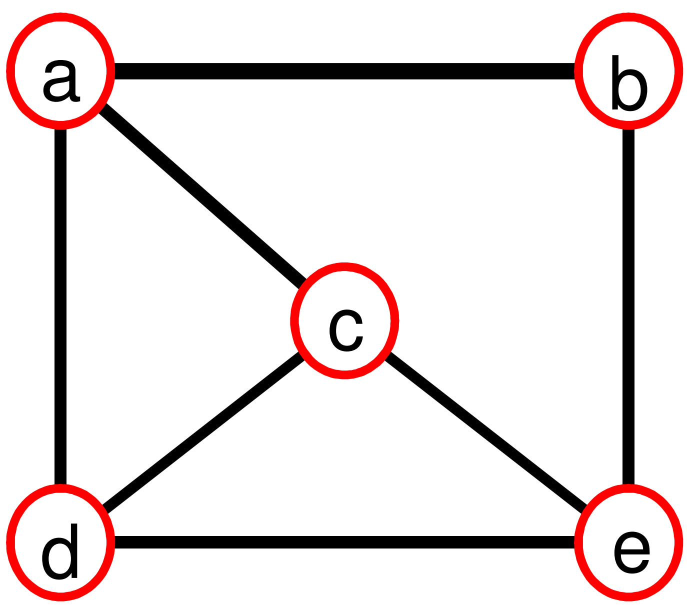
Graphs

Manoj Kumar
DTU, Delhi

What is a Graph?

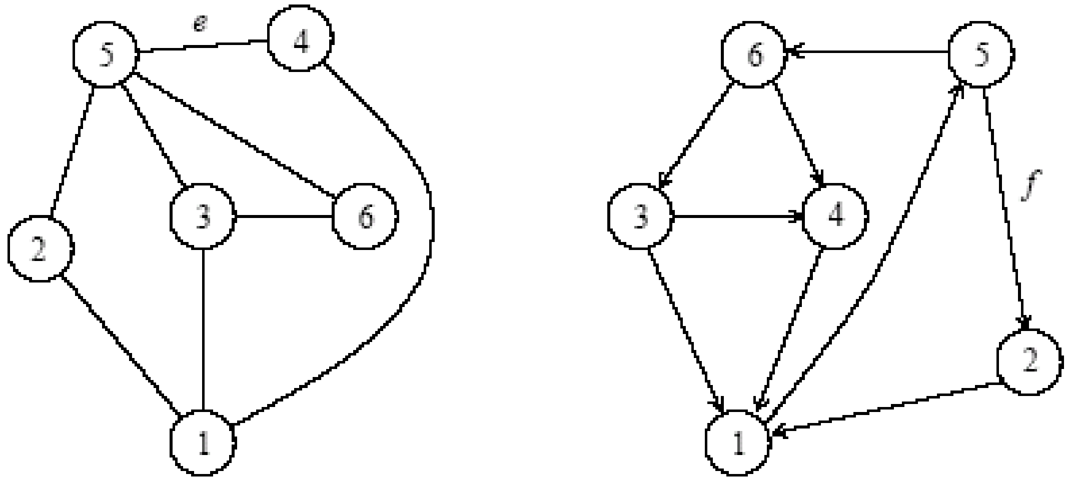
- A graph G = (V,E) is composed of:
  - V: set of vertices
  - E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices
- Example:



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

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

Directed v/s Undirected graphs

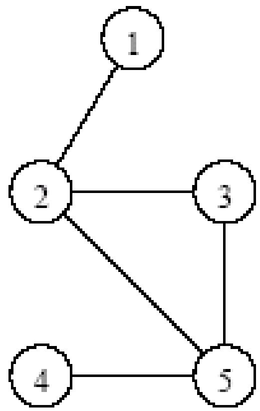


(a) (b)

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

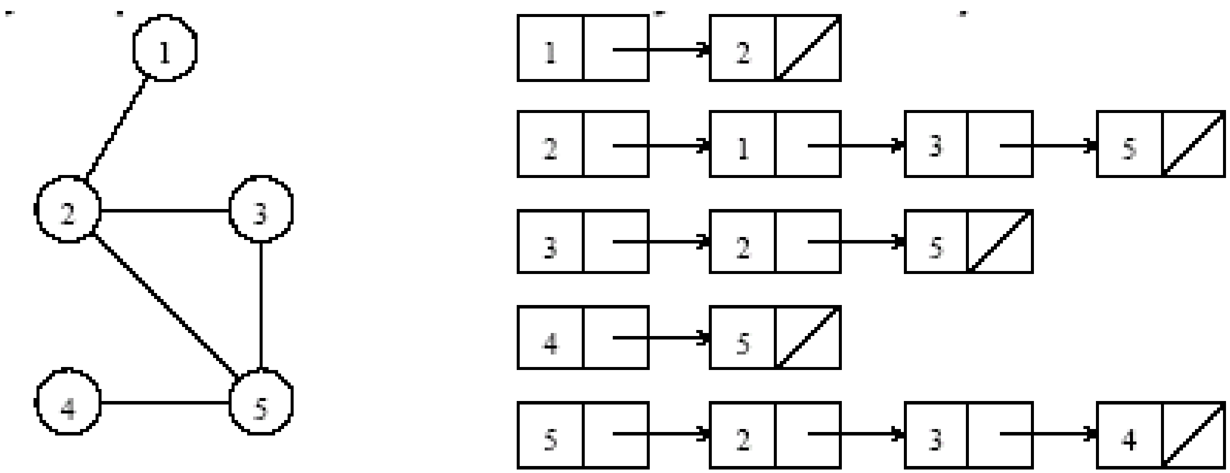
(b) A directed graph is one in which each edge is a directed pair of vertices, $<v_0, v_1> \neq <v_1,v_0>$

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.

SAMSUNG

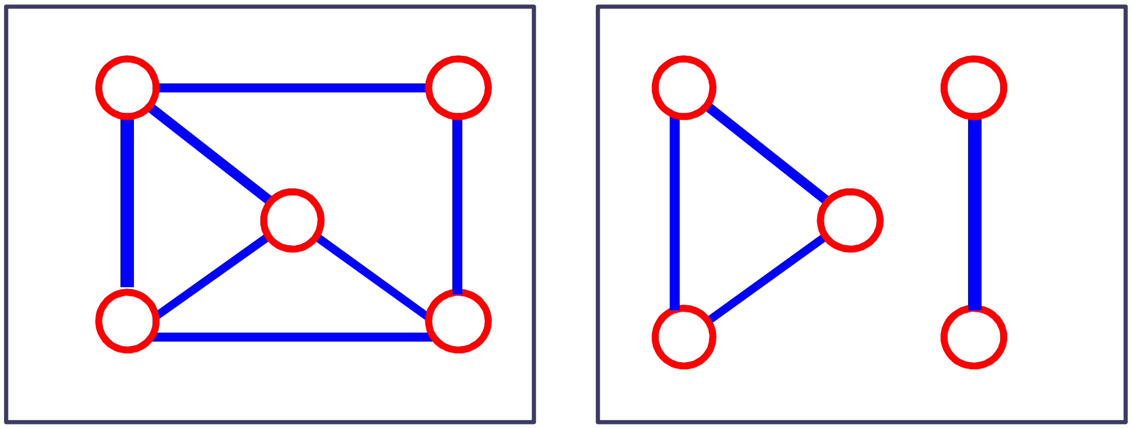
DTU Delhi Technological University

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 $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.

Connected graph

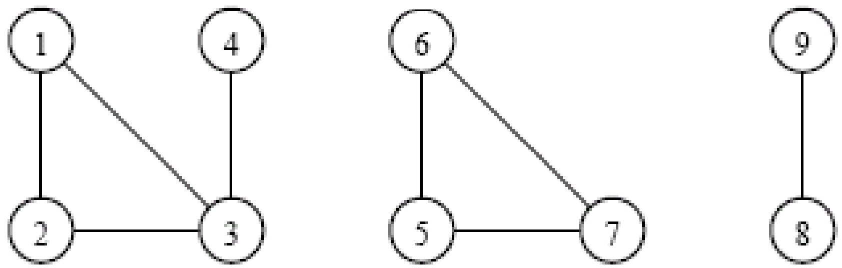
- connected graph: any two vertices are connected by some path



Connected not connected

Connected Components

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



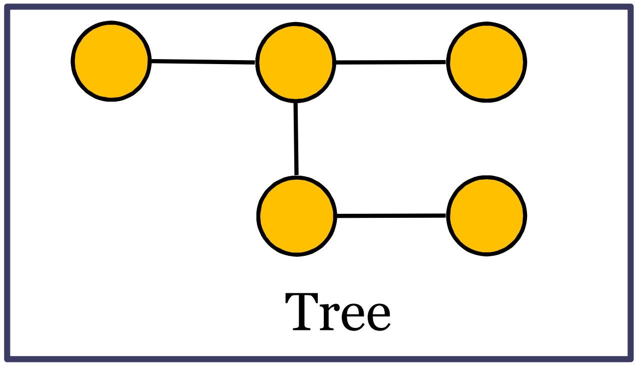
- A graph with three connected components: {1, 2, 3,4}; {5, 6, 7}; and {8, 9}.

SAMSUNG

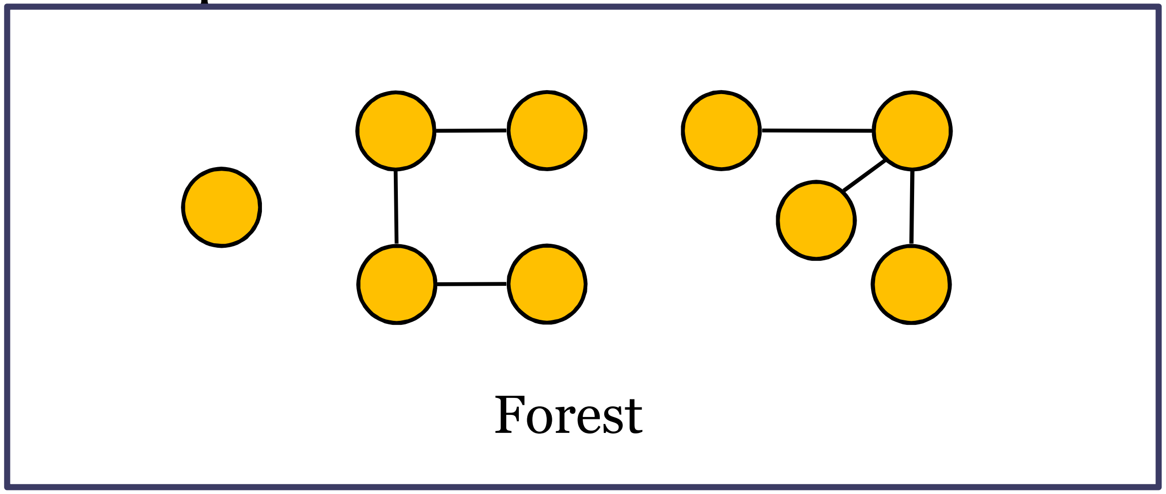
DTU Delhi Technological University

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

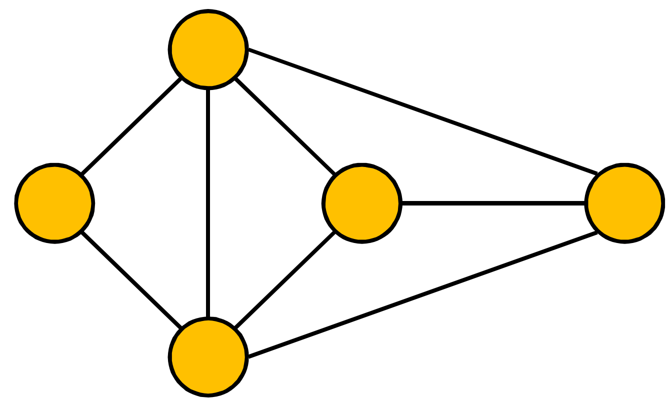


- A forest is an undirected graph without cycles
- The connected components of a forest are trees

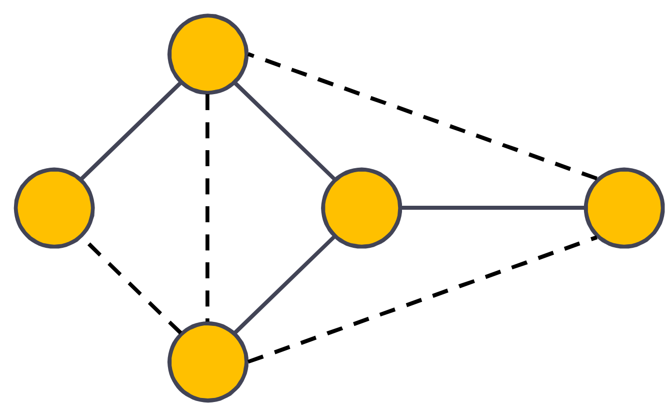


Spanning Trees and Forests

- 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



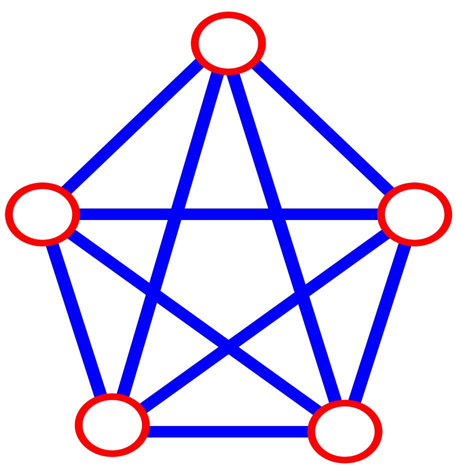
Graph



Spanning tree

Connectivity

- Let **n** = #vertices, and **m** = #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



