ECE 09468/09568
Discrete Event Systems

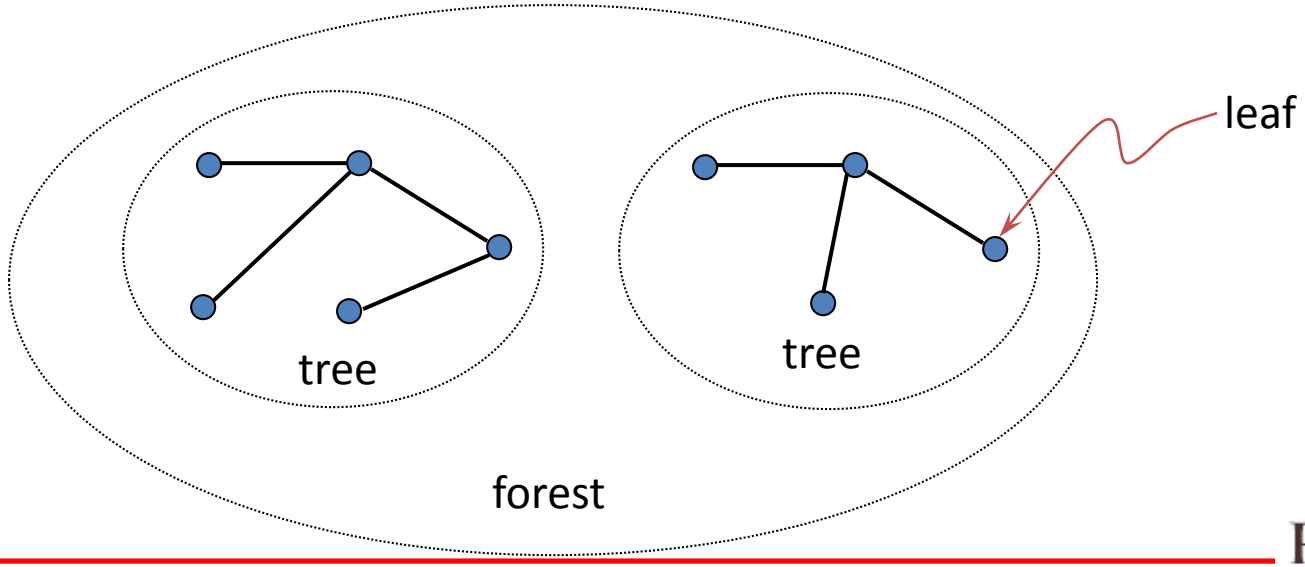
Lecture 3: **Tree –A Special Graph**

Dr. Ying (Gina) Tang
Department of Electrical and Computer Engineering
Rowan University

Definition of a Tree

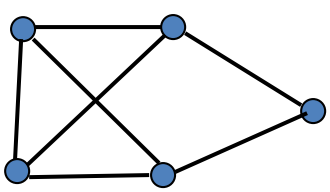
What is a tree?

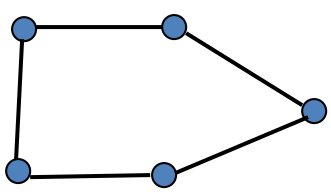
- A tree is an acyclic connected graph in which any two vertices are connected by a simple path.
- A leaf is a vertex of degree 1
- The forest is an acyclic graph (i.e., a disjoint union of trees



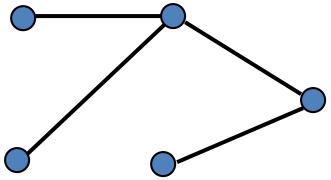
Special Trees

- A spanning subgraph of G is a subgraph that has the same vertex set as G.
- A spanning tree is a spanning subgraph that is a tree





