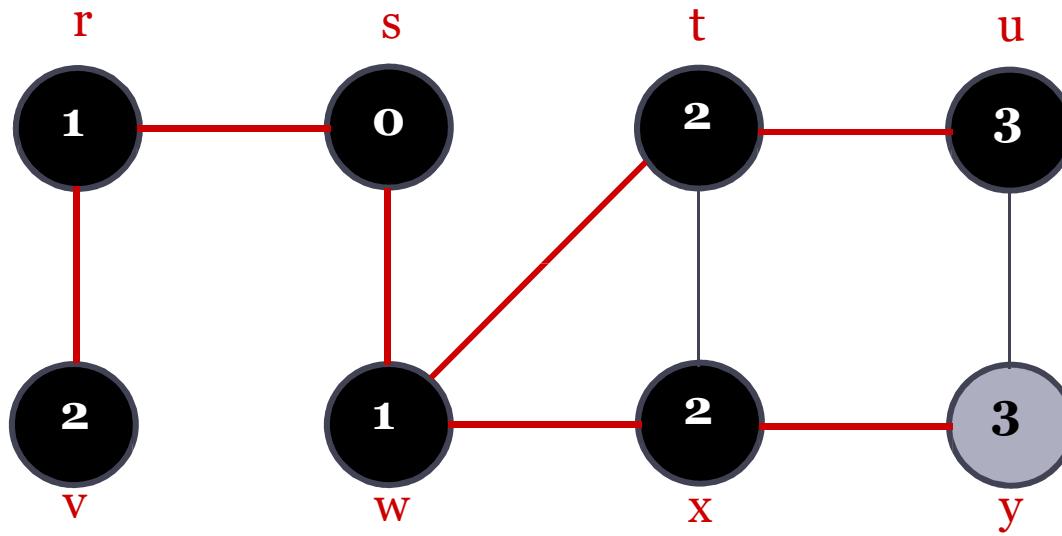
Q: v u y
2 3 3

BFS: Example



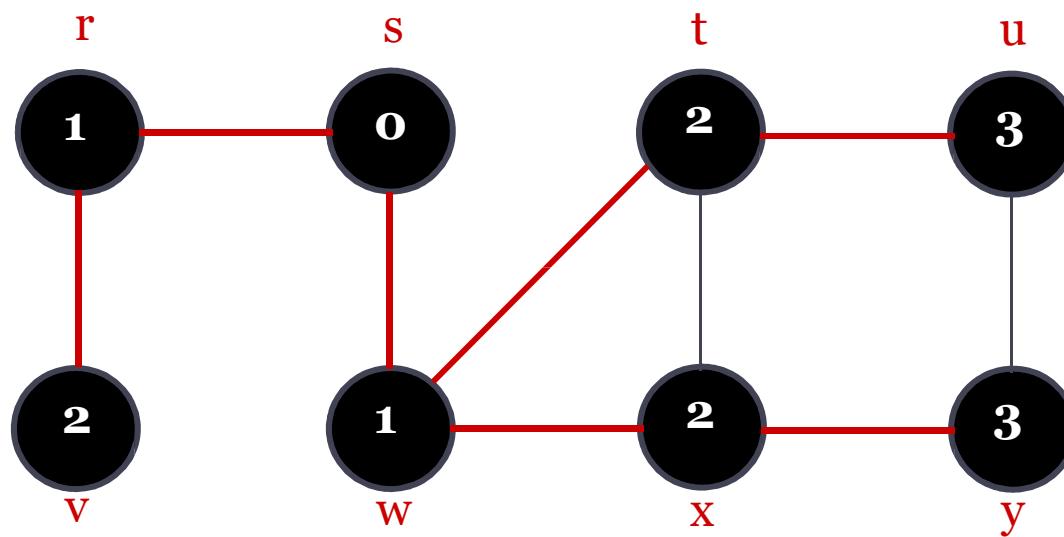
Q: u y
3 3

BFS: Example



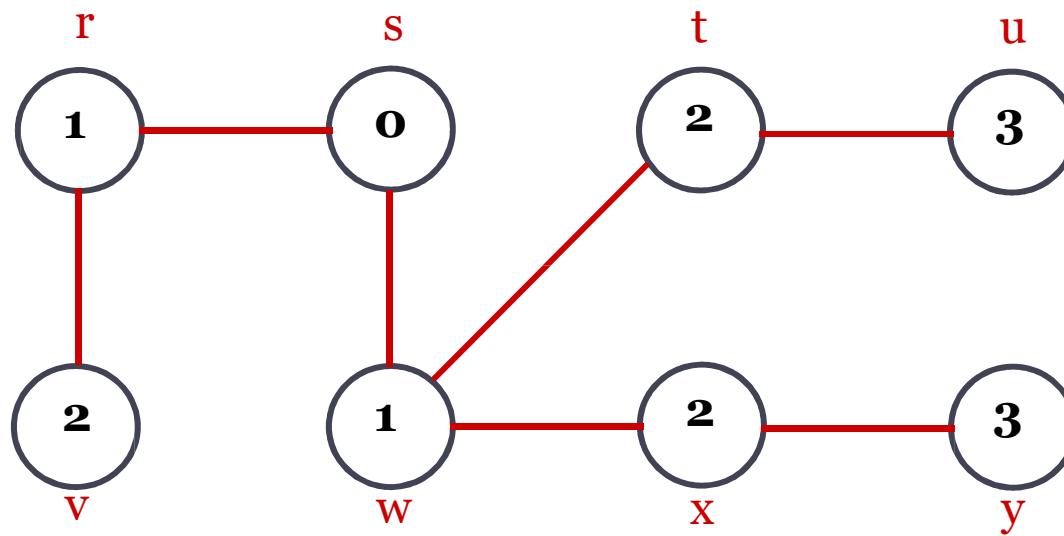
Q: y
3

BFS: Example



Q: \emptyset

BFS: Example



BF Tree

BFS: Analysis

Initialization takes $O(V)$.

Traversal Loop

After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(V)$.

The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $\Theta(E)$.

Summing up over all vertices \Rightarrow total running time of BFS is $O(V+E)$, linear in the size of the adjacency list representation of graph.

Depth First Search traversal

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
 - 2 timestamps on each vertex. Integers between 1 and $2|V|$.
 - $d[v] = \text{discovery time}$ (v turns from white to gray)
 - $f[v] = \text{finishing time}$ (v turns from gray to black)
 - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u 's adjacency list.
- Uses the same coloring scheme for vertices as BFS.

DFS: Algorithm

DFS(G)

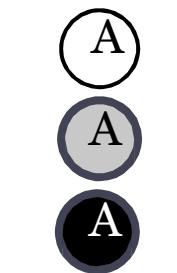
1. **for** each vertex $u \in V[G]$
2. **do** $\text{color}[u] \leftarrow \text{WHITE}$
3. $\pi[u] \leftarrow \text{NIL}$
4. $\text{time} \leftarrow 0$
5. **for** each vertex $u \in V[G]$
6. **do if** $\text{color}[u] = \text{WHITE}$
7. **then** DFS-Visit(u)

Uses a global timestamp **time**.

DFS-Visit(u)

1. $\text{color}[u] \leftarrow \text{GRAY}$ // White vertex
 u has been discovered
2. $\text{time} \leftarrow \text{time} + 1$
3. $d[u] \leftarrow \text{time}$
4. **for** each $v \in \text{Adj}[u]$
5. **do if** $\text{color}[v] = \text{WHITE}$
6. **then** $\pi[v] \leftarrow u$
 DFS-Visit(v)
7. $\text{color}[u] \leftarrow \text{BLACK}$ // Blacken u ;
 it is finished.
9. $f[u] \leftarrow \text{time} \leftarrow \text{time} + 1$

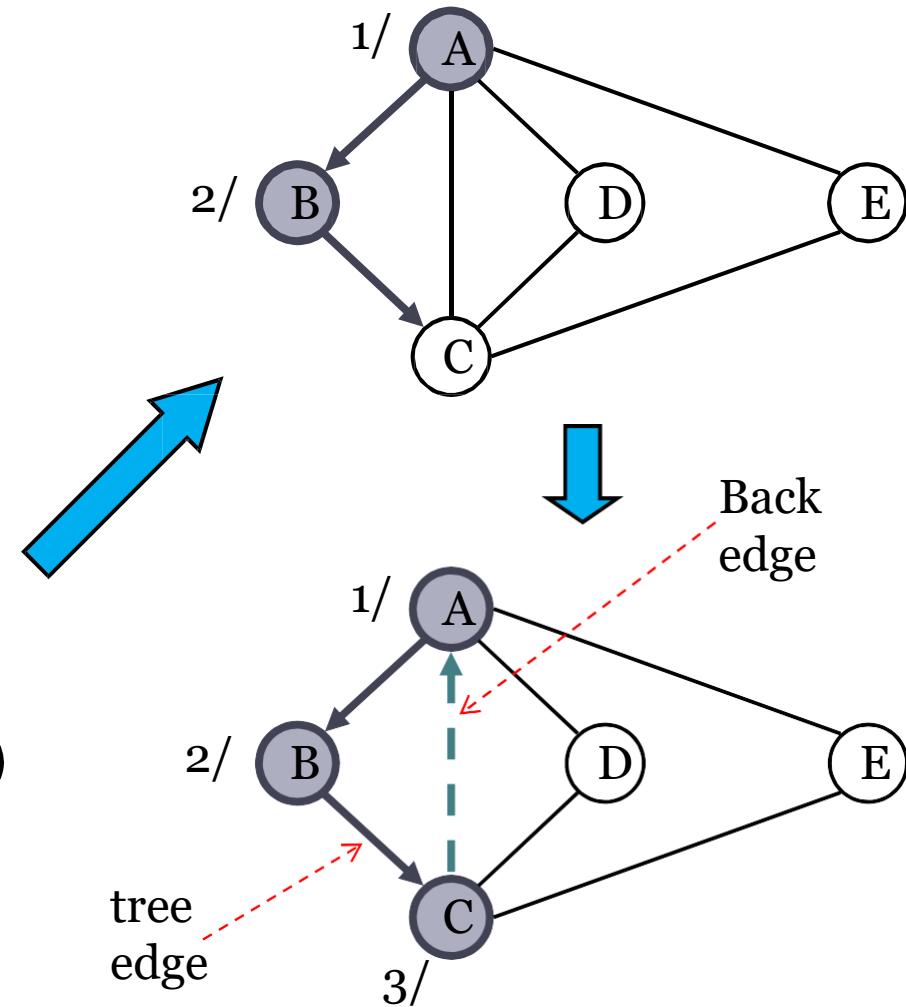
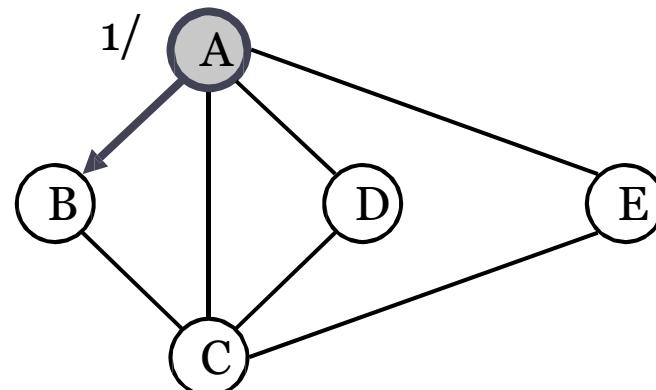
DFS: Example