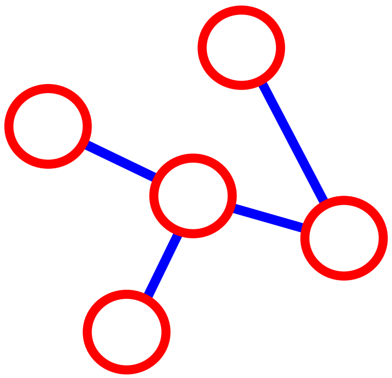
n = 5
m = (5 ∗ 4)/2 = 10

Connectivity

n = #vertices

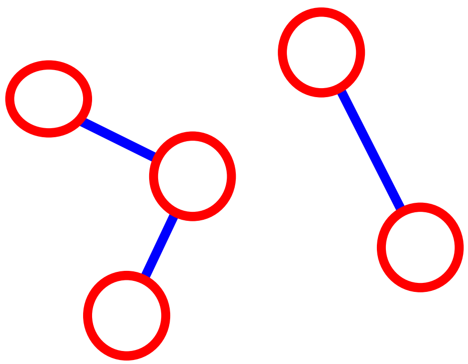
m = #edges

- For a tree **m** = **n** - 1



n = 5
m = 4

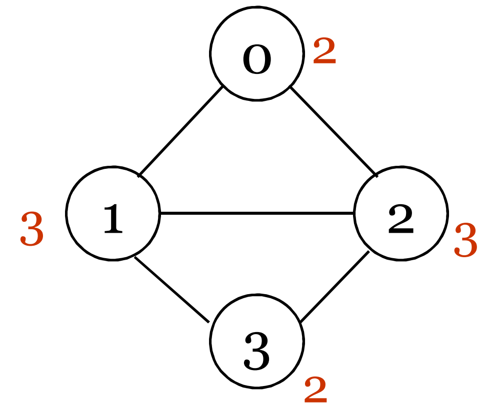
If **m** < **n** - 1, G is not connected



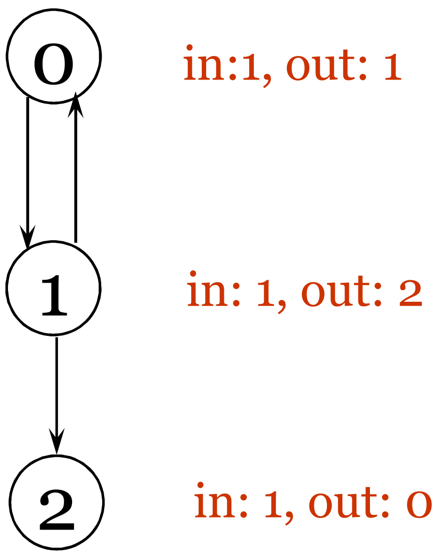
n = 5
m = 3

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.
  - $\pi[v] = u$ such that (u, v) is last edge on shortest path s v.
    - u is v's predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices.

SAMSUNG

DTU Delhi Technological University

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

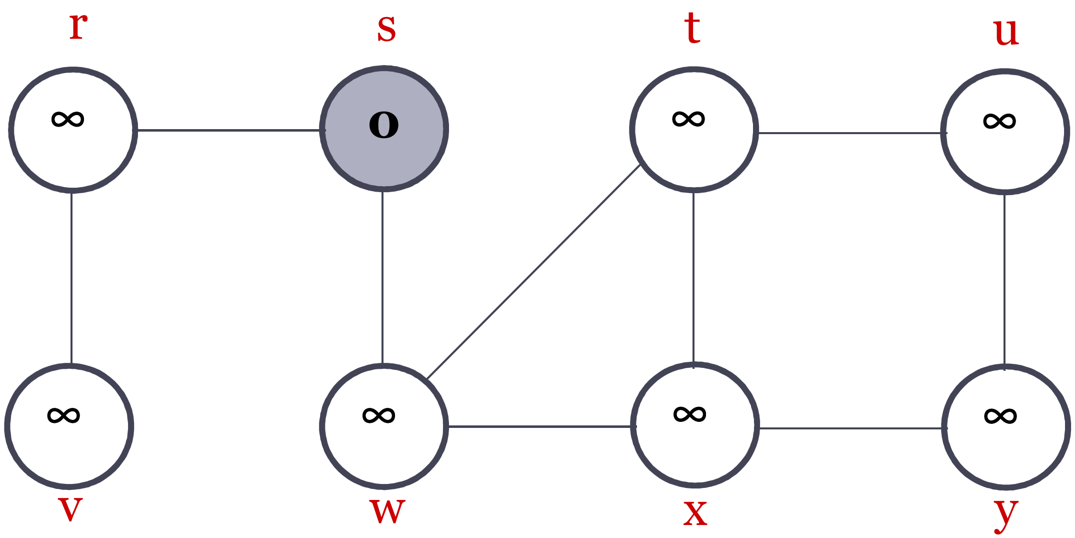
BFS(G,s)

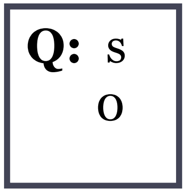
```
for each vertex u in V[G] – {s}
    do color[u] ← white
       d[u] ← ∝
       π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
    do u ← dequeue(Q)
        for each v in Adj[u]
            do if color[v] = white
                then color[v] ← gray
                     d[v] ← d[u] + 1
                     π[v] ← u
                     enqueue(Q,v)
        color[u] ← black
```

white: undiscovered
gray: discovered
black: finished

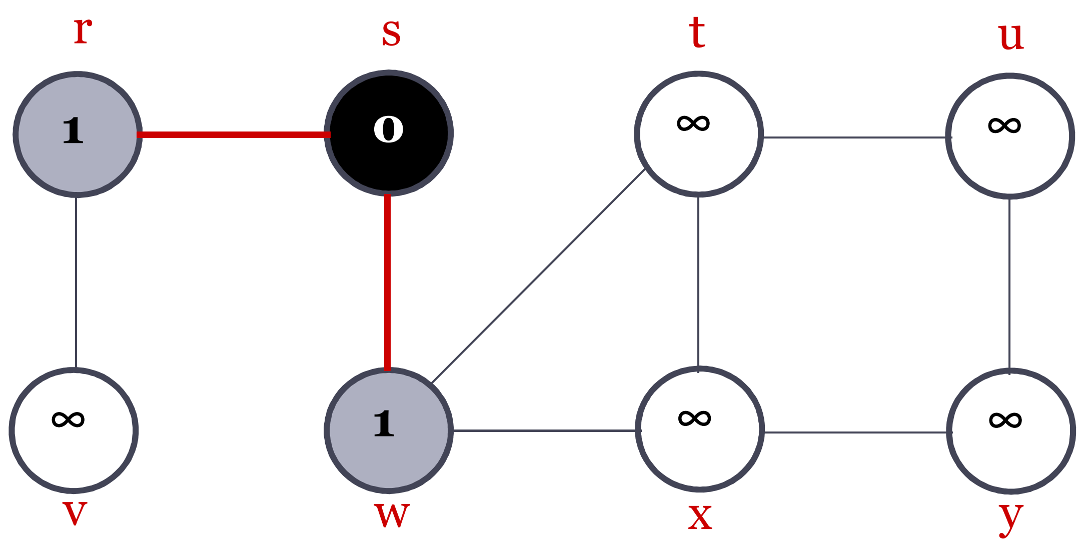
Q: a queue of discovered vertices
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