Spanning subgraph

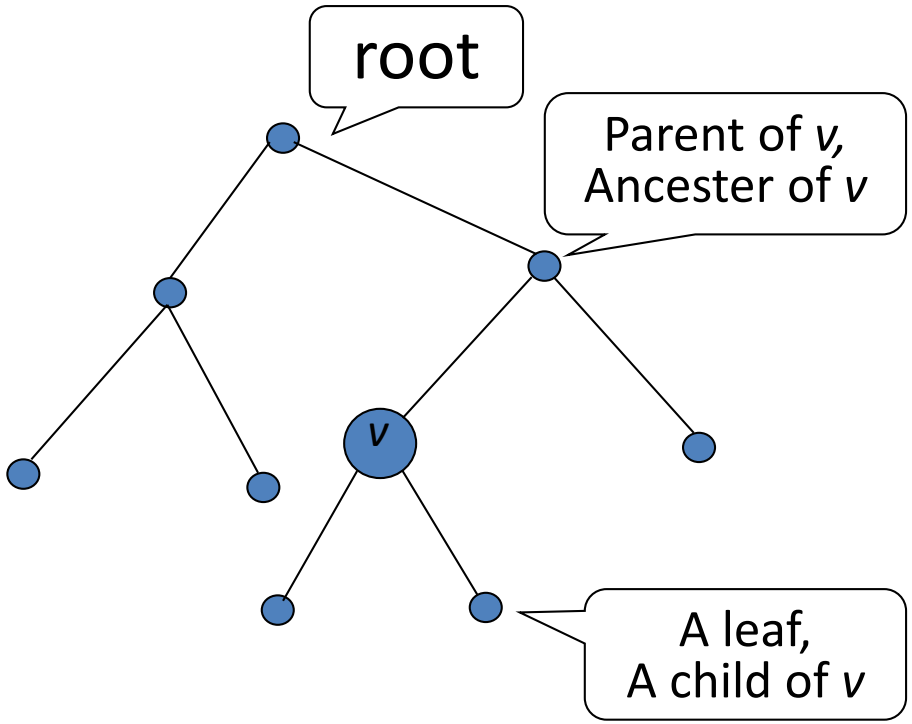


Spanning tree

- Two methods to systematically create a spanning tree
  - Cutting-down: choosing any cycle in G, remove one of the cycle's edges, and repeat the process until no cycles are left
  - Building-up: select edges of G one at a time in such a way that no cycles are created, and repeat it until all nodes are included

Special Trees

- A rooted tree is a tree with one node being chosen as a root, in which case the edges have a natural orientation away from the root
- The parent of a vertex in a rooted tree is the vertex connected to it on the path to the root.
- A child of a vertex *v* in a rooted tree is a vertex of which *v* is the parent
- A leaf is a vertex without children

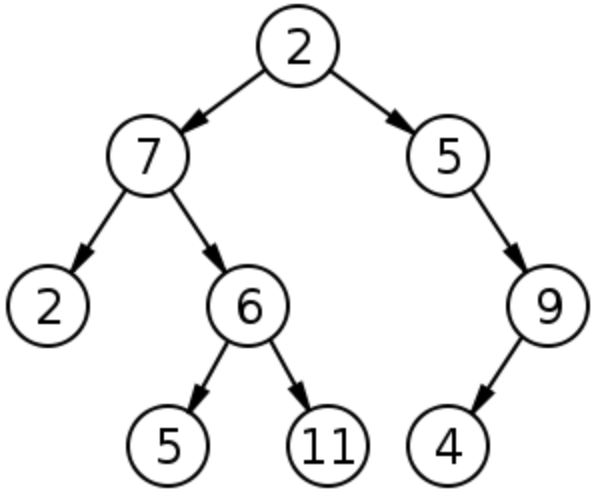


Special Trees

- For a rooted tree:
  - Depth of a node: the length of the path from the root to the node
  - Height of a tree: the length of the path from the root to the deepest node in the tree
  - Siblings: nodes that share the same parent
  - Ancestor of a node v_i: a node v_j that exists on the path from v_i to the root. The node v_i is then a descendant of v_j
  - Size of a node: the number of descendant the node has including itself

Special Trees

- A **binary tree** is a tree where each vertex has at most two children, and each child of a vertex is designated as its **left child** or **right child**
- The subtrees rooted at the children of the root are the **left subtree** and the **right subtree** of the tree



Minimal Spanning Tree Problem

Input: Given a weighted graph G =(V_G, E_G) and a starting vertex *v*. The weight of each edge (*i, j*), e_{ij}, is w_{ij} *and* $w_{ij}=\infty$ if there is no edge connecting vertices *i* and *j*

Idea: Maintain a tree T including *v* at the beginning and the total weight of all the edges in T is minimized

Applications: phone network design – a company with several branches at different locations would like to lease phone lines to connect them up with the minimal lease fees.

Minimal Spanning Tree – Prim's Algorithm

Initialization: Set T= (V_T, E_T), where V_T={v} and E_T=∅.