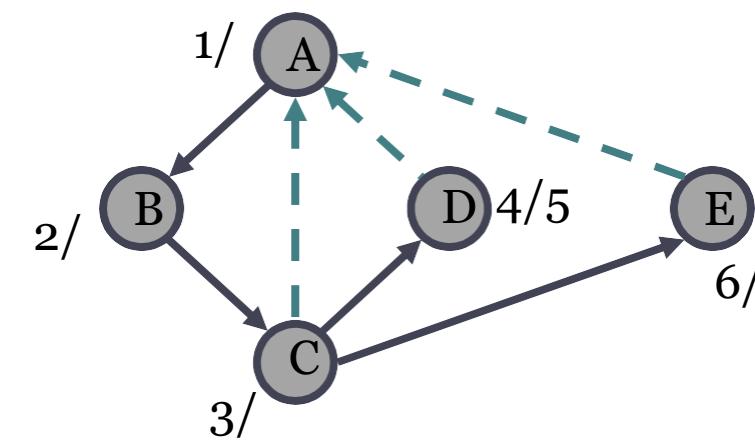
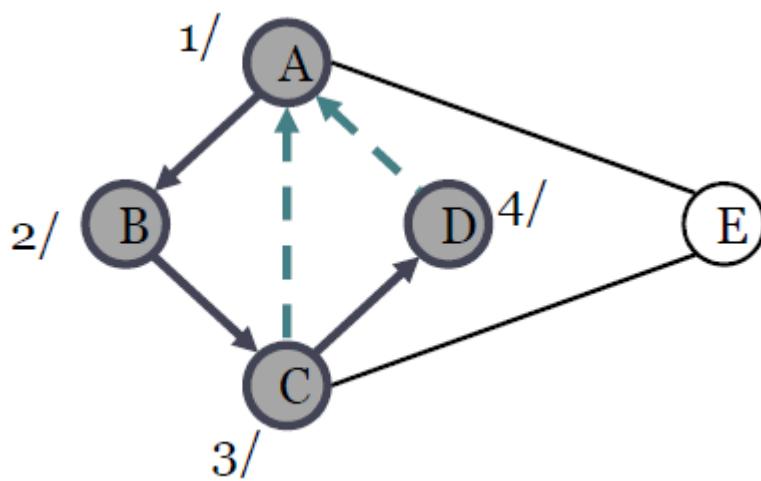
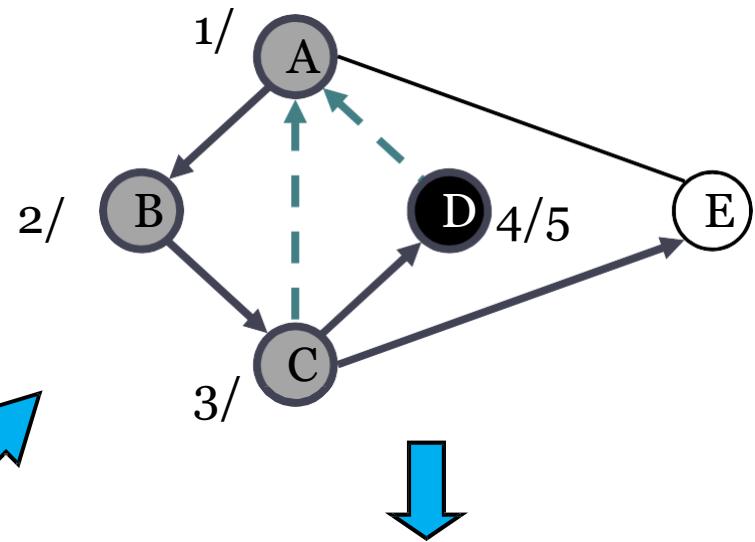
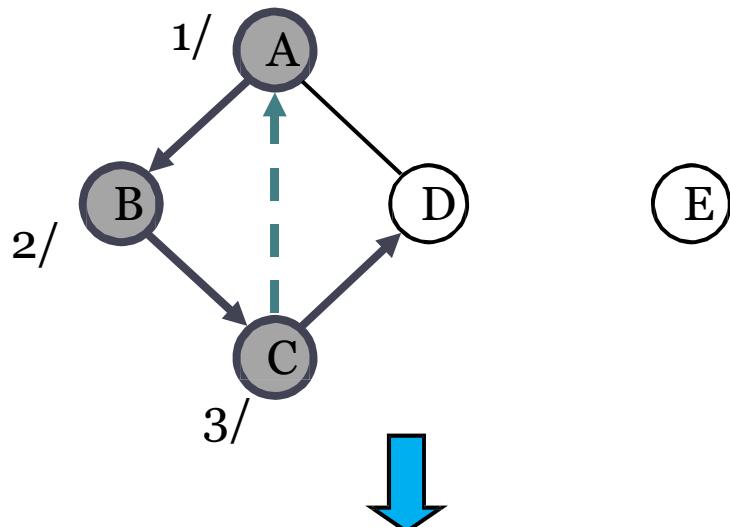
— unexplored edge

→ discovery edge

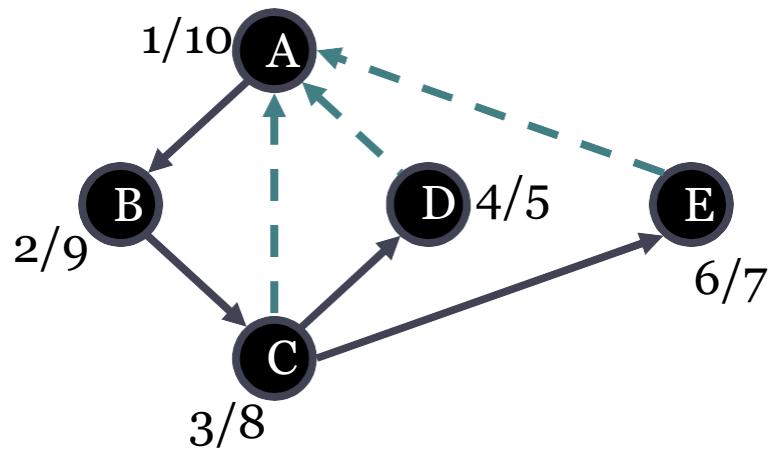
→ back edge



Example...



Example...



(u,v) is Back edge if
 $d(v) < d(u)$

(u,v) is tree edge if
vertex v is discovered first from
vertex u .

Properties of DFS

Property 1

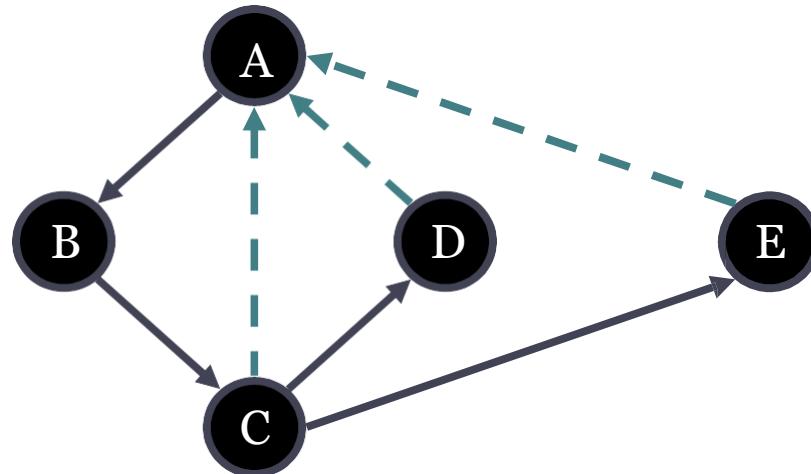
$\text{DFS-VISIT}(G, u)$ visits all the vertices and edges in the connected component of v .

Property 2

The discovery edges labeled by $\text{DFS-VISIT}(G, v)$ form a spanning tree of the connected component of v .

Property 3

The $\text{DFS}(G)$ form a forest of spanning trees of the connected components of G .



Analysis of DFS

Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.

DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time.

Lines 4-7 of DFS-Visit is executed $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$

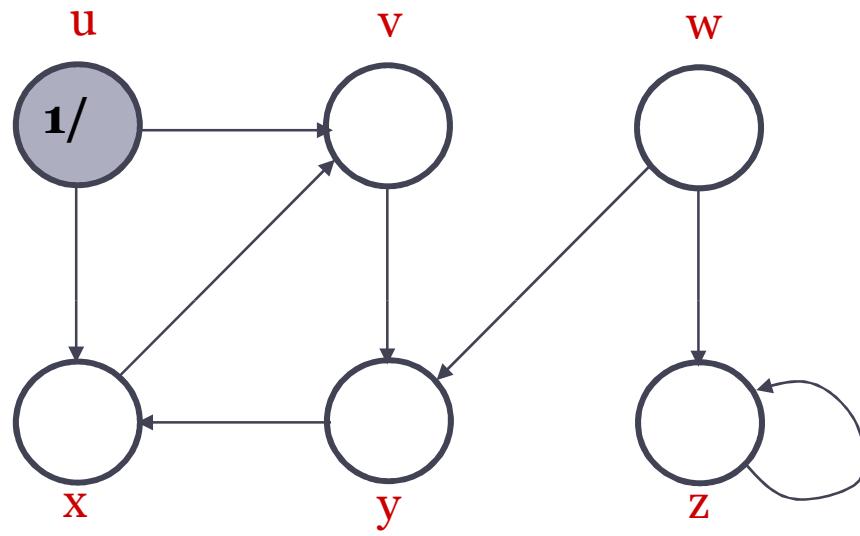
Total running time of DFS is $\Theta(V+E)$.

DFS on directed Graph

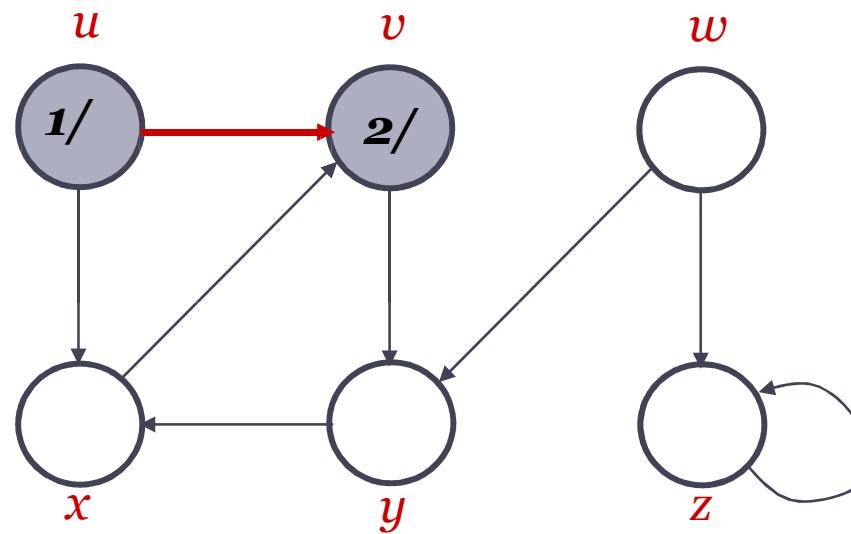
Four type of edges are produces

1. **Tree edges:** are edges (u,v) if v was first discovered by exploring edge (u,v) .
2. **Back edges:** are edges (u,v) connecting a vertex u to an ancestor v in DFS tree. Self loops are also called back edges.
3. **Forward edges:** are non-tree edges (u,v) connecting a vertex u to a descendent v in DFS tree.
4. **Cross edges:** are all other edges. Can go between vertices in the same DFS tree or they can go between vertices in different DFS trees.