BFS: Example



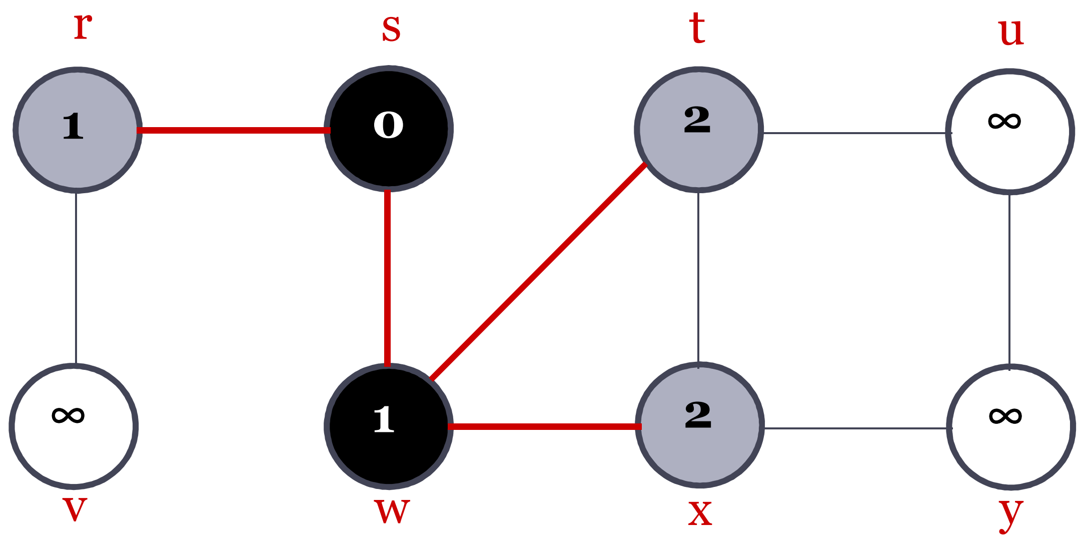


BFS: Example



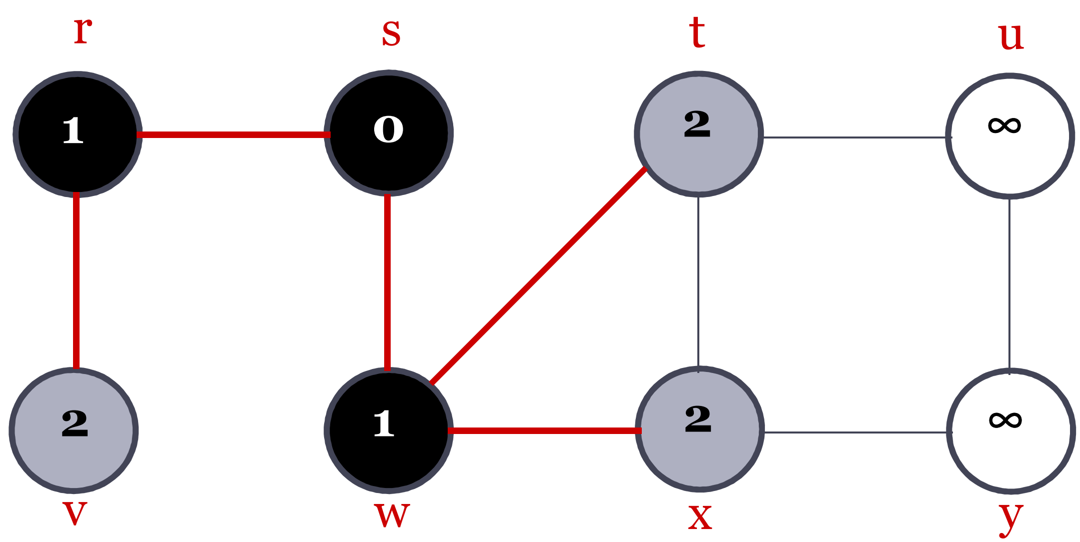
Q: w r
1 1

BFS: Example



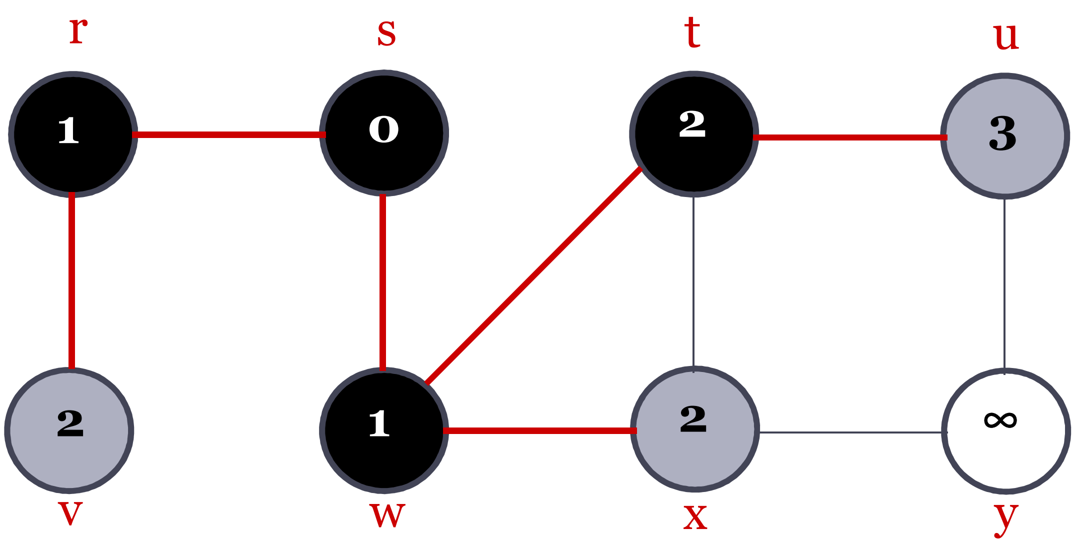
Q: r t x
1 2 2

BFS: Example



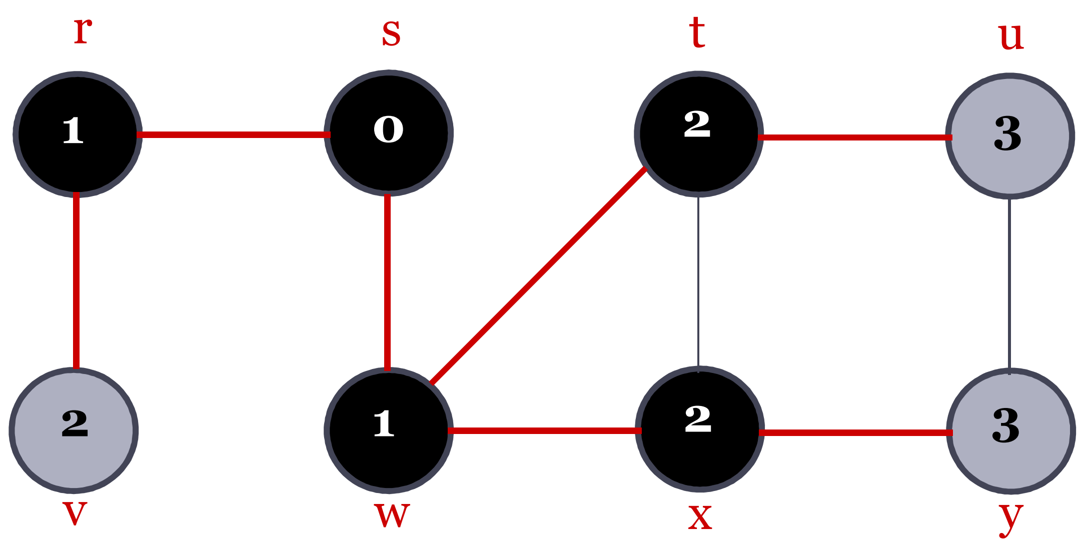
Q: t x v
2 2 2

BFS: Example



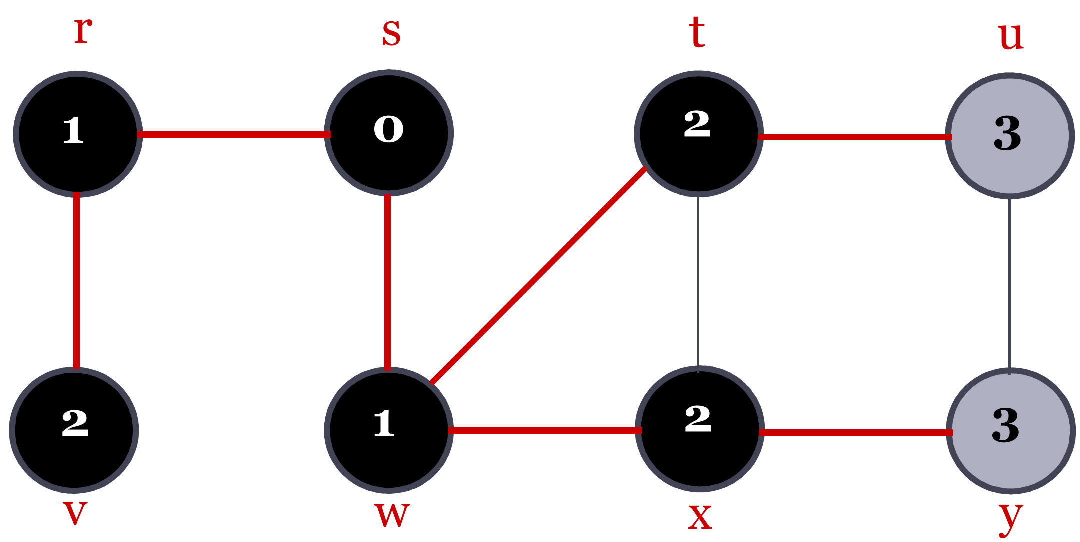
Q: x v u
2 2 3

BFS: Example



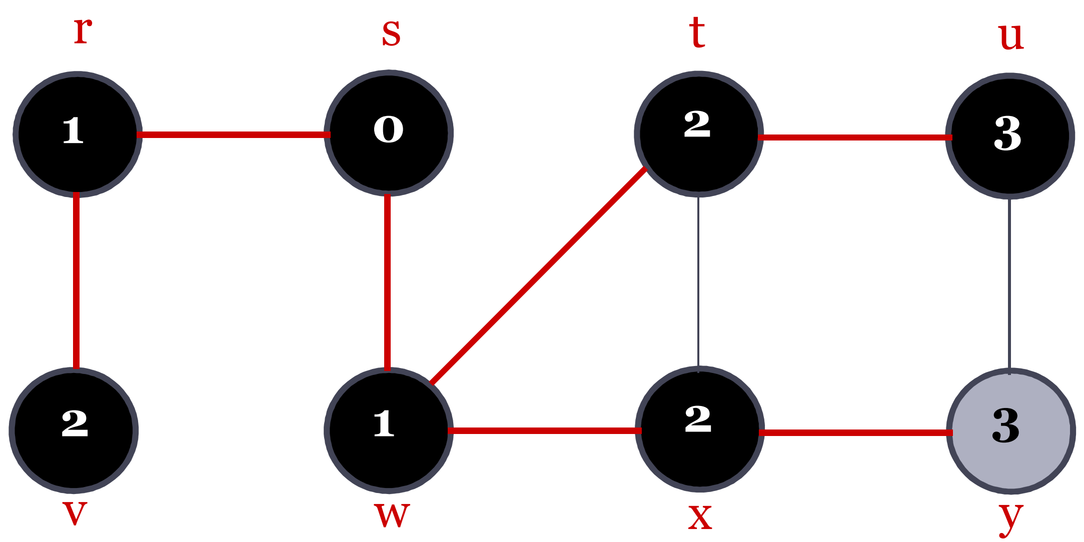
Q: v u y
2 3 3

BFS: Example



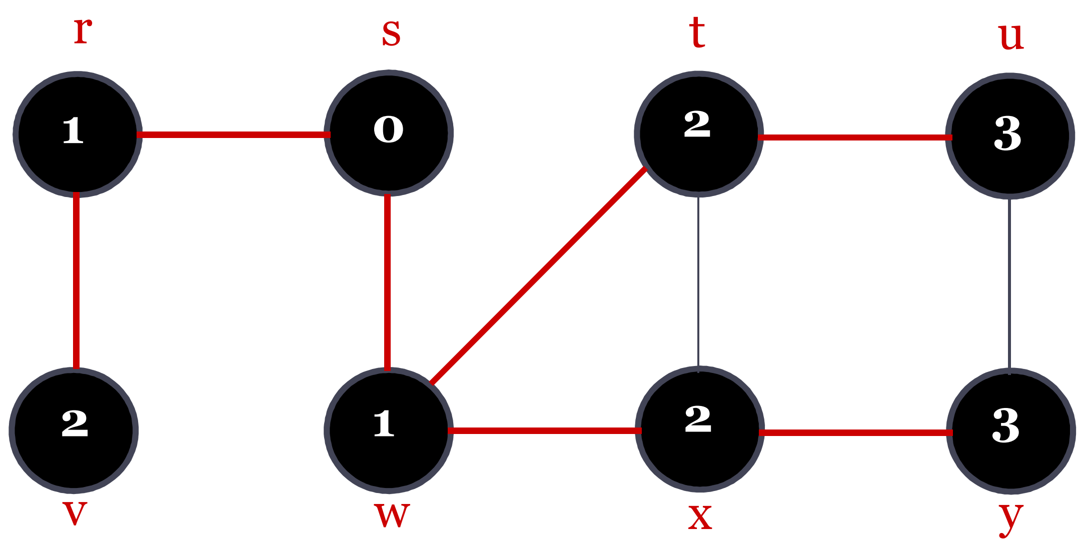
Q: u y
3 3

BFS: Example



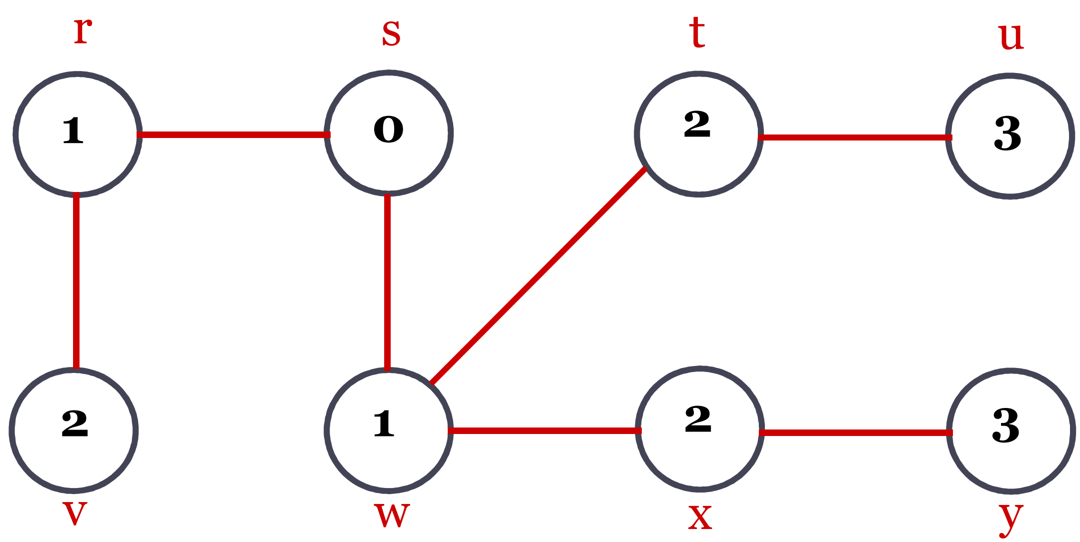
Q: y
3

BFS: Example



Q: ∅

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 => 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|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***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)

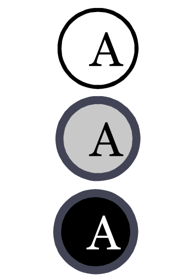
for each vertex $u \in V[G]$
  do *color*[u] $\leftarrow$ WHITE
    $\pi[u] \leftarrow$ NIL
time $\leftarrow 0$
for each vertex $u \in V[G]$
  do if *color*[u] = WHITE
    then DFS-Visit(u)

Uses a global timestamp ***time***.