Iteration:

- Select a vertex ***u*** outside V_T such that

  $w_{xu}=\min_{x \in T,\, u \notin T} w_{xu}$

- Add u to V_T and add (x, u) to E_T

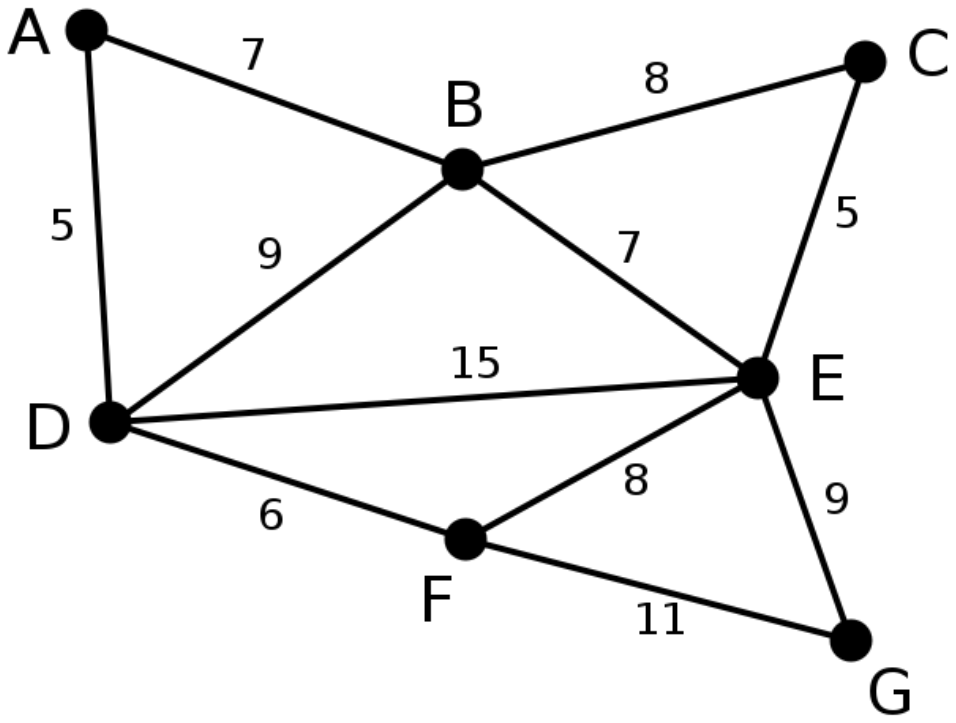
- The iteration continues until $V_T=V_G$.

Minimal Spanning Tree – Kruskal's Algorithm

- Find the cheapest edge in G (if there is more than one, pick one at random). Mark it with a special color
- Find the cheapest unmarked edge in G that does not close a colored circuit. Mark this edge.
- Repeat Step 2 until every node of G is reached out

Minimal Spanning Tree Example:

- Try **Prim's** **and** **Kruskal's** algorithms individually

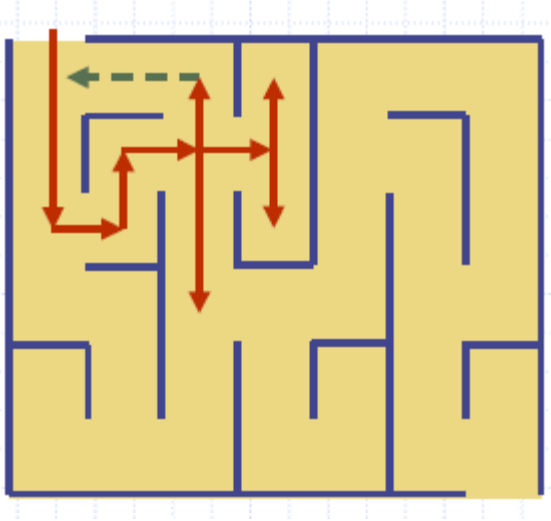


Tree/Graph Traversal –Depth First Search (DFS)

Input: An unweighted graph (or digraph) and a starting vertex *u*.