DFS on directed Graph

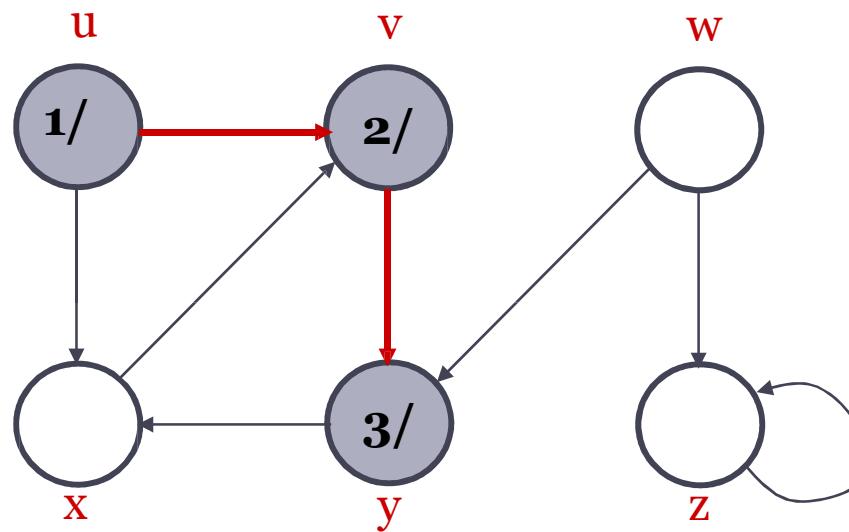


Example (DFS)



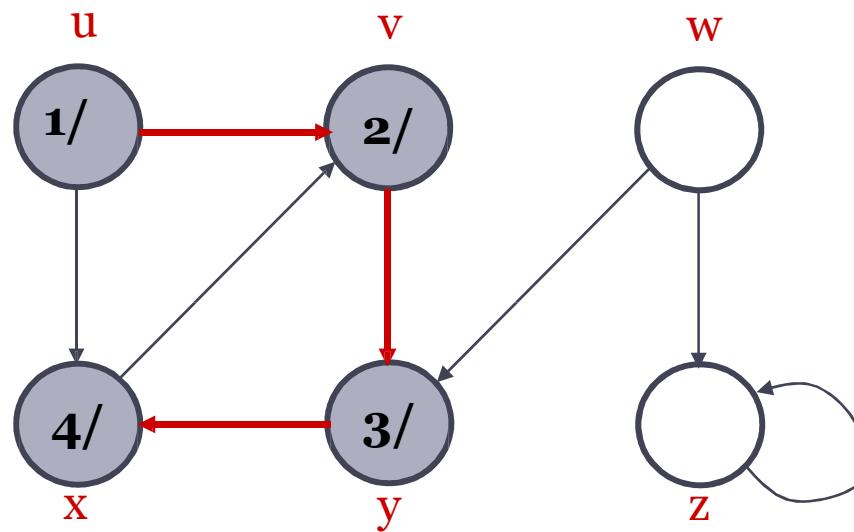
Consider edge (u,v)

Example (DFS)



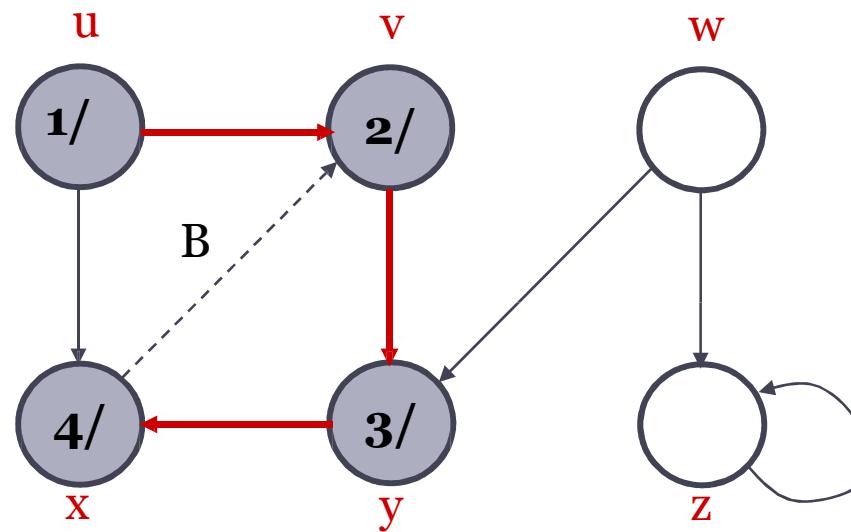
From v , Consider edge (v, y)

Example (DFS)



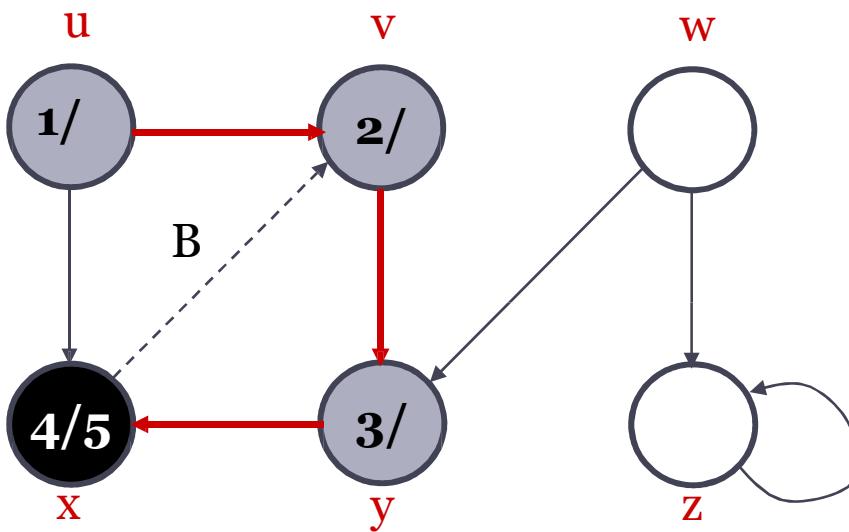
From y , Consider edge (y, x)

Example (DFS)



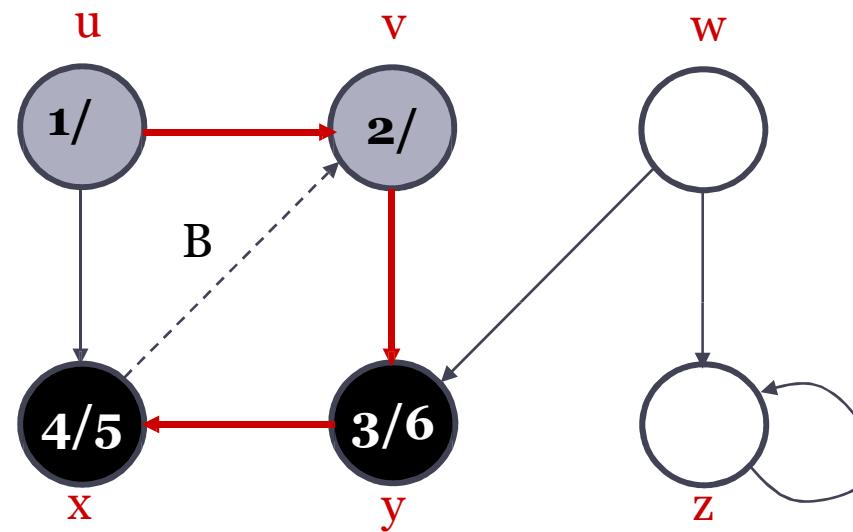
From x , Consider edge (x, v) :
do not include in tree

Example (DFS)



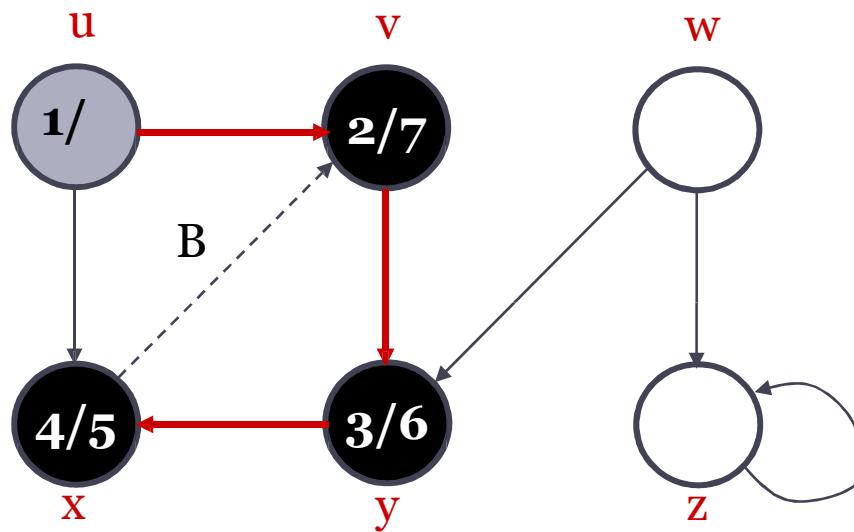
Vertex x , no more edges, finish it.

Example (DFS)



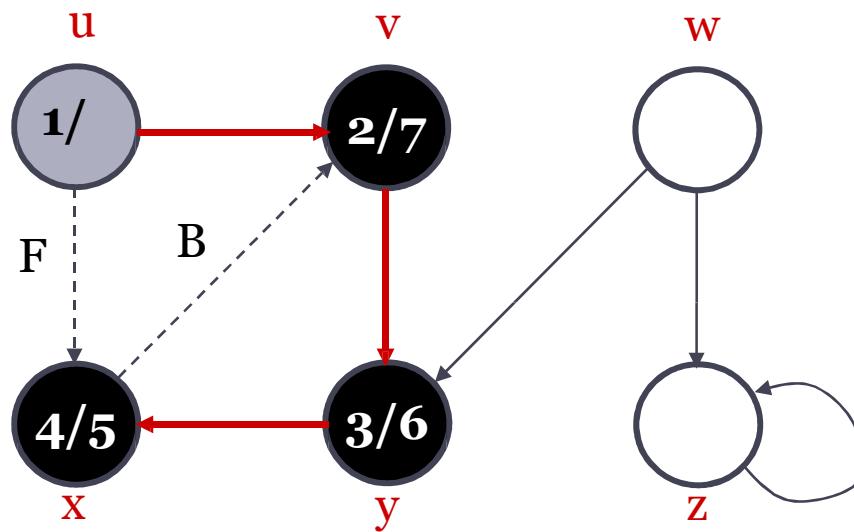
From y , no more edges, finish it

Example (DFS)



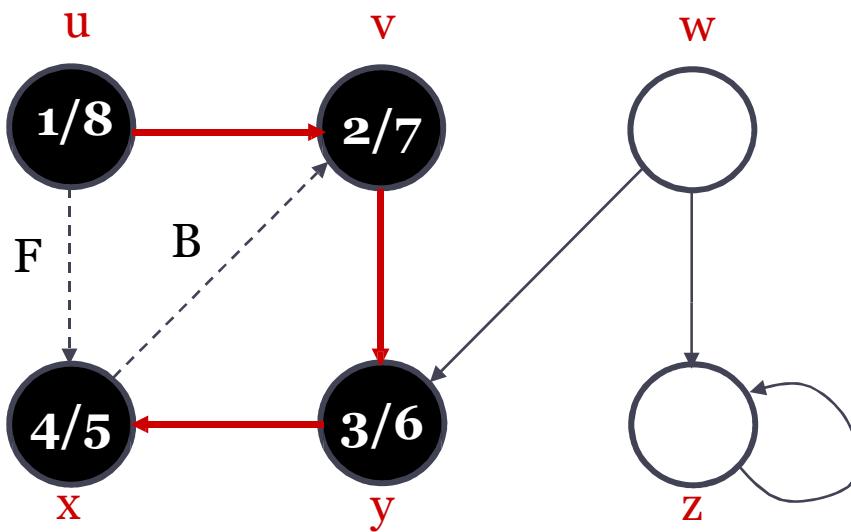
From v , no more edges, finish it

Example (DFS)