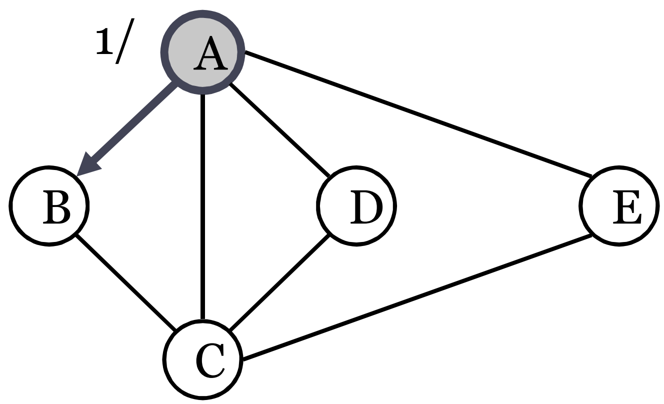
DFS-Visit(u)

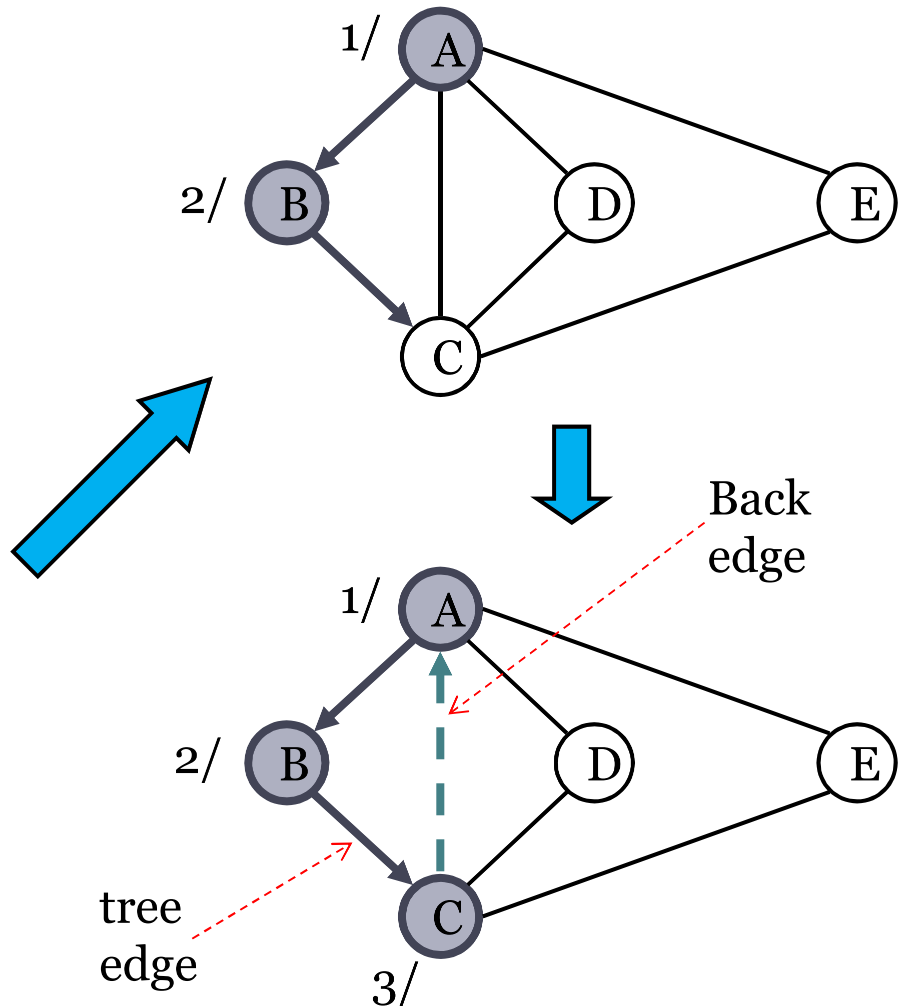
color[u] $\leftarrow$ GRAY // White vertex u has been discovered
time $\leftarrow$ *time* + 1
$d[u] \leftarrow$ *time*
for each $v \in Adj[u]$
  do if *color*[v] = WHITE
    then $\pi[v] \leftarrow u$
      DFS-Visit(v)
color[u] $\leftarrow$ BLACK // Blacken u; it is finished.
$f[u] \leftarrow$ *time*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

DFS: Example

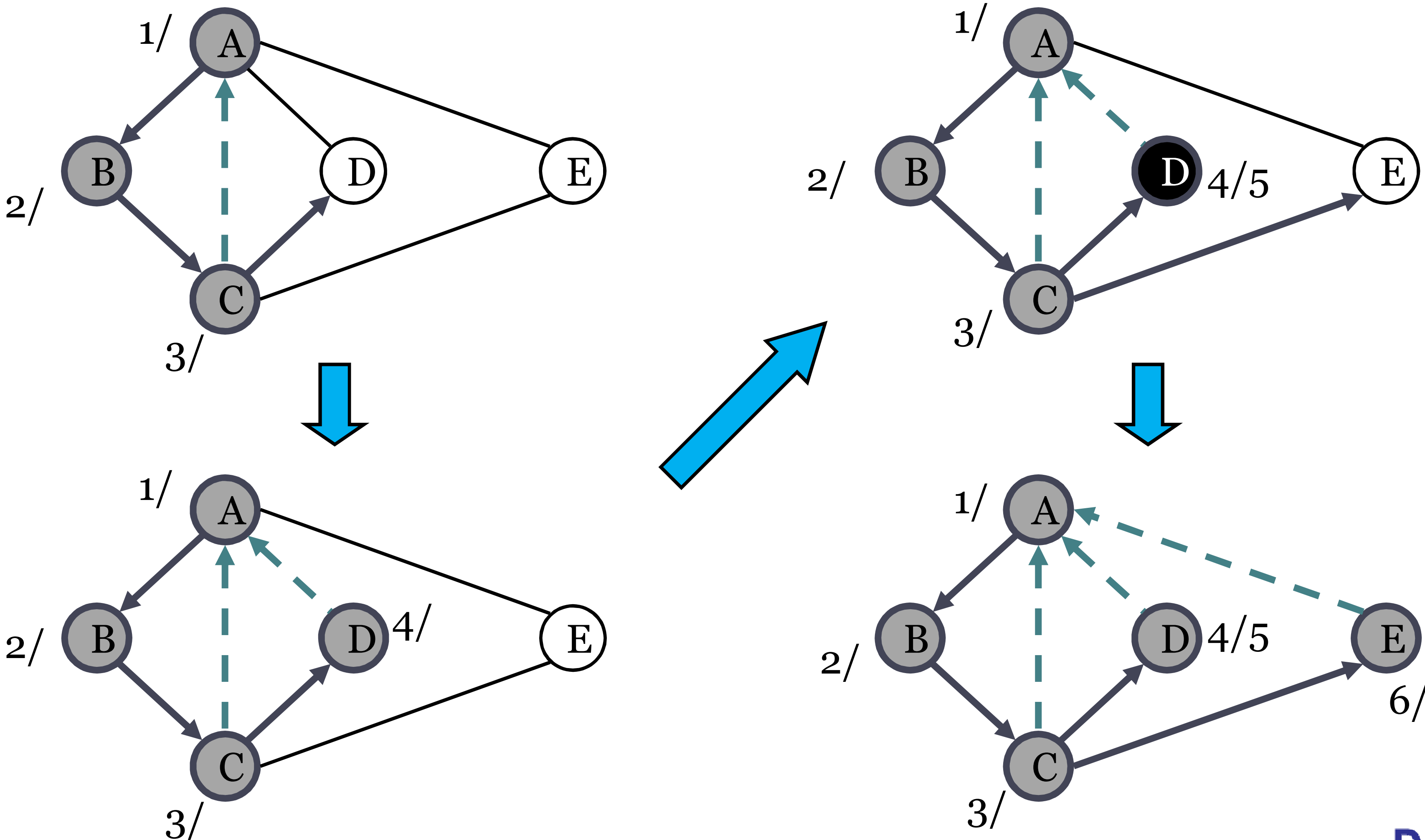


- unexplored vertex
- discovered vertex
- finished vertex
- unexplored edge
- discovery edge
- back edge

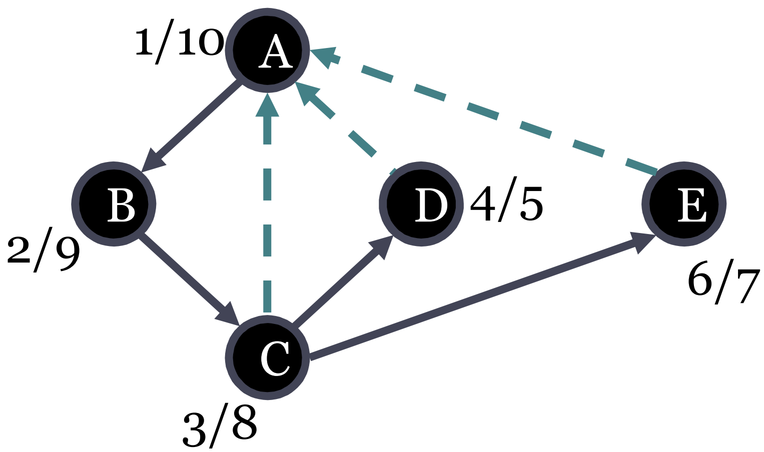




Example…



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

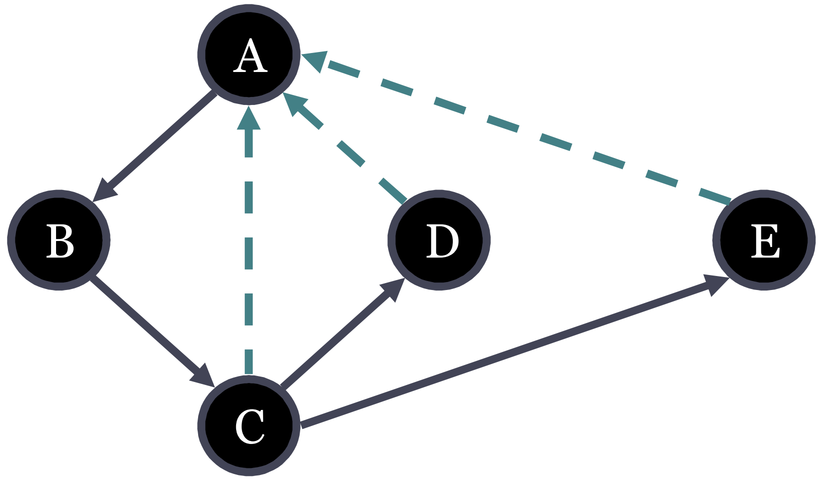
DFS-VISIT(*G, u*) visits all the vertices and edges in the connected component of ***v*****.**

Property 2

The discovery edges labeled by ***DFS-VISIT*****(*G, v*)** form a spanning tree of the connected component of ***v*****.**

Property 3

The ***DFS*****(*G*)** form a forest of spanning trees of the connected components of ***G*****.**



Analysis of DFS

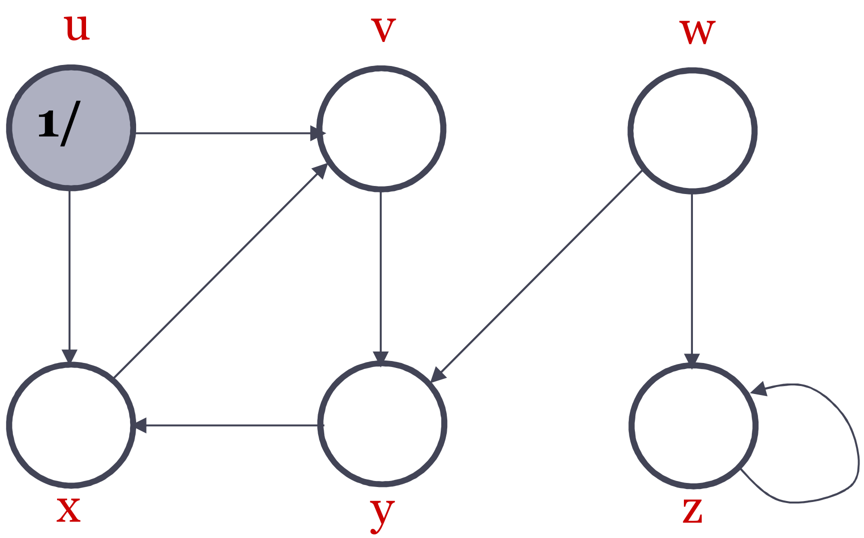
- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time.
- Lines 4-7 of DFS-Visit is executed |Adj[v]| times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$
- Total running time of DFS is $\Theta(V+E)$.

DFS on directed Graph

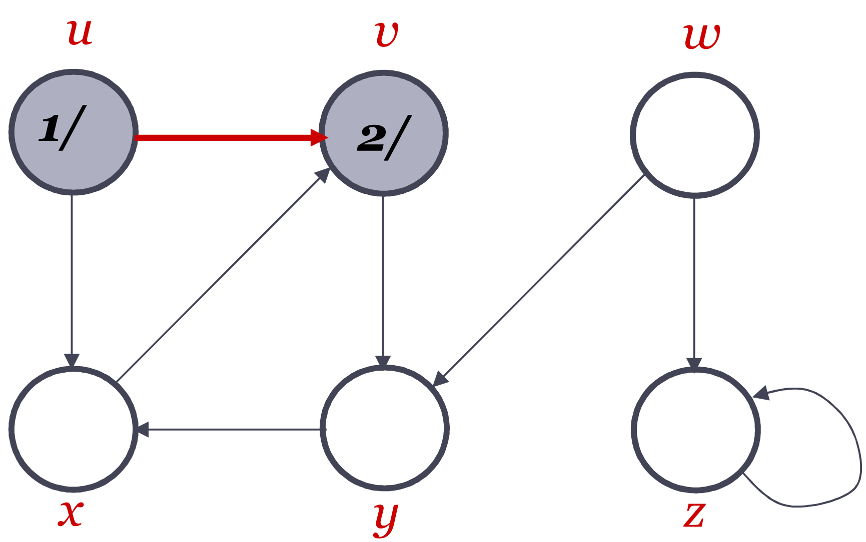
Four type of edges are produces

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

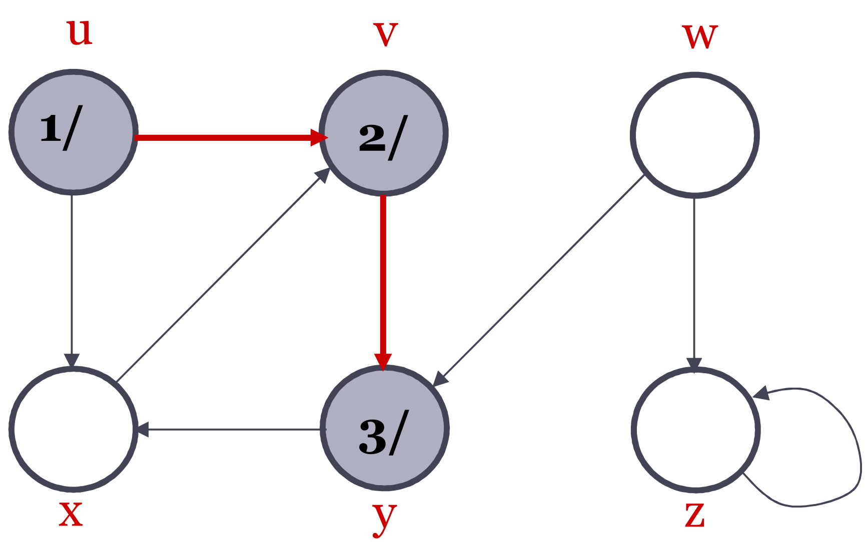


Example (DFS)