Idea: Maintain a set *R* of edges that have been discovered but a set *S* of edges that are back edges

Applications: Find solutions to a maze; Find strong components of a given graph…



Tree/Graph Traversal –Depth First Search:

- Initialize all vertices in G

```
for v := 1 to n do
    mark[v] := unvisited;
for v := 1 to n do
    if mark[v] = unvisited then
        dfs(v)
```

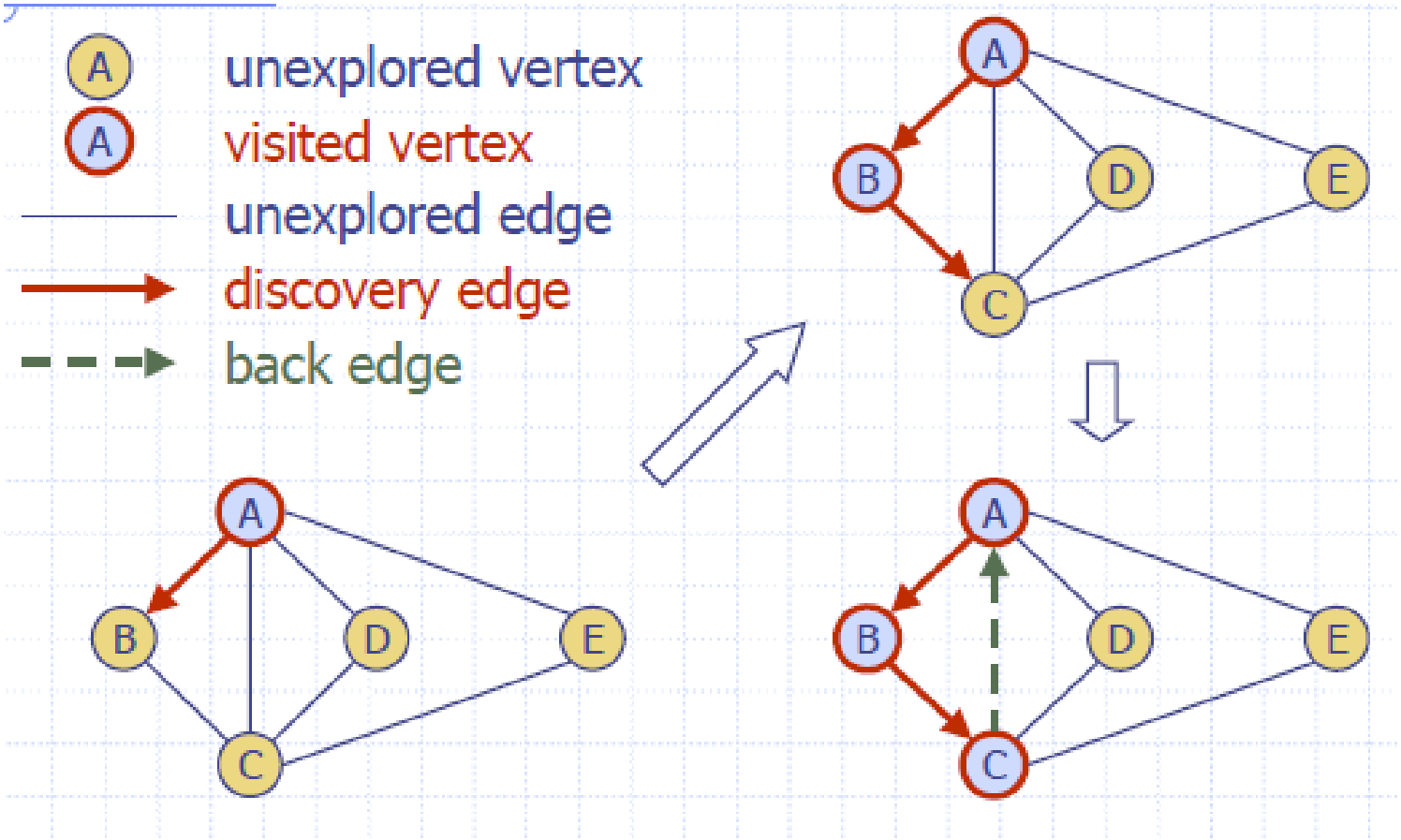
- Run DFS

```
        procedure dfs ( v: vertex );
            var
                w: vertex;
            begin
(1)             mark[v] := visited;
(2)             for each vertex w on L[v] do
(3)                 if mark[w] = unvisited then
(4)                     dfs(w)
            end; { dfs }
```

Note that : $L(v)$ is an adjacency set of vertex v.