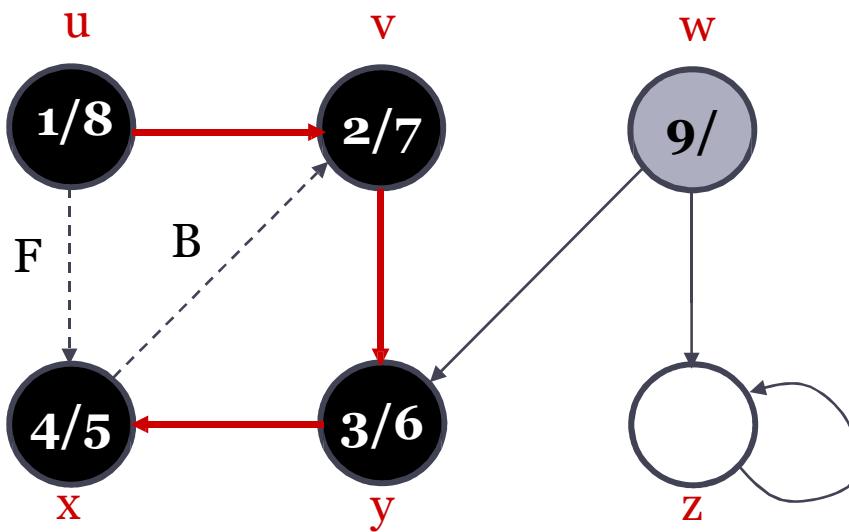
From u , consider edge (u,x) ,
do not include

Example (DFS)



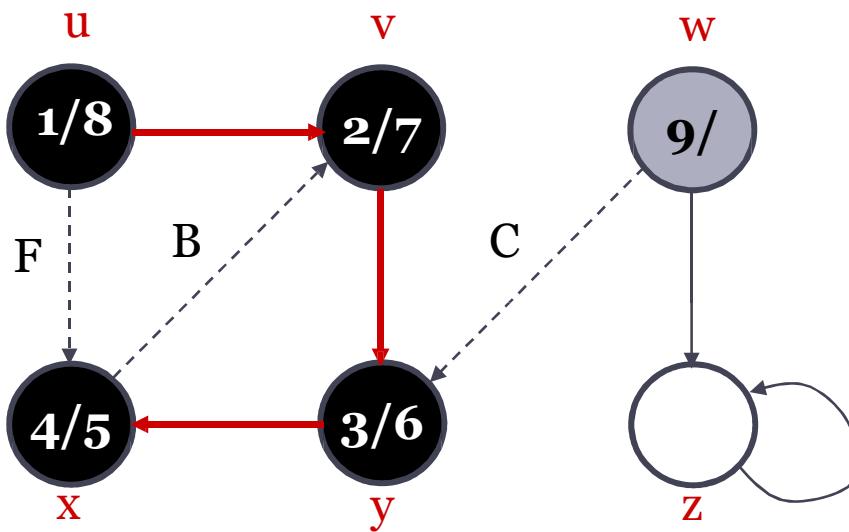
From u , no more edges, finish it

Example (DFS)



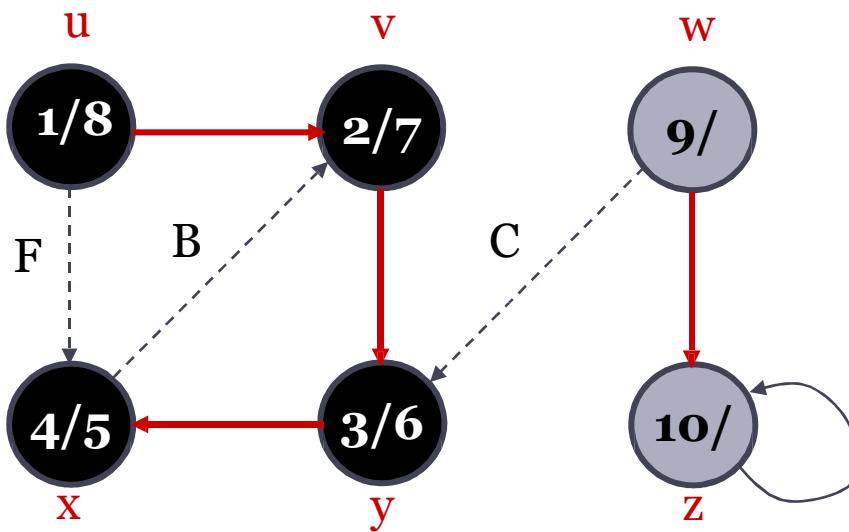
DFS from u ends, start again from w

Example (DFS)

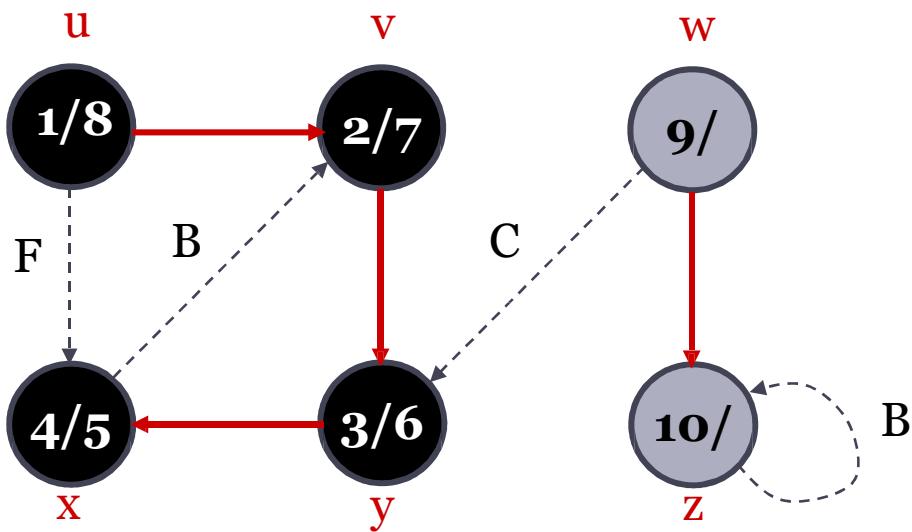


From w ends, Consider (w,y) again
from w

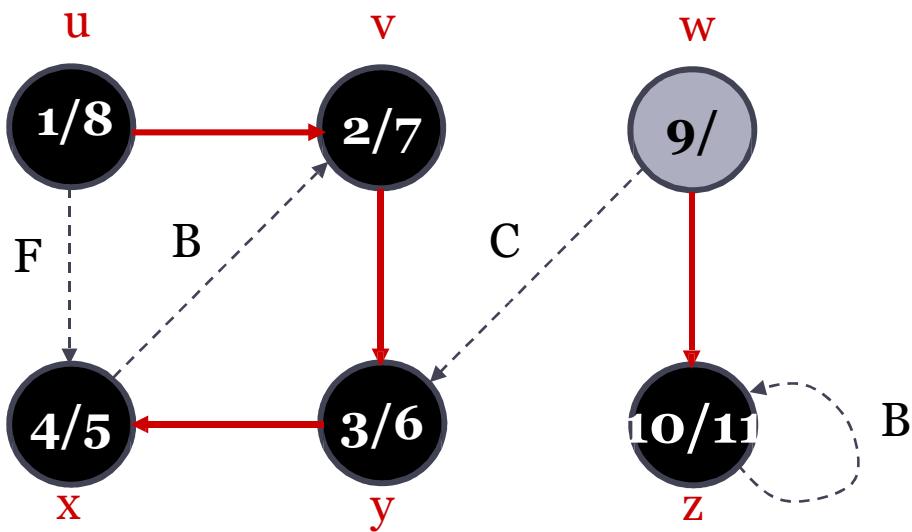
Example (DFS)



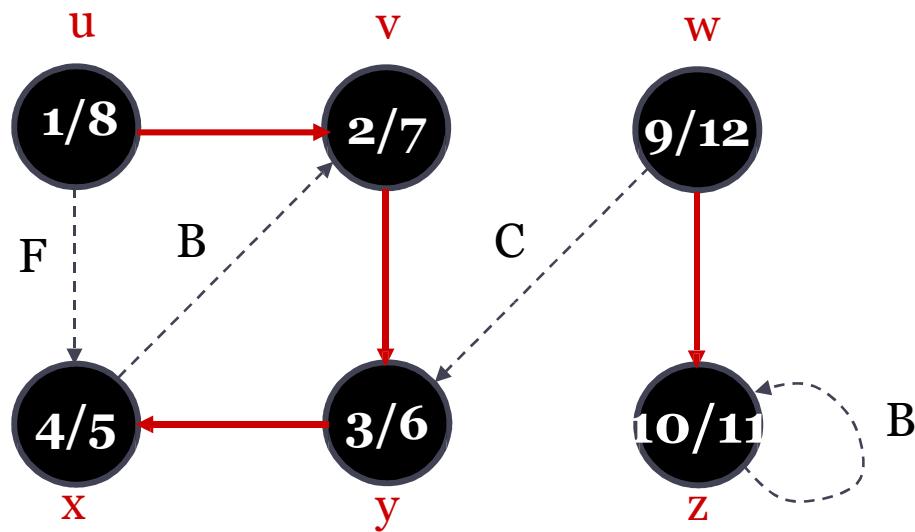
Example (DFS)



Example (DFS)



Example (DFS)



Classification of edges in DFS tree

Each edge (u,v) can be classified by the color of the vertex v that is reached when edge is first explored.

1. WHITE indicates a tree edge.
2. GRAY indicates a back edge.
3. BLACK indicates a forward edge or cross edge.

In case 3, if $d[u] < d[v]$: it is a forward edge.