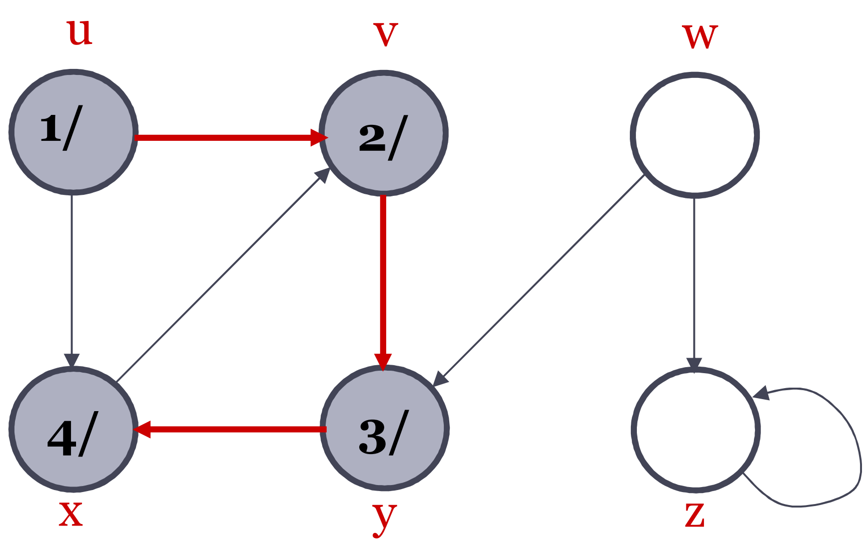
Consider edge (u,v)

Example (DFS)



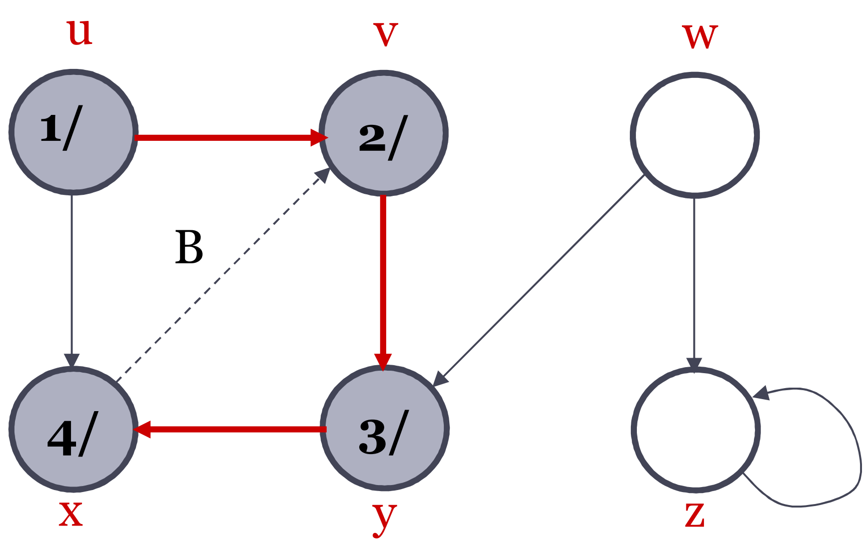
From v, Consider edge (v, y)

Example (DFS)



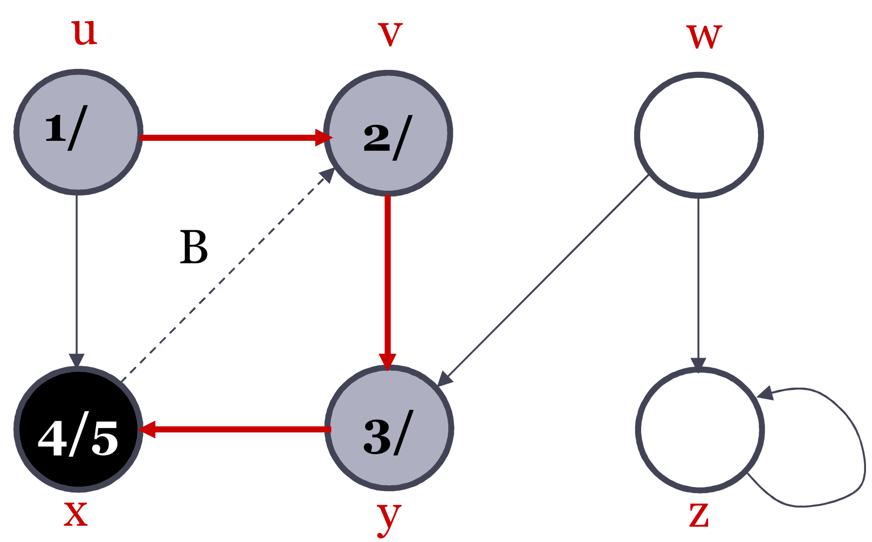
From y, Consider edge (y, x)

Example (DFS)



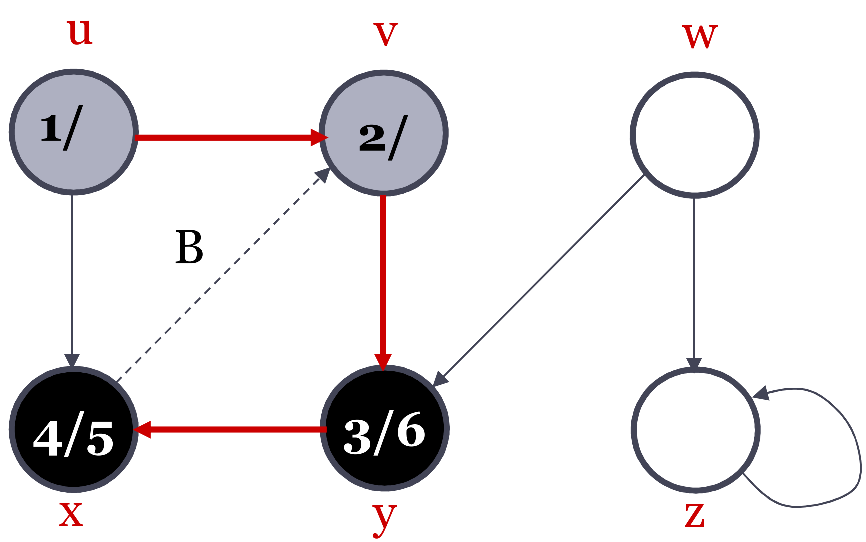
From x, Consider edge (x, v) :
do not include in tree

Example (DFS)



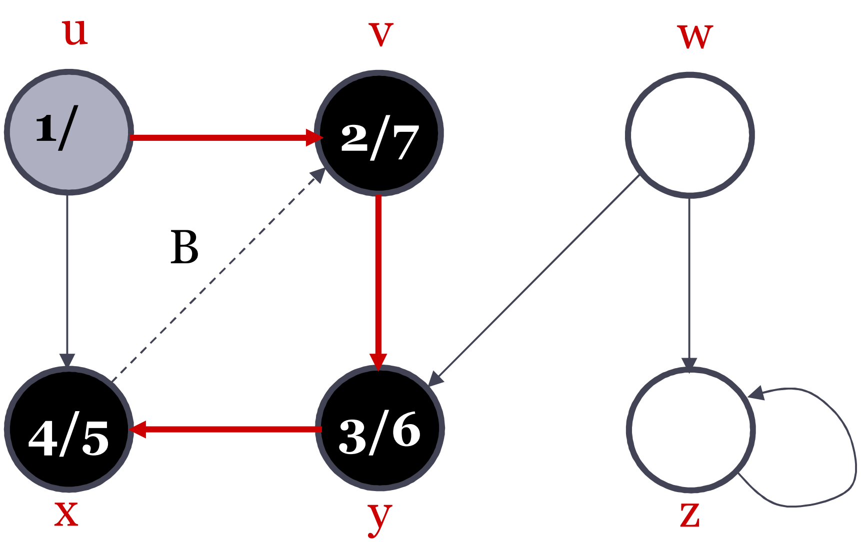
Vertex *x*, no more edges, finish it.

Example (DFS)



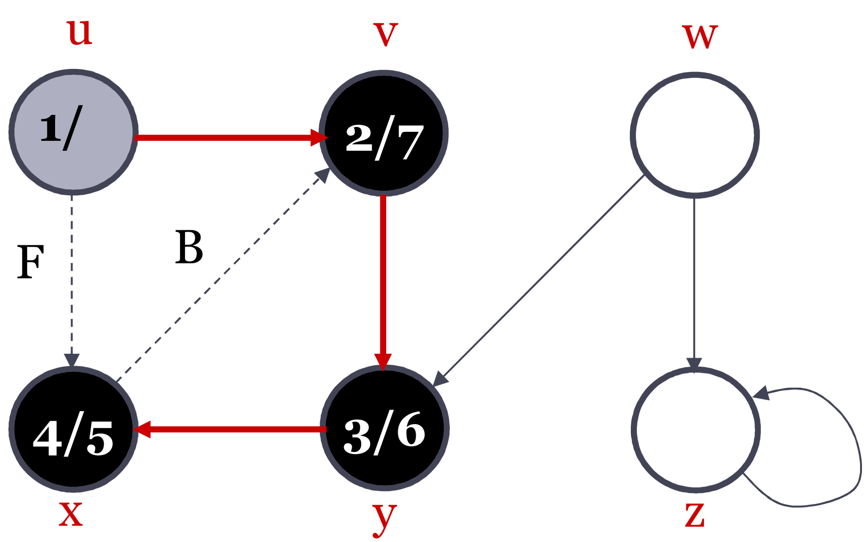
From y, no more edges, finish it

Example (DFS)



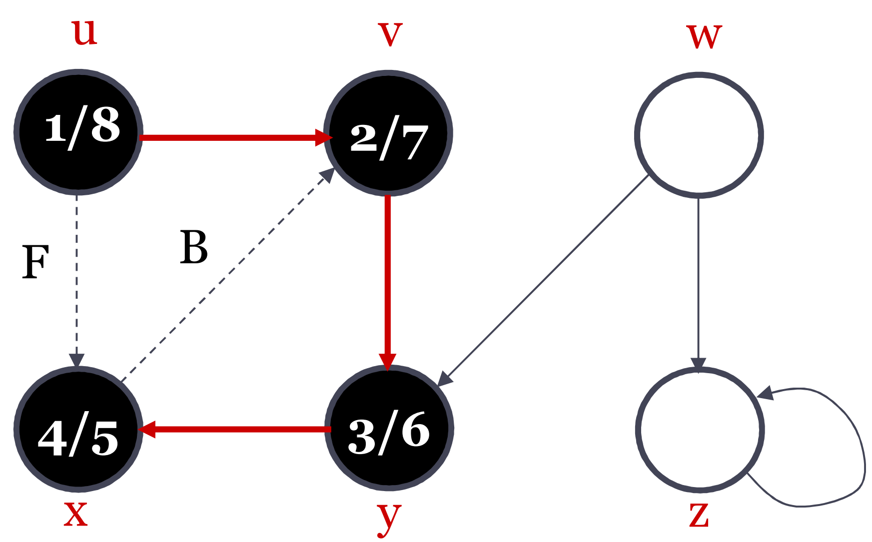
From v, no more edges, finish it

Example (DFS)