In case 3, if $d[u] > d[v]$: it is a cross edge

DFS :Applications

Path Finding:

We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern

We call $\text{DFS}(G, u)$ with u as the start vertex

We use a stack S to keep track of the path between the start vertex and the current vertex

As soon as destination vertex z is encountered, we return the path as the contents of the stack

Path Finding:

Algorithm $\text{pathDFS}(G, v, z)$

1. **for each** vertex $u \in V[G]$
2. **do** $\text{color}[u] = \text{WHITE}$
3. $\text{Done} = \text{FALSE}$
4. $\text{pathDFS-VISIT}(G, v, z)$

Algorithm $\text{pathDFS-VISIT}(G, v, z)$

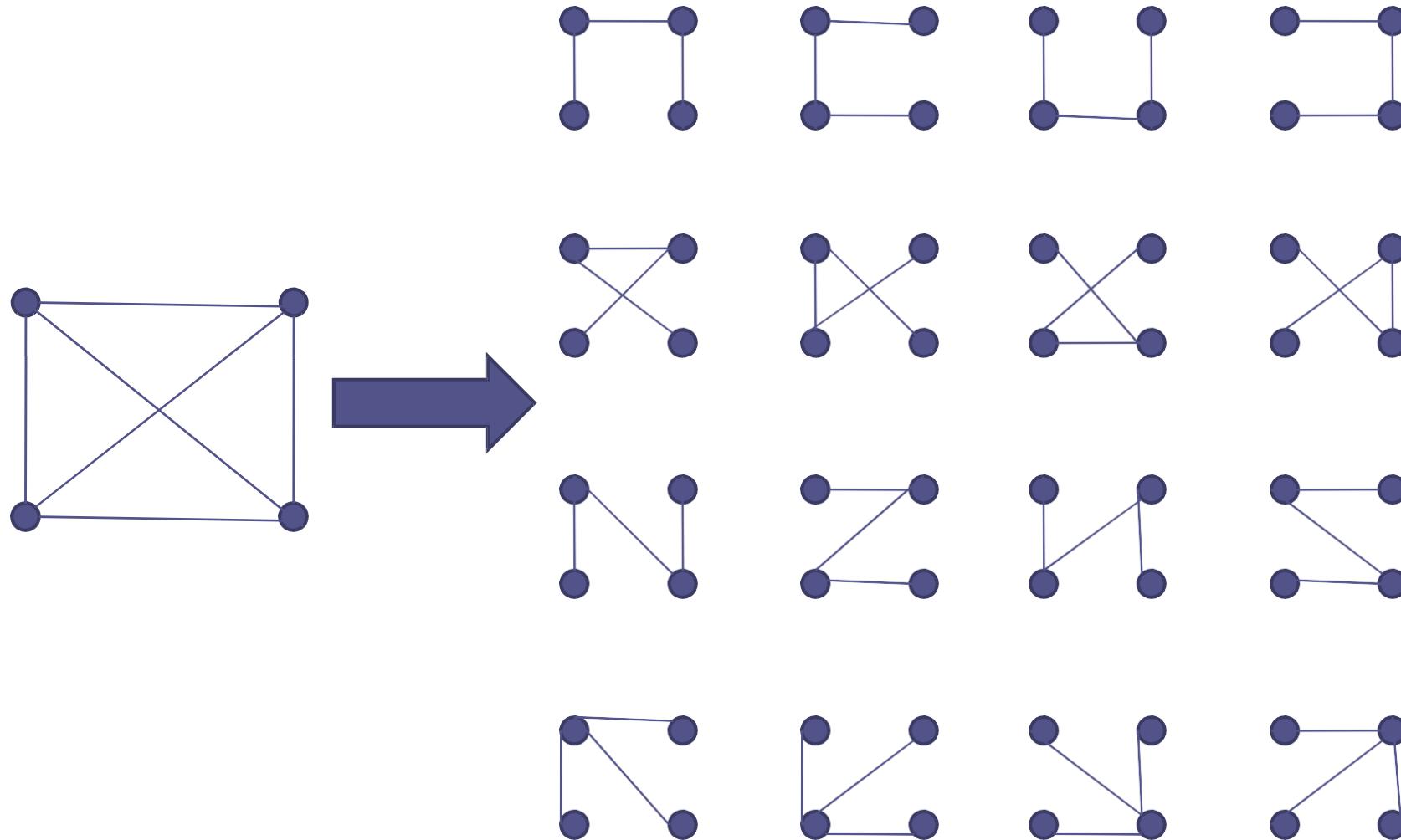
1. $\text{Color}[v] = \text{GRAY}$
2. $S.\text{push}(v)$
3. If $(v = z)$
4. **THEN** $\text{Done} = \text{TRUE}$
5. **return** $S.\text{elements}$
6. **for each** $u \in \text{Adj}[v]$
7. **do if** $(\text{color}[u] = \text{WHITE})$
8. **THEN** $\text{pathDFS}(G, u, z)$
9. **if** (Done) **THEN return;**
10. $S.\text{pop}()$
11. $\text{Color}[v] = \text{BLACK}$

Minimum Spanning Trees

Spanning Trees

- A *spanning tree* of a graph is a tree and is a subgraph that contains all the vertices.
- A graph may have many spanning trees; for example, the complete graph on four vertices has sixteen spanning trees:

Spanning trees



Minimum Spanning Trees (MSTs)

Suppose that the edges of the graph have weights or lengths. The weight of a tree will be the sum of weights of its edges.

Based on the example, we can see that different trees have different lengths.

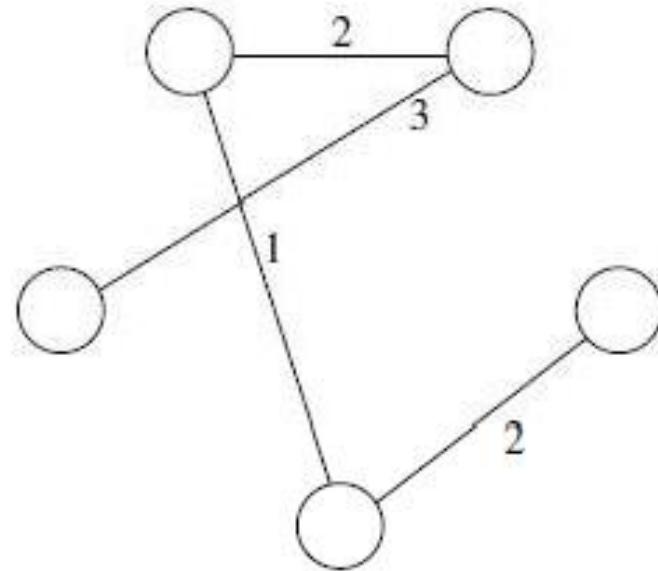
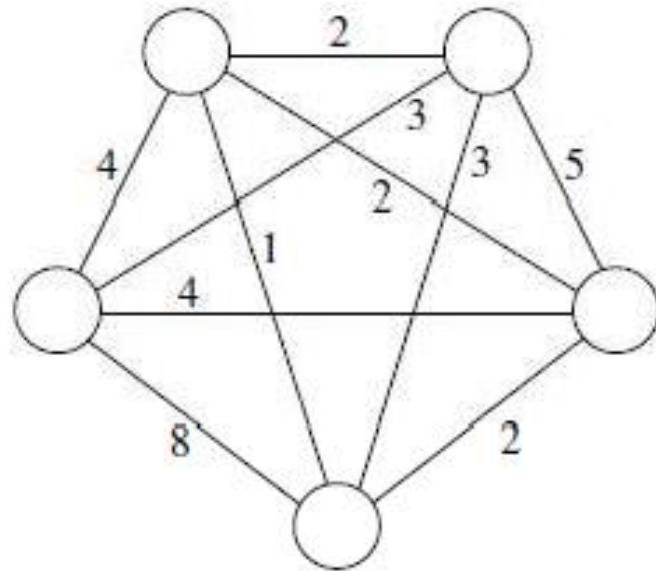
The question is: how to find the minimum length spanning tree?

The question can be solved by many different algorithms, here are two classical minimum-spanning tree algorithms :

Kruskal's Algorithm

Prim's Algorithm

Minimum Spanning Tree



An undirected graph and its minimum spanning tree

MST: Problem

- Undirected, connected graph $G = (V, E)$
- Weight function $W: E \rightarrow R$ (assigning cost or length or other values to edges)
- Cost/weight of MST: sum of weights of all edges in MST.
- Problem is to find a Minimum spanning tree: tree that connects all the vertices and having minimum weight.

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

Generic MST Algorithm

Generic-MST (G, w)

```
1  $A \leftarrow \emptyset$  // Contains edges that belong to a MST
2 while  $A$  does not form a spanning tree do
3     Find an edge  $(u, v)$  that is safe for  $A$ 
4      $A \leftarrow A \cup \{ (u, v) \}$ 
5 return  $A$ 
```

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

The algorithm manages a set of edges A maintaining the following loop invariant

- Prior to each iteration, A is a subset of some minimum spanning tree.
- At each step, an edge is determined that can be added to A without violating this invariant. Such an edge is called a Safe Edge.

Kruskal's Algorithm

Create a forest of trees from the vertices

Repeatedly merge trees by adding “**safe edges**”
until only one tree remains

A “safe edge” is an edge of minimum weight
which does not create a cycle

Kruskal's Algorithm

Edge based algorithm

Add the edges one at a time, in increasing weight order

The algorithm maintains A – a **forest of trees**. An edge is accepted if it connects vertices of distinct trees

We need a data structure that maintains a partition, i.e., a collection of disjoint sets

Make-Set(v): $S \leftarrow \{v\}$

Union(S_i, S_j): $S \leftarrow S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$

FindSet(S, x): returns unique $S_i \in S$, where $x \in S_i$

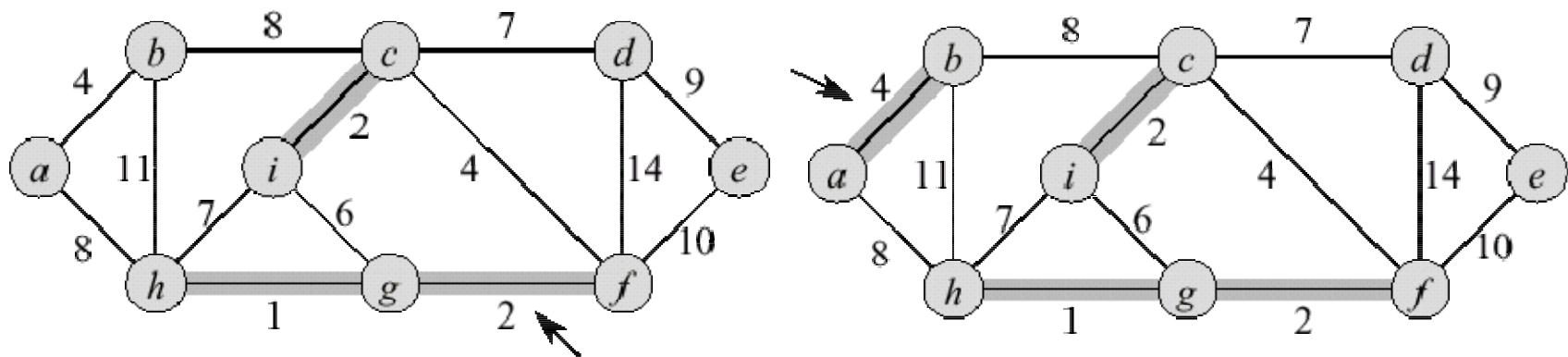
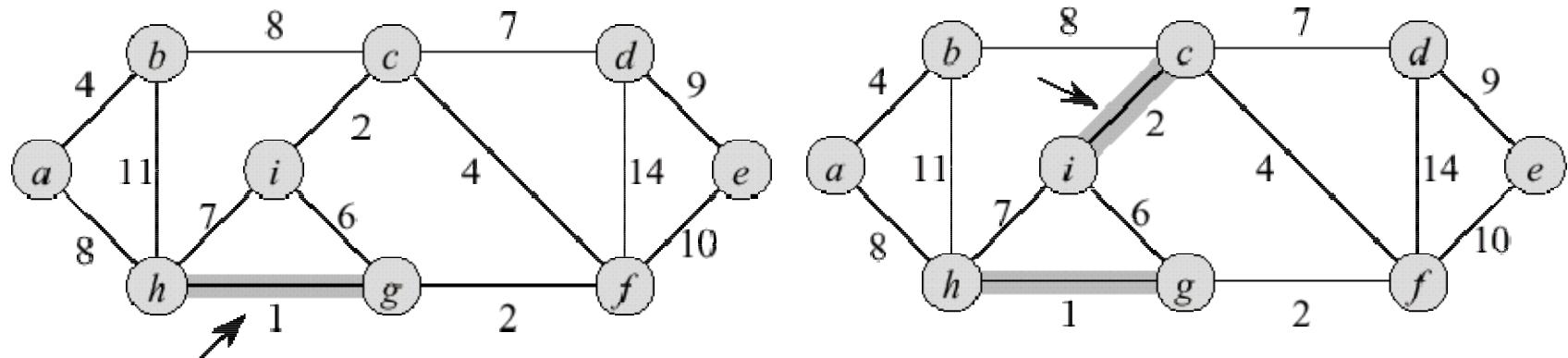
Kruskal's Algorithm

The algorithm adds the cheapest edge that connects two trees of the forest

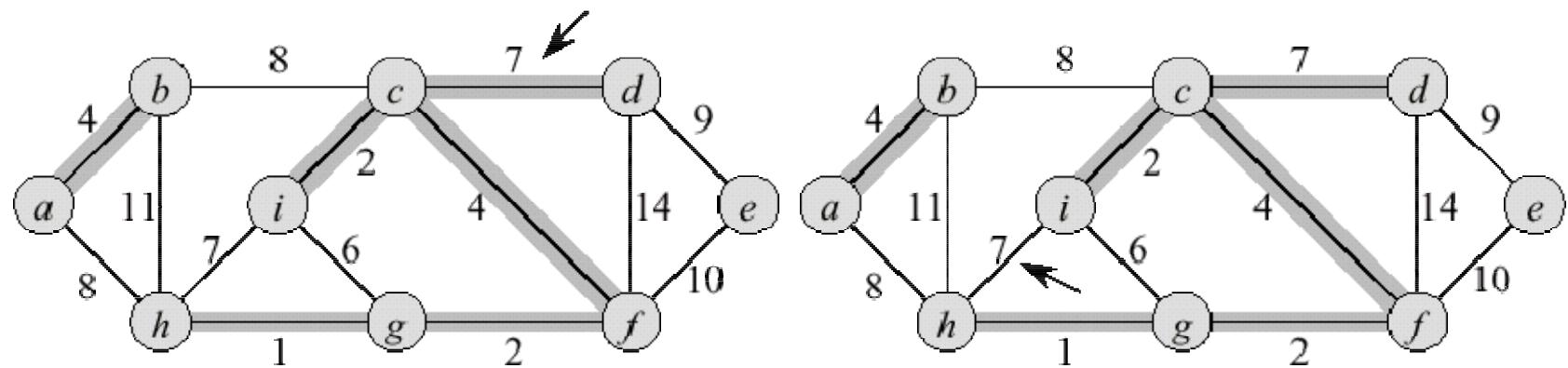
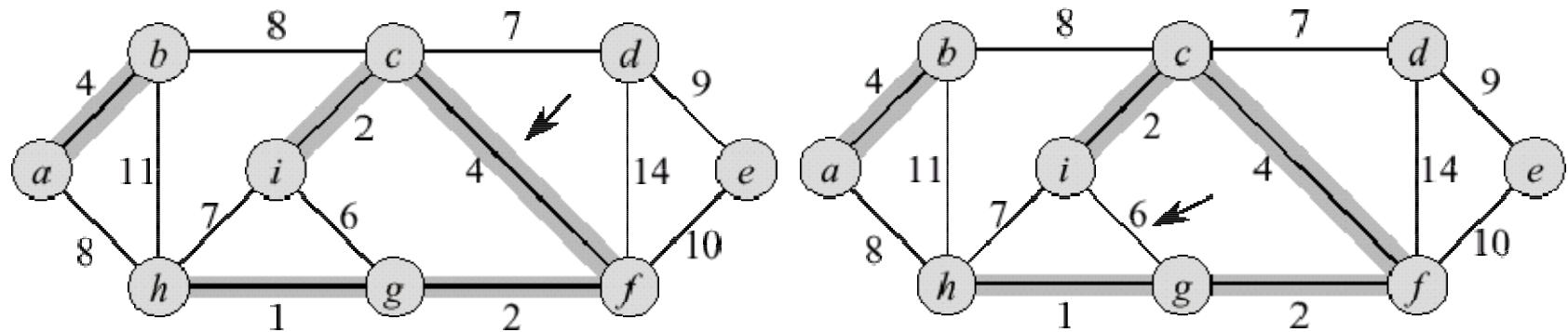
MST-Kruskal (G, w)

```
1 A ← Ø // set of edges forming MST
2 for each vertex  $v \in V[G]$  do
3     Make-Set( $v$ )
4 sort the edges of  $E$  by non-decreasing weight  $w$ 
5 for each edge  $(u, v) \in E$ , in order by non-
decreasing weight do
6     if Find-Set( $u$ ) ≠ Find-Set( $v$ ) then
7         A ← A ∪ { $(u, v)$ }
8         Union( $u, v$ ) // Union of sets containing u and v
9 return A
```

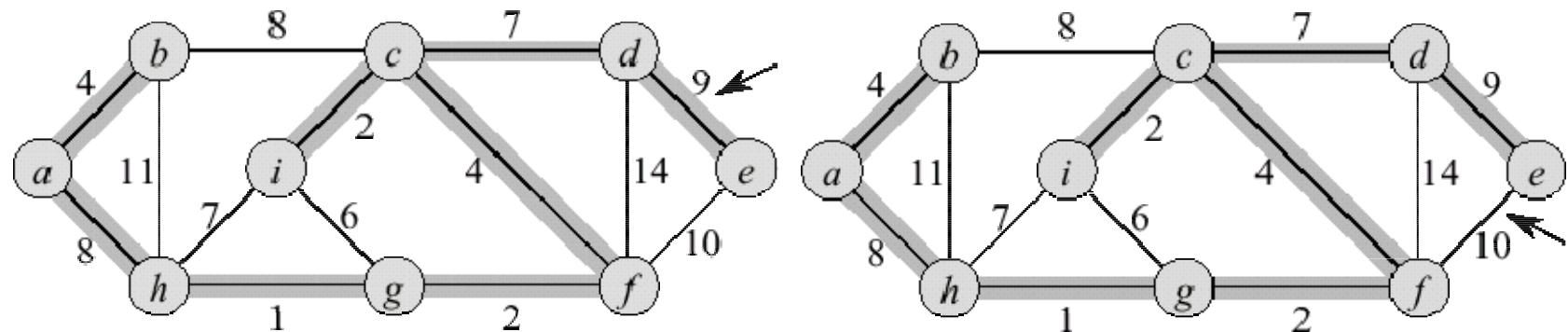
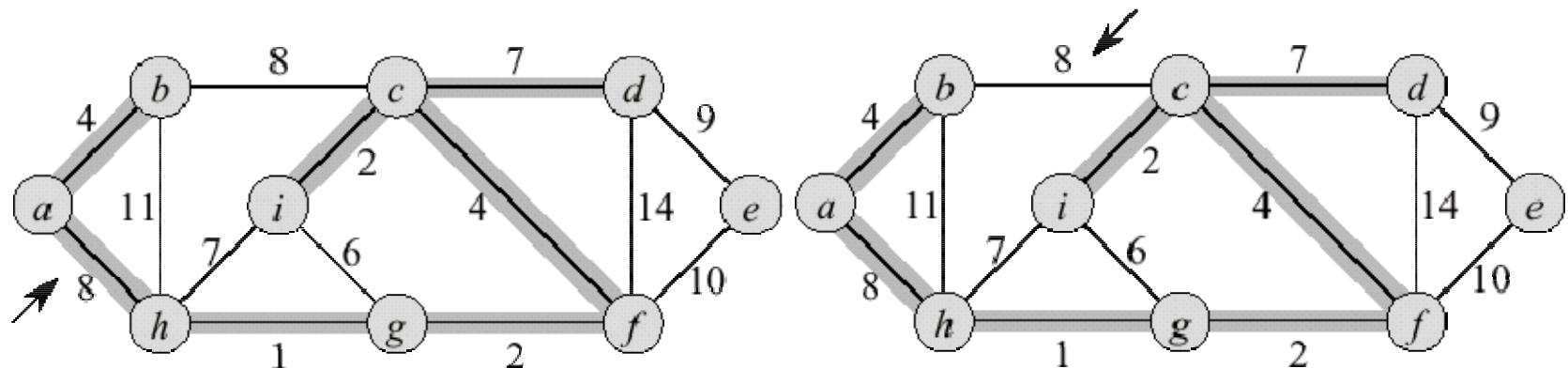
Kruskal's algorithm: example



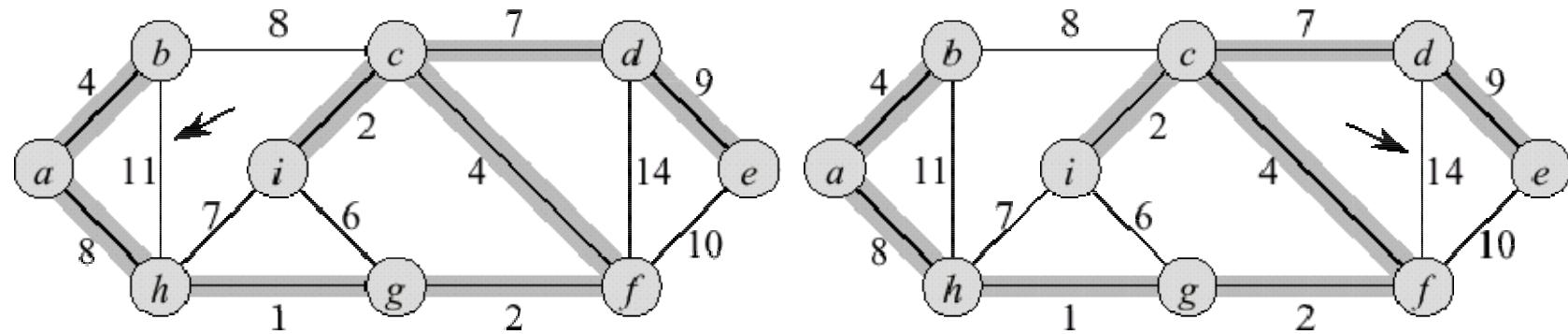
Kruskal's algorithm: example...



Kruskal's algorithm: example...



Kruskal's algorithm: example...



Kruskal's Algorithm: Running Time

Initialization $O(V)$ time

Sorting the edges $\Theta(E \lg E) = \Theta(E \lg V)$ (why?)

$O(E)$ calls to FindSet

Union costs

Let $t(v)$ – the number of times v is moved to a new cluster

Each time a vertex is moved to a new cluster the size of the cluster containing the vertex at least doubles: $t(v) \leq \log V$

Total time spent doing Union $\sum_{v \in V} t(v) \leq |V| \log |V|$
Total time: $O(E \lg V)$

Prim's Algorithm

Vertex based algorithm

It is a greedy algorithm.

Start by selecting an arbitrary vertex, include it into the current MST.

Grow the current MST by inserting into it the vertex closest to one of the vertices already in current MST.