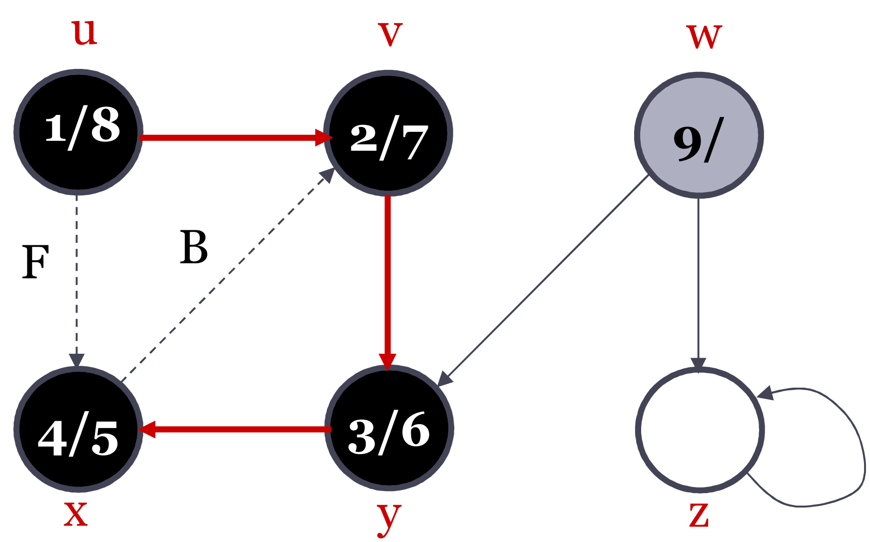
From *u*, consider edge (*u*,*x*),
do not include

Example (DFS)



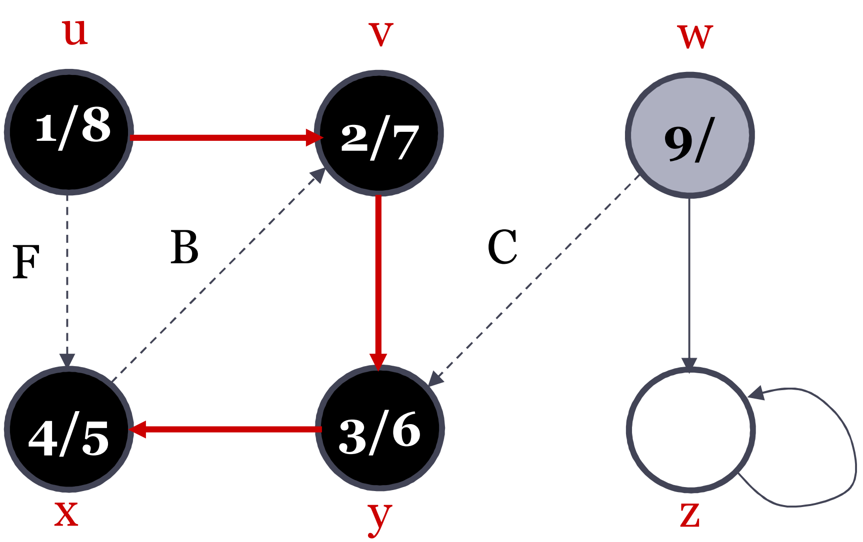
From u, no more edges, finish it

Example (DFS)



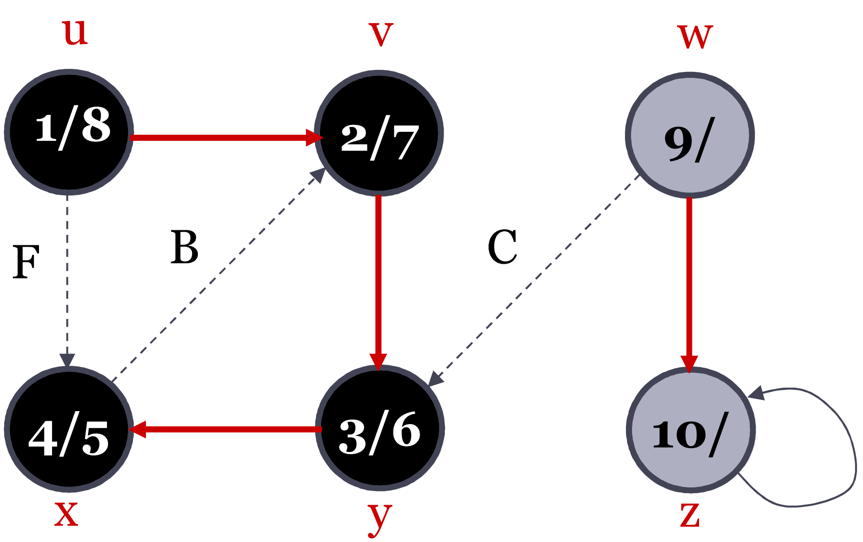
DFS from *u* ends, start again from *w*

Example (DFS)

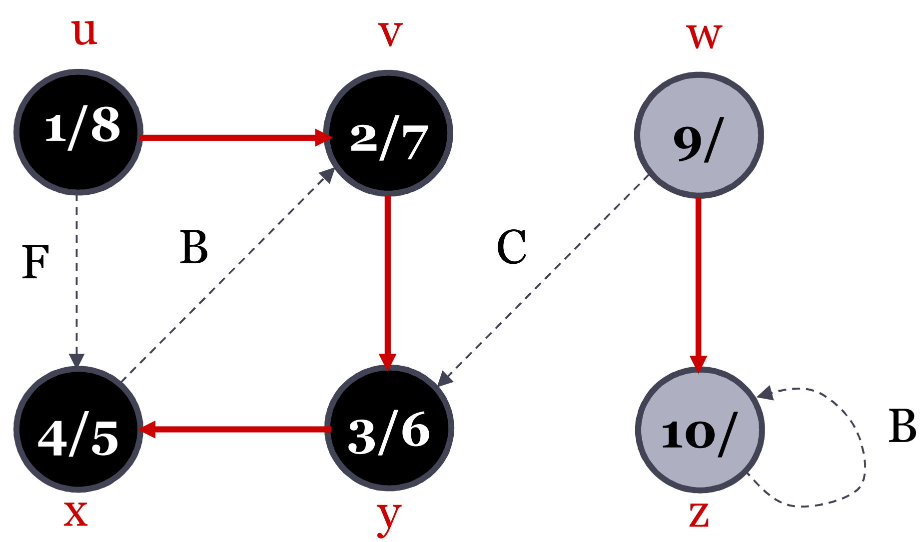


From *w* ends, Consider *(w,y)* again from *w*

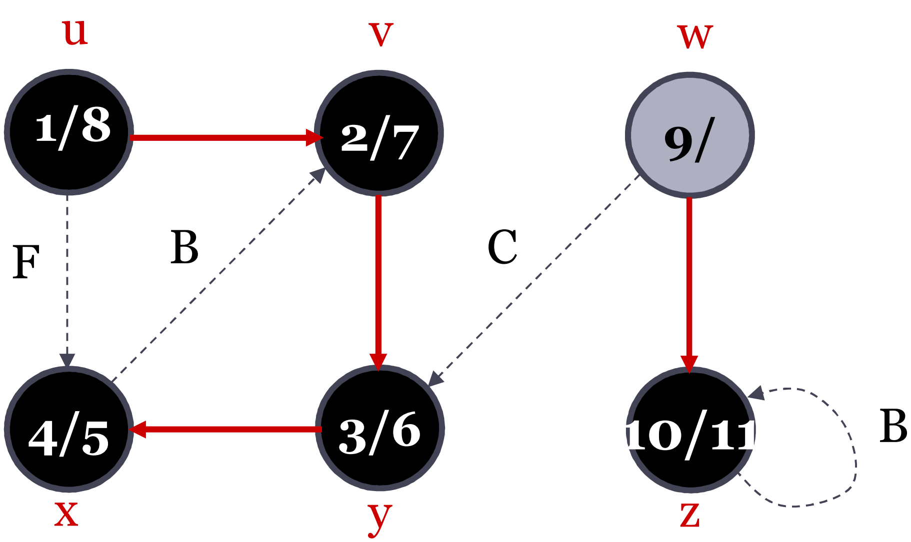
Example (DFS)



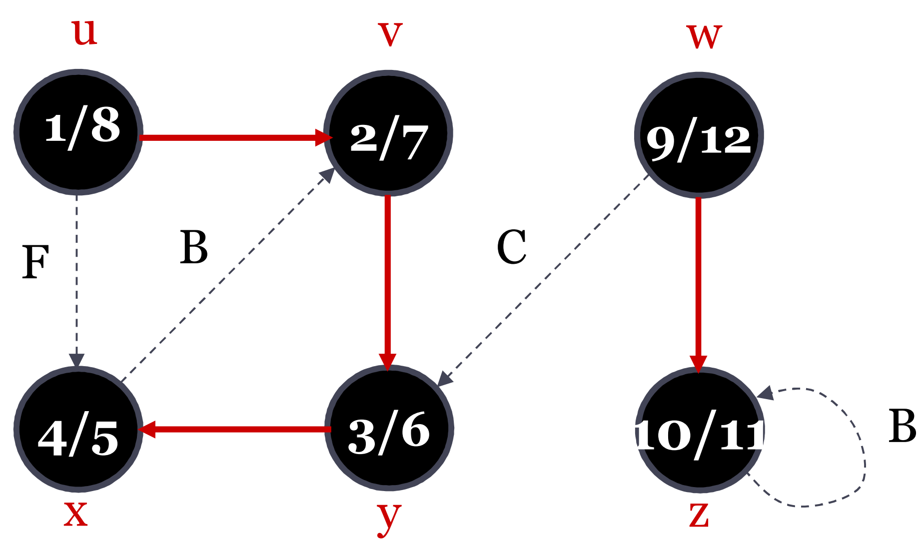
Example (DFS)



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 ***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 ***pathDFS***(***G, v, z***)

1. **for each** vertex u ϵ V[G]
2. **do** color[u] ← WHITE
3. Done=FALSE
4. pathDFS-VISIT(G,v,z)

Algorithm ***pathDFS-VISIT***(***G, v, z***)

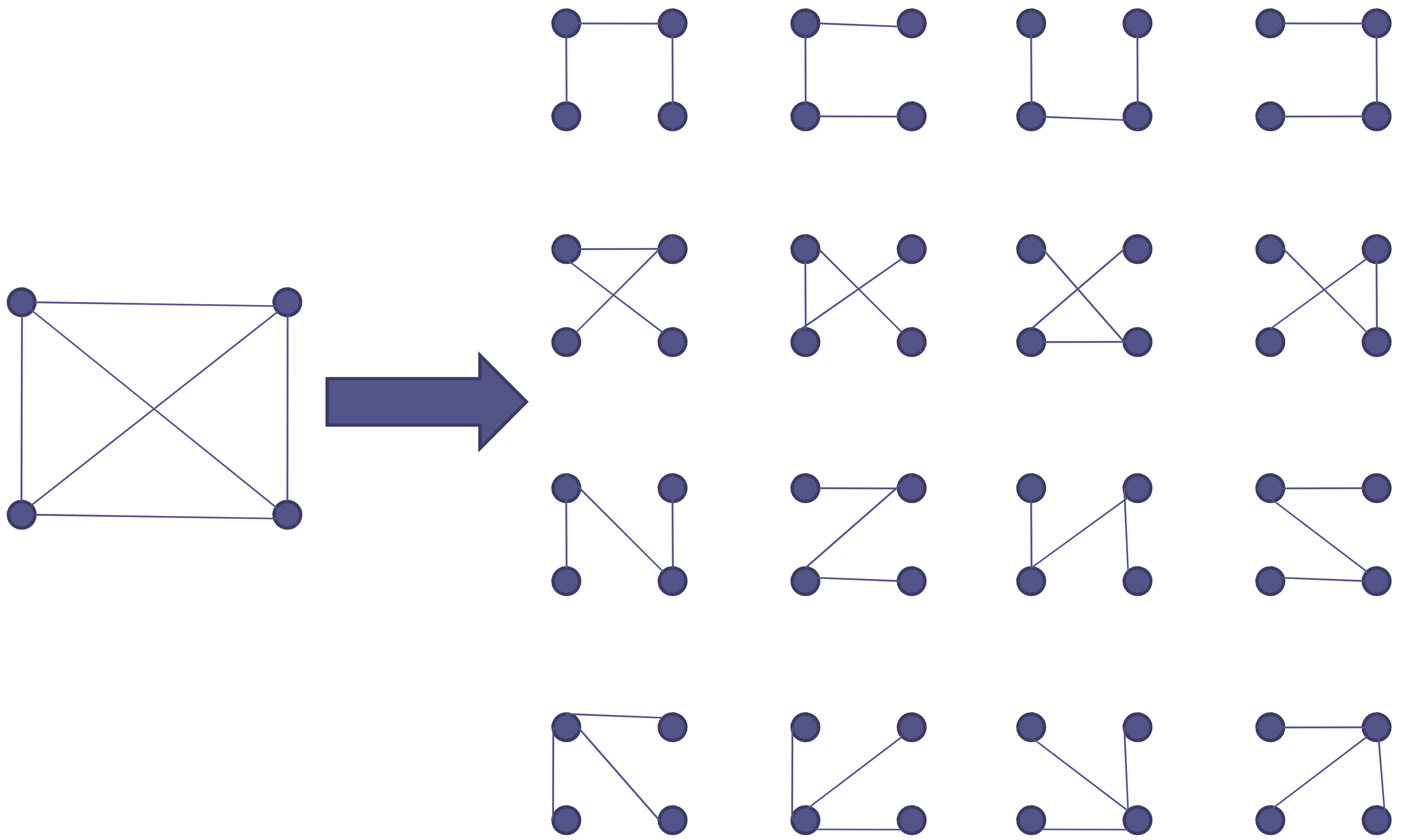
1. Color[v] ← GRAY
2. *S.push*(*v*)
3. If (*v* = *z)*
4. ***THEN*** *Done=TRUE*
5. return S.elements
6. **for each** *u* ∈ *Adj[v]*
7. **do** **if** (color[u] = WHITE)
8. **THEN** pathDFS(G,u,z)
9. **if** (Done) **THEN** return;
10. S.pop()
11. Color[v] ← BLACK