Grows one tree T, **one vertex at a time**

A cloud covering the portion of T already computed

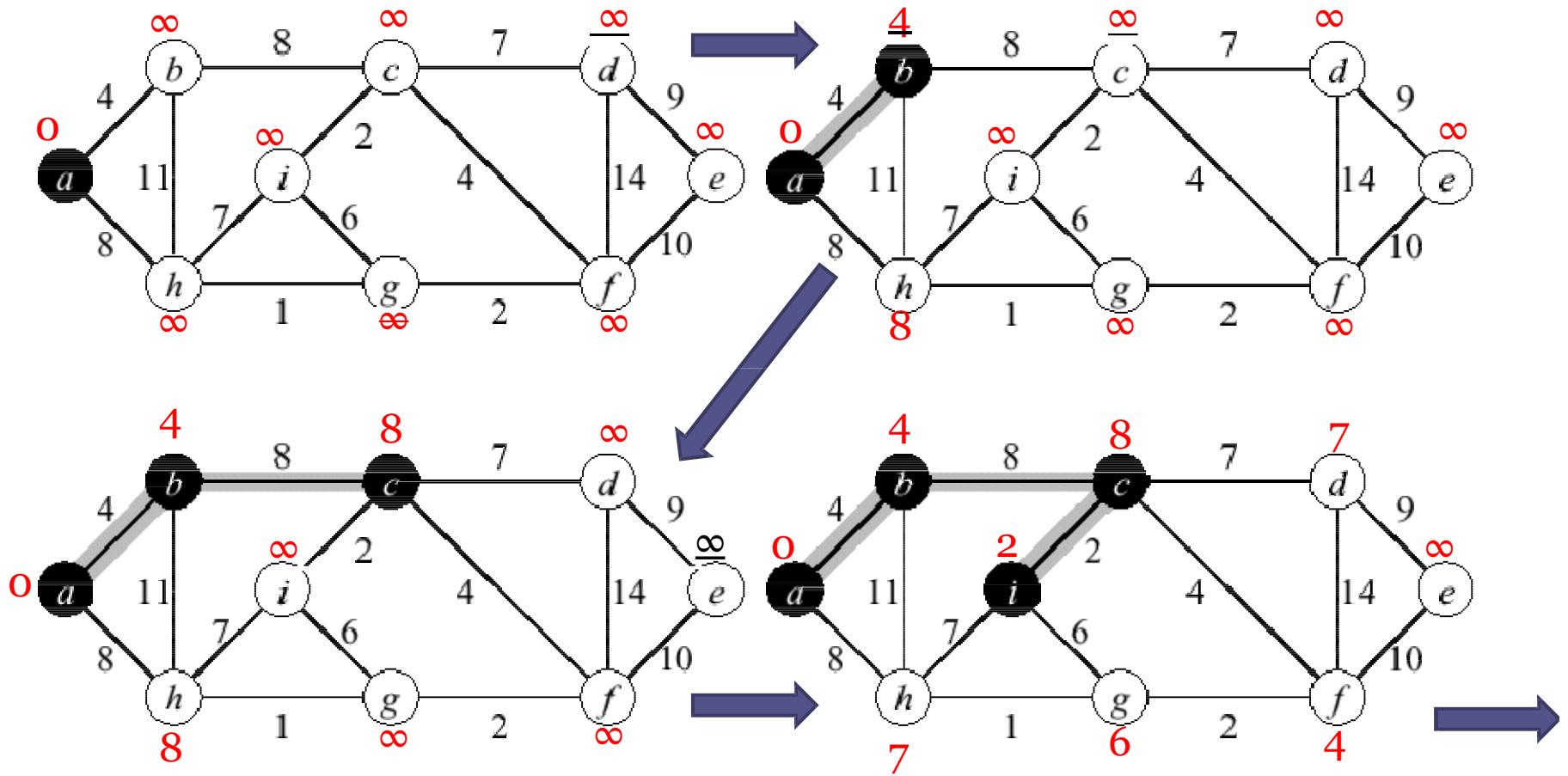
Label the vertices v outside the cloud with $key[v]$ – the minimum weight of an edge connecting v to a vertex in the cloud, $key[v] = \infty$, if no such edge exists

Prim's Algorithm

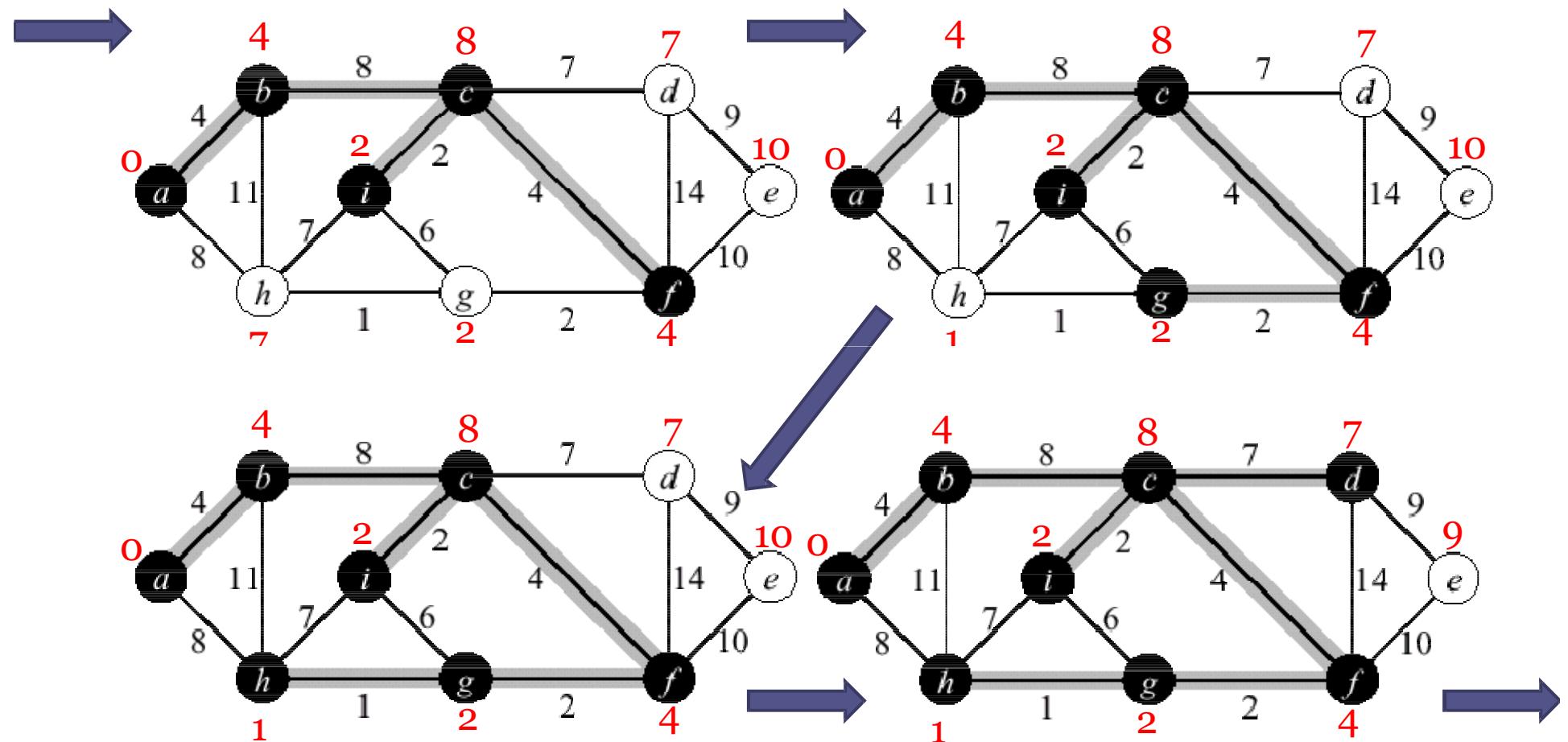
```
MST-Prim(G, w, r)
```

```
01 Q ← V[G] // Q a priority queue - vertices out of T
02 for each  $u \in Q$ 
3.   key[u] ←  $\infty$ 
4.   key[r] ← 0
5.    $\pi[r] \leftarrow NIL$ 
6. while  $Q \neq \emptyset$  do
07    $u \leftarrow ExtractMin(Q)$  // making u part of T
08     for each  $v \in Adj[u]$  do
09       if  $v \in Q$  and  $w(u, v) < key[v]$  then
10          $\pi[v] \leftarrow u$ 
11         key[v] ←  $w(u, v)$ 
```

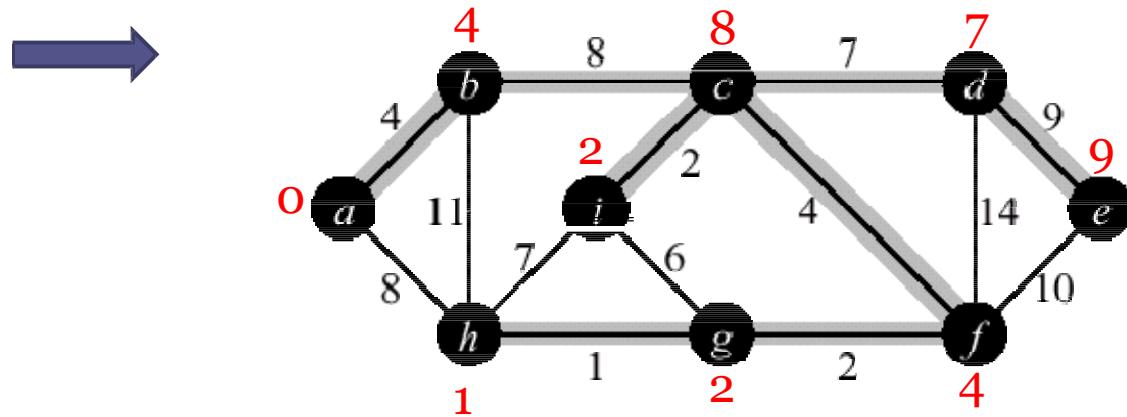
Prim's Algorithm: example



Prim's Algorithm: example...

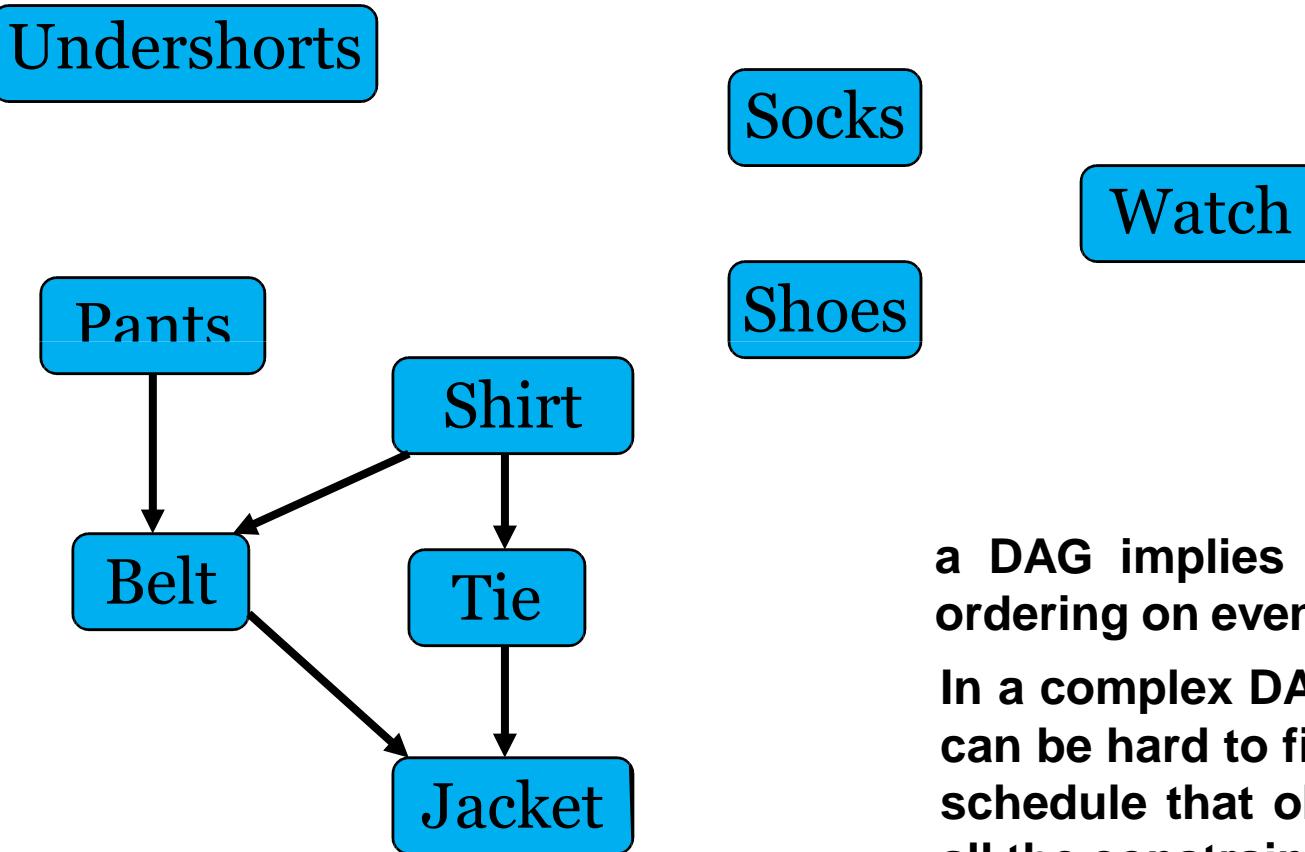


Prim's Algorithm: example...



Directed Acyclic Graph (DAG)

A Directed Graph without a cycle.



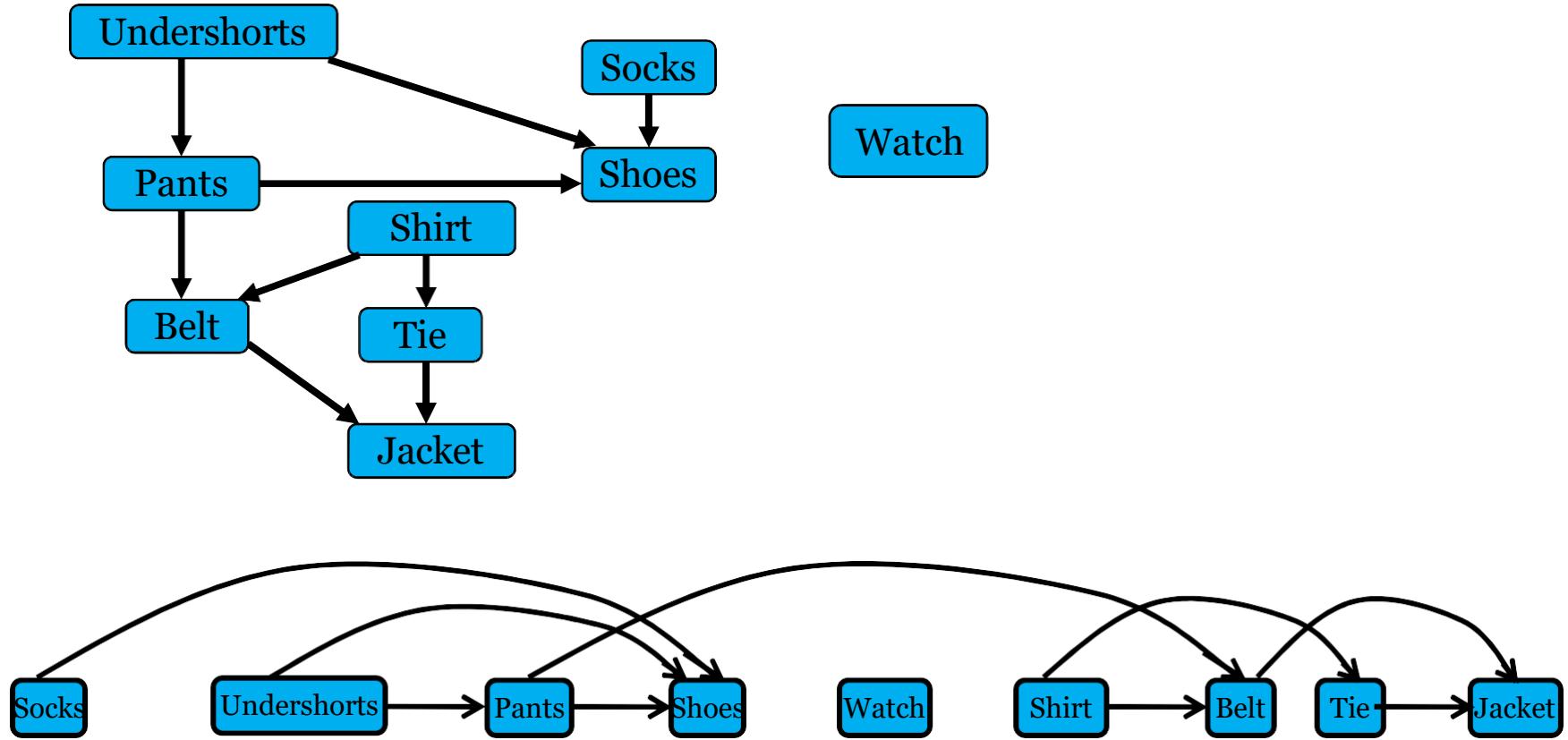
a DAG implies an ordering on events

In a complex DAG, it can be hard to find a schedule that obeys all the constraints.

Topological Sort

- For a directed acyclic graph $G = (V,E)$, a topological sort is a linear ordering of all vertices of G such that if G contains an edge (u,v) , then u appears before v in the ordering.
- A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

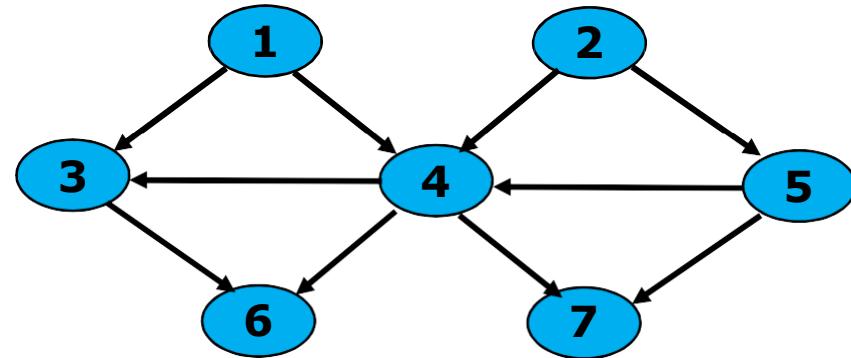
Topological Sort: Example



Topological sort

- There are often many possible topological sorts of a given DAG (*Directed Acyclic Graph*)
- Topological orders for this DAG :

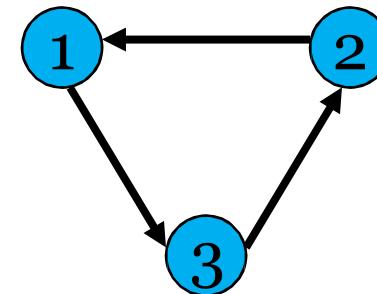
- 1,2,5,4,3,6,7
- 2,1,5,4,7,3,6
- 2,5,1,4,7,3,6
- *Etc.*



- Each topological order is a *feasible schedule*.

Topological Sorts for Cyclic Graphs?

Impossible!



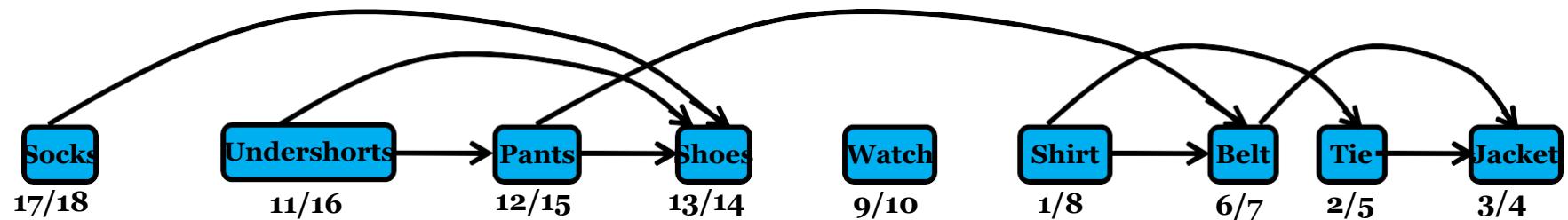
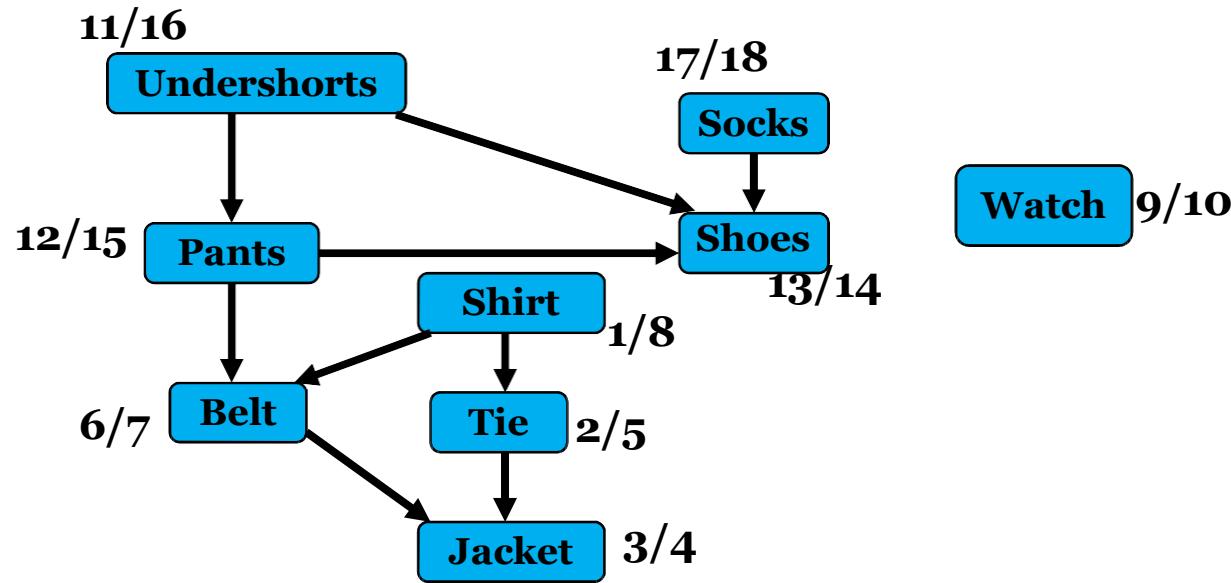
- If v and w are two vertices on a cycle, there exist paths from v to w *and* from w to v .
- Any ordering will contradict one of these paths

Topological Sort: Algorithm

TOPOLOGICAL-SORT(G)

1. Call DFS(G) to compute finishing time $f[v]$ 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.

Topological Sort



All edges of G are going from left to right only

THANKS