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:

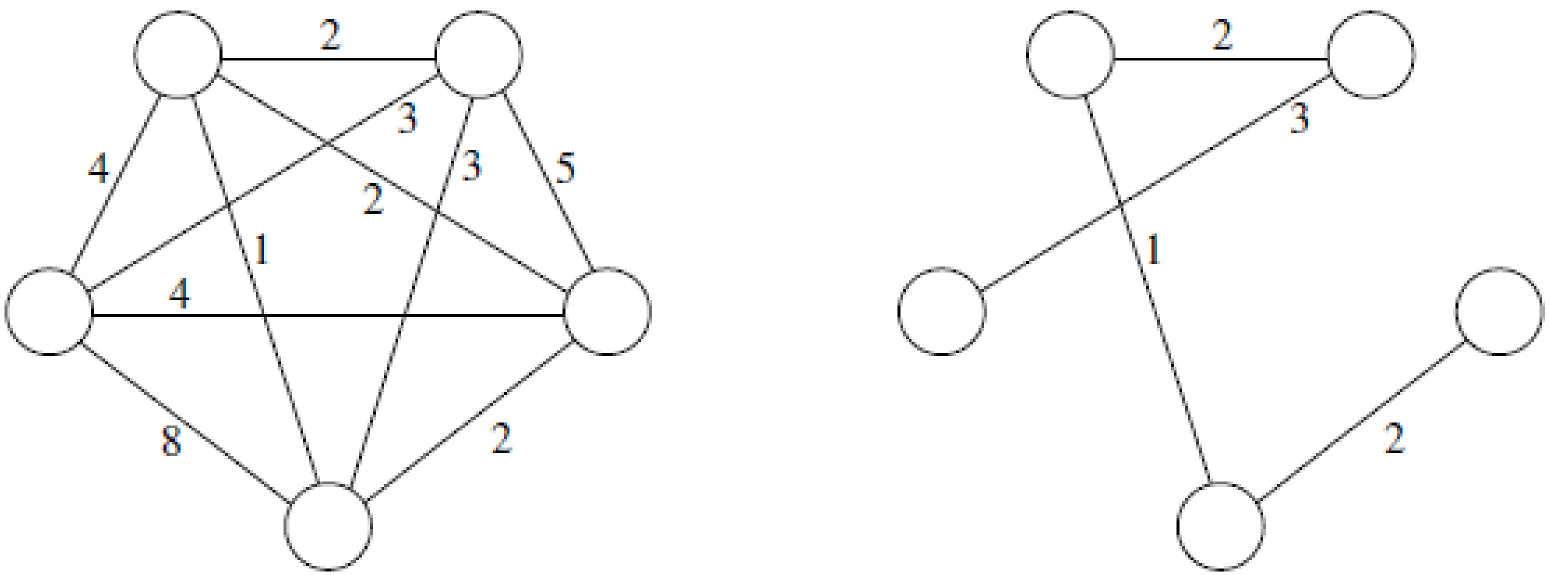
Spanning trees



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

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

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 it if 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

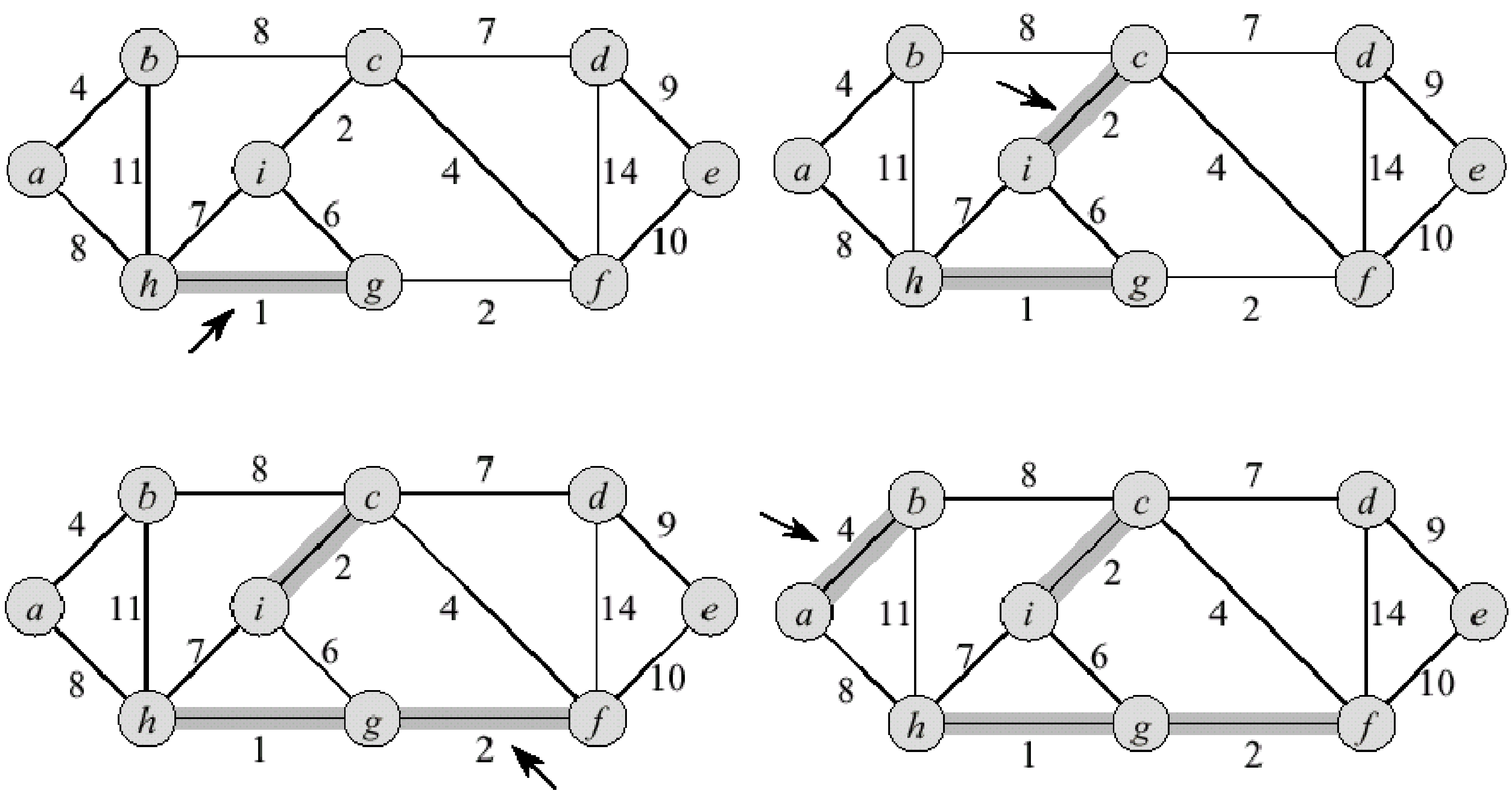
- The algorithm adds the cheapest edge that connects two trees of the forest

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

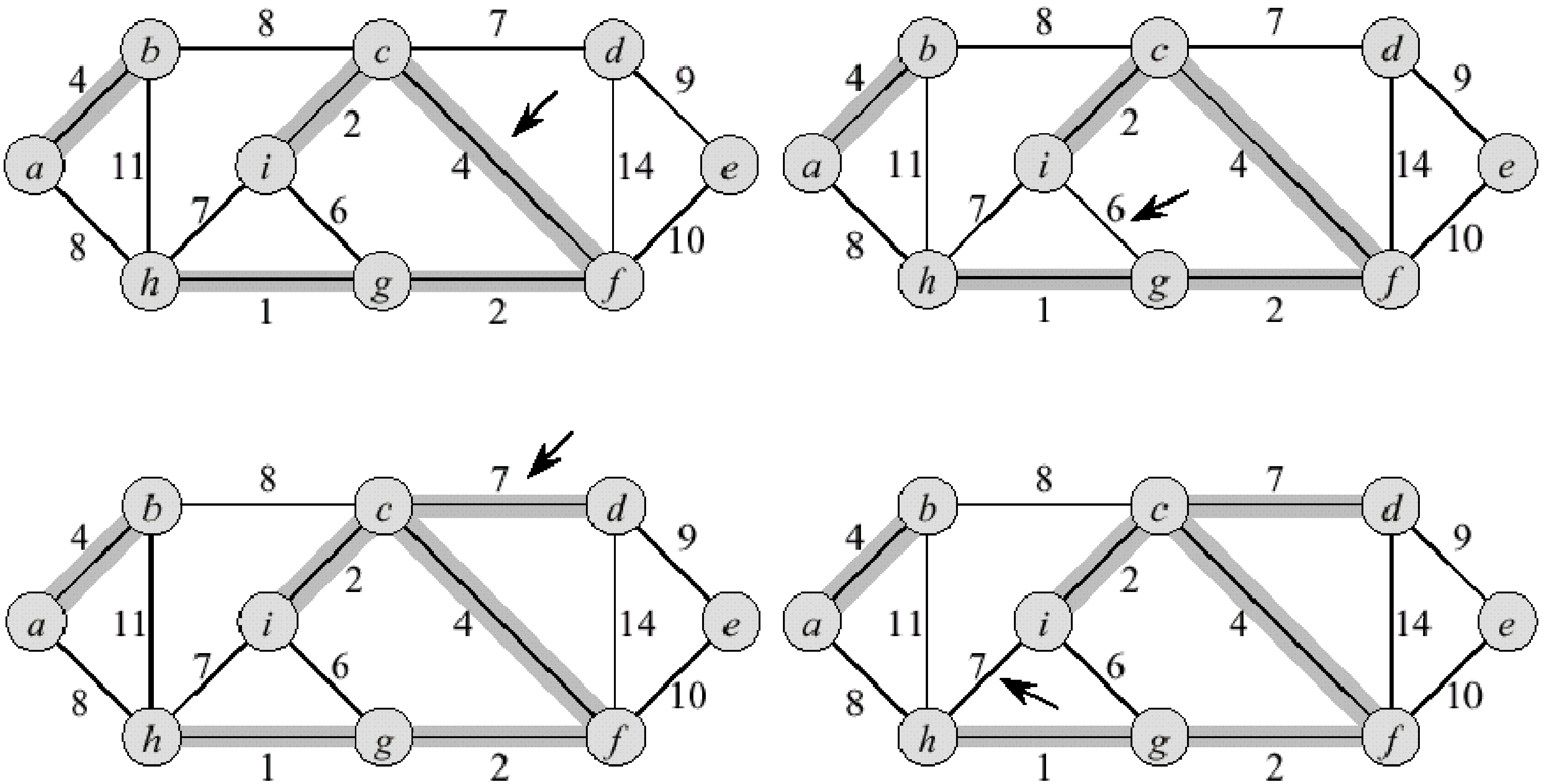
SAMSUNG

DTU
Delhi Technological University

Kruskal’s algorithm: example



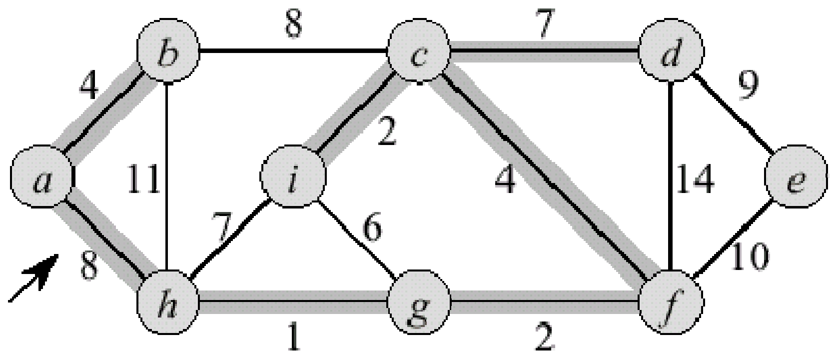
Kruskal's algorithm: example…

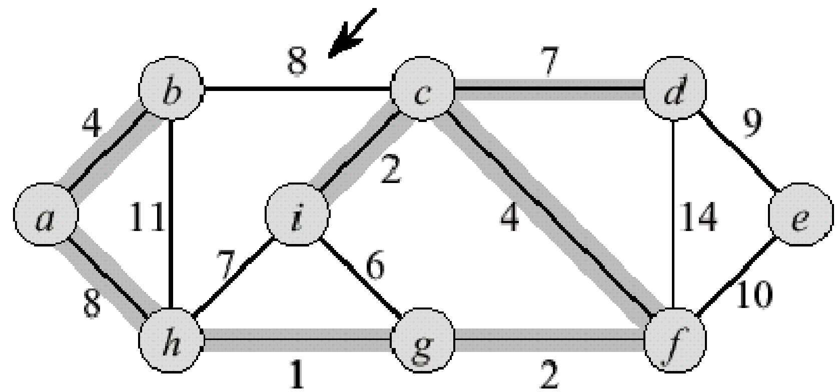


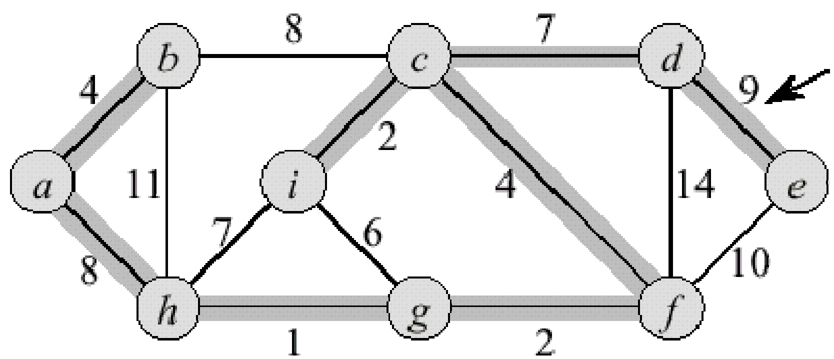
SAMSUNG

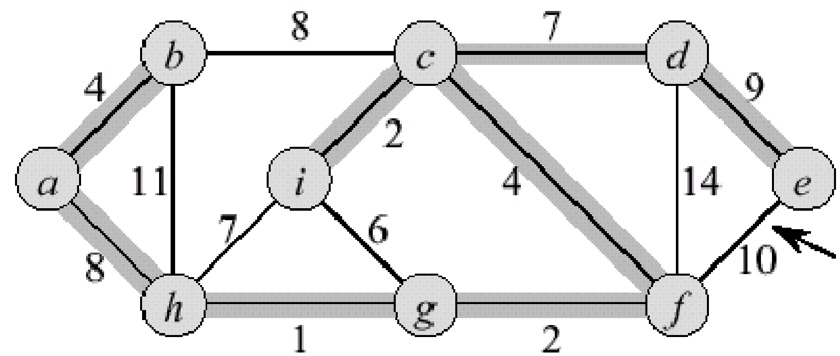
DTU Delhi Technological University

Kruskal's algorithm: example…

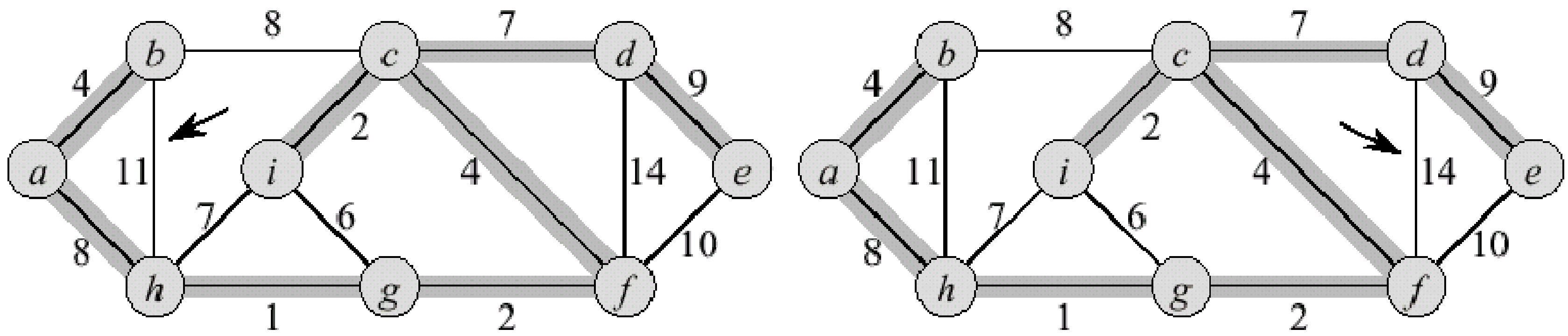








Kruskal's algorithm: example…



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 weigth of an edge connecting v to a vertex in the cloud, $key[v] = \infty$, if no such edge exists

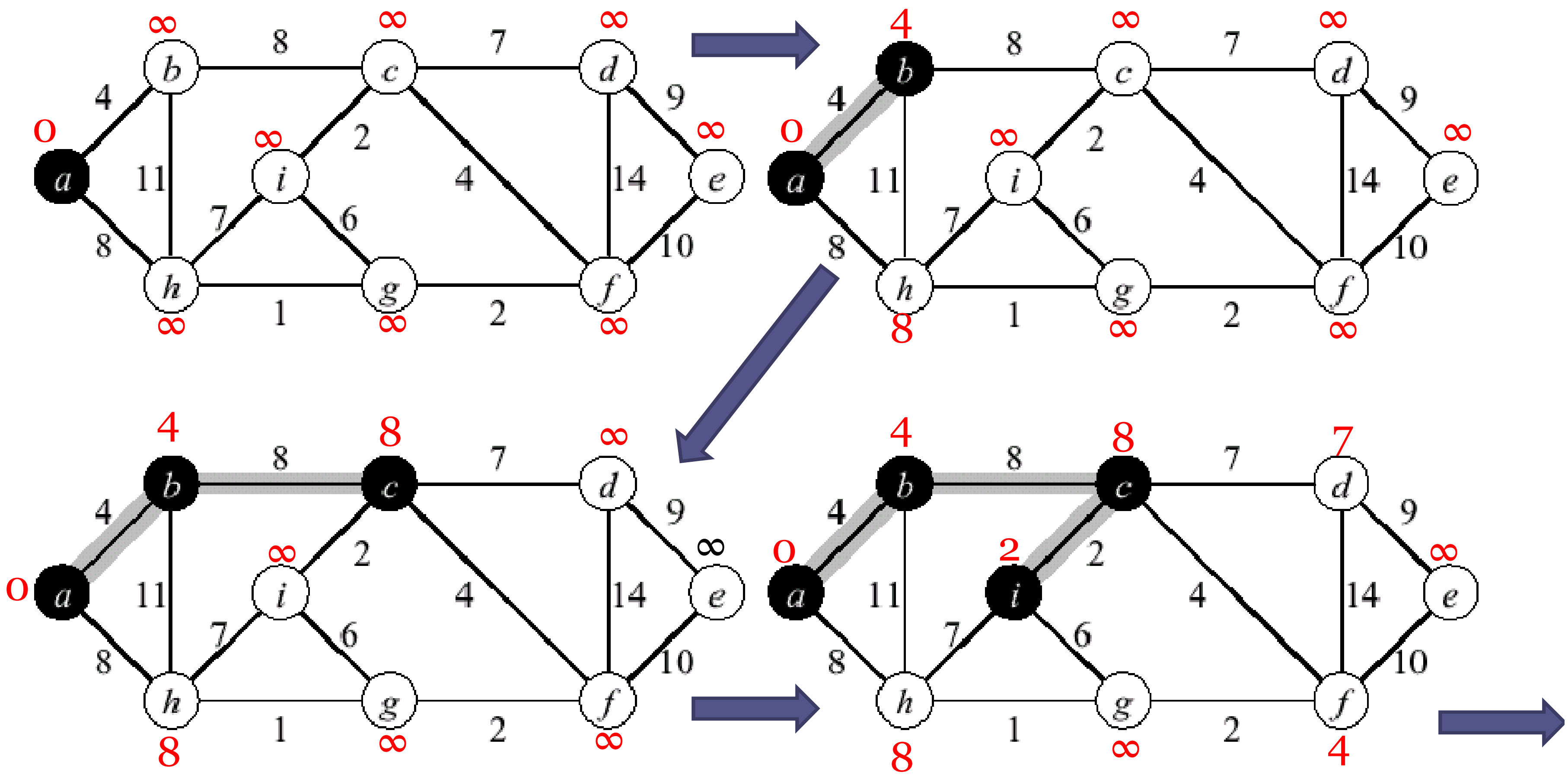
SAMSUNG

DTU Delhi Technological University

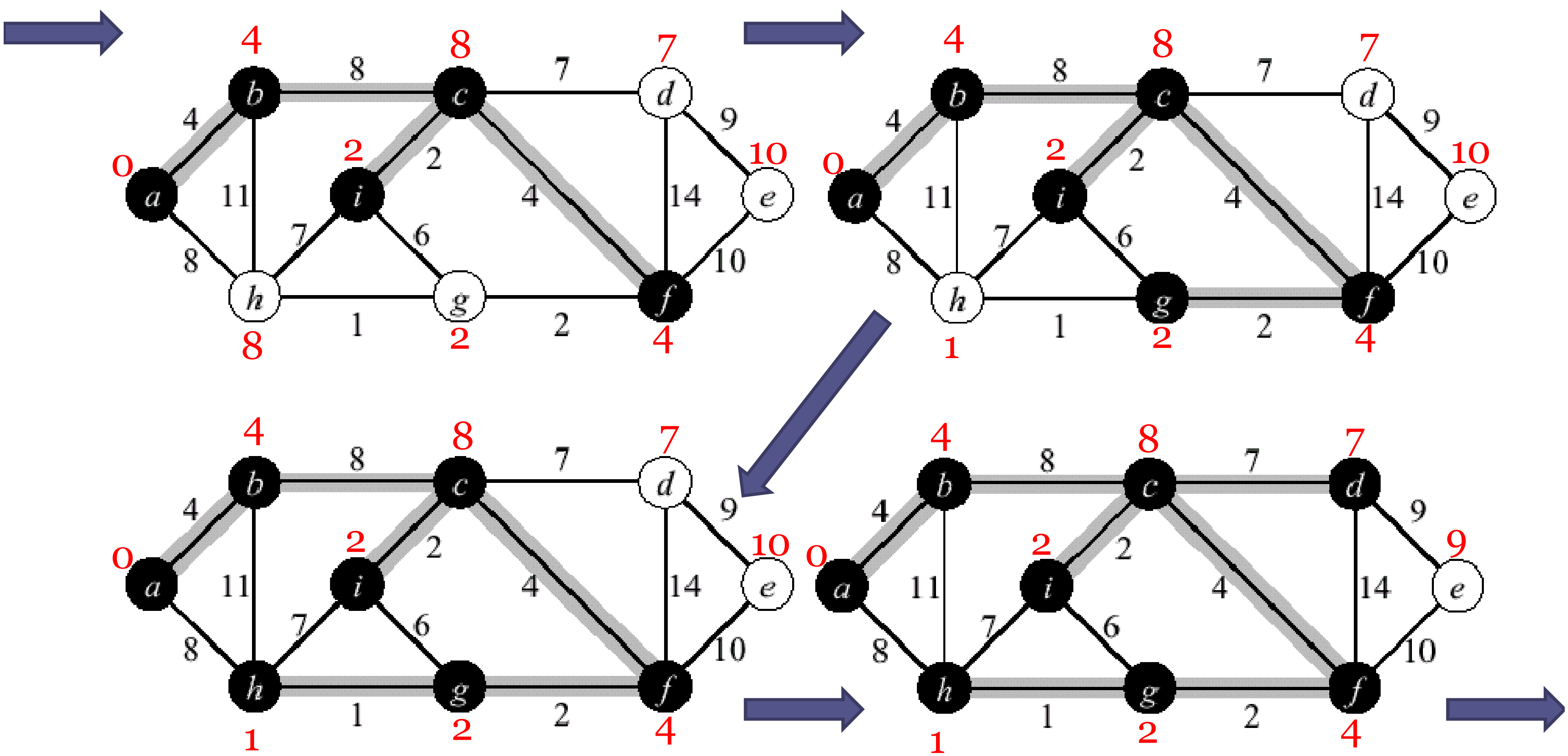
Prim's Algorithm

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

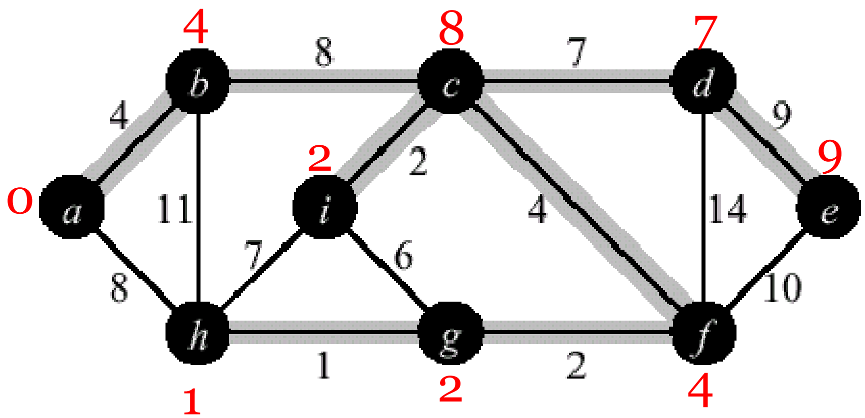
Prim’s Algorithm: example



Prim's Algorithm: example…



Prim's Algorithm: example…



SAMSUNG

DTU Delhi Technological University