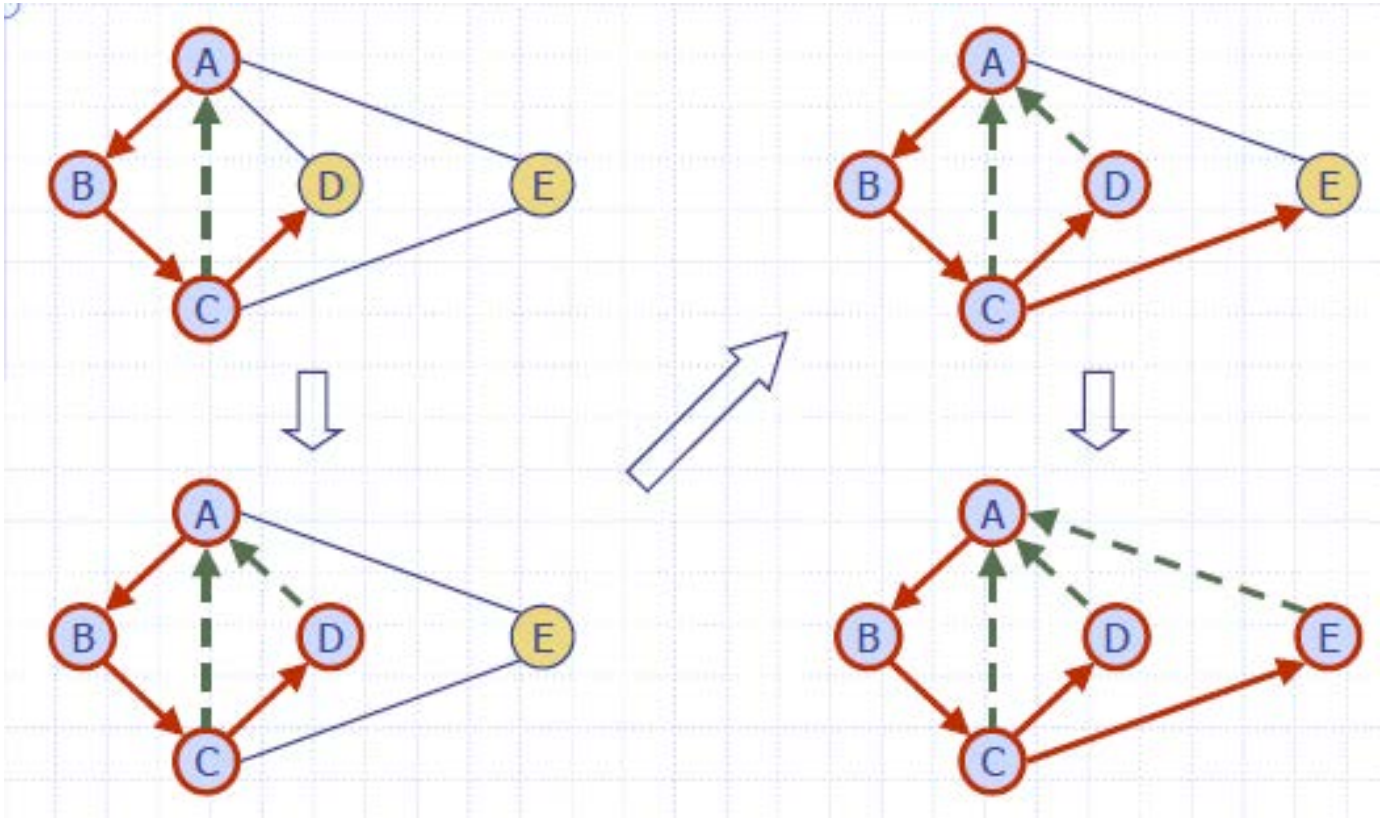
Algorithms for Trees and Graphs

Tree/Graph Traversal –Depth First Search:



Algorithms for Trees and Graphs

Tree/Graph Traversal –Depth First Search:



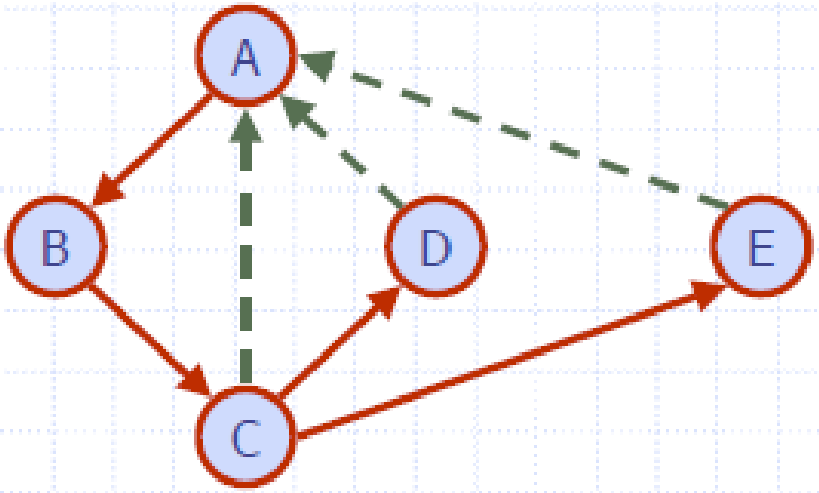
Properties of DFS:

Property 1

$DFS(G, v)$ visits all the vertices and edges in the connected component of v

Property 2

The discovery edges labeled by $DFS(G, v)$ form a spanning tree of the connected component of v

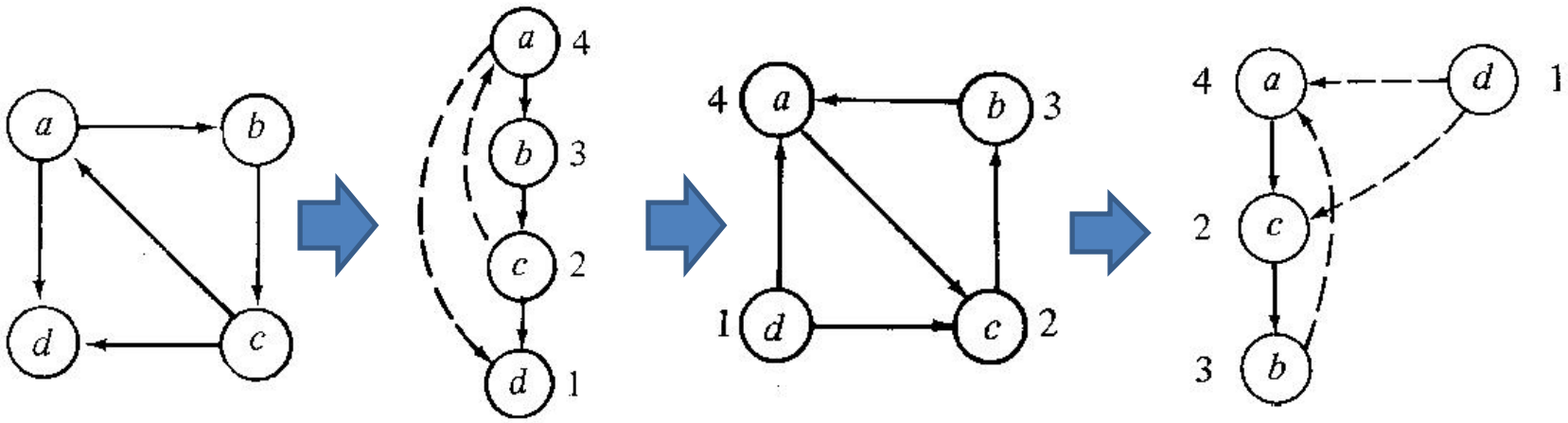


Applications of DFS:

- Find strong components of a given digraph G
  - Perform a DFS on G and number the vertices in order of completion of the recursive calls (i.e., assign a number to vertex v after line (4) of dfs procedure)
  - Construct a new directed graph G' by reversing the direction of every edge in G
  - Perform DFS on G' , starting the search from the highest-numbered vertex according to the numbering assigned at Step 1. If the DFS does not reach all vertices, start the next DFS from the highest-numbered remaining vertex
  - Each tree in the resulting spanning forest is a strong component of G

Applications of DFS:

- Find strong components of a given digraph G



Original graph G | DFS on G | Graph G' | DFS spanning forest

Tree Algorithms

Applications of DFS:

- Find a path of a given two vertices v, u in the graph G
- Find a cycle of a given two vertices v, u in the graph G
- …

Tree/Graph Traversal –Breadth First Search (BFS)

Input: An unweighted graph (or digraph) and a starting vertex *s*.

Idea: Maintain a set *R* of vertices that have been reached but not searched and a set *S* of vertices that have been searched. The set *R* is maintained as a First-In First-Out list (queue), so the first vertices found are the first vertices explored

Tree/Graph Traversal –BFS

Initialization:

$R=\{s\}$, $S=\emptyset$, $d(s)=0$, and $d(v)=\infty$, for $\forall v \in V$ and $v \neq s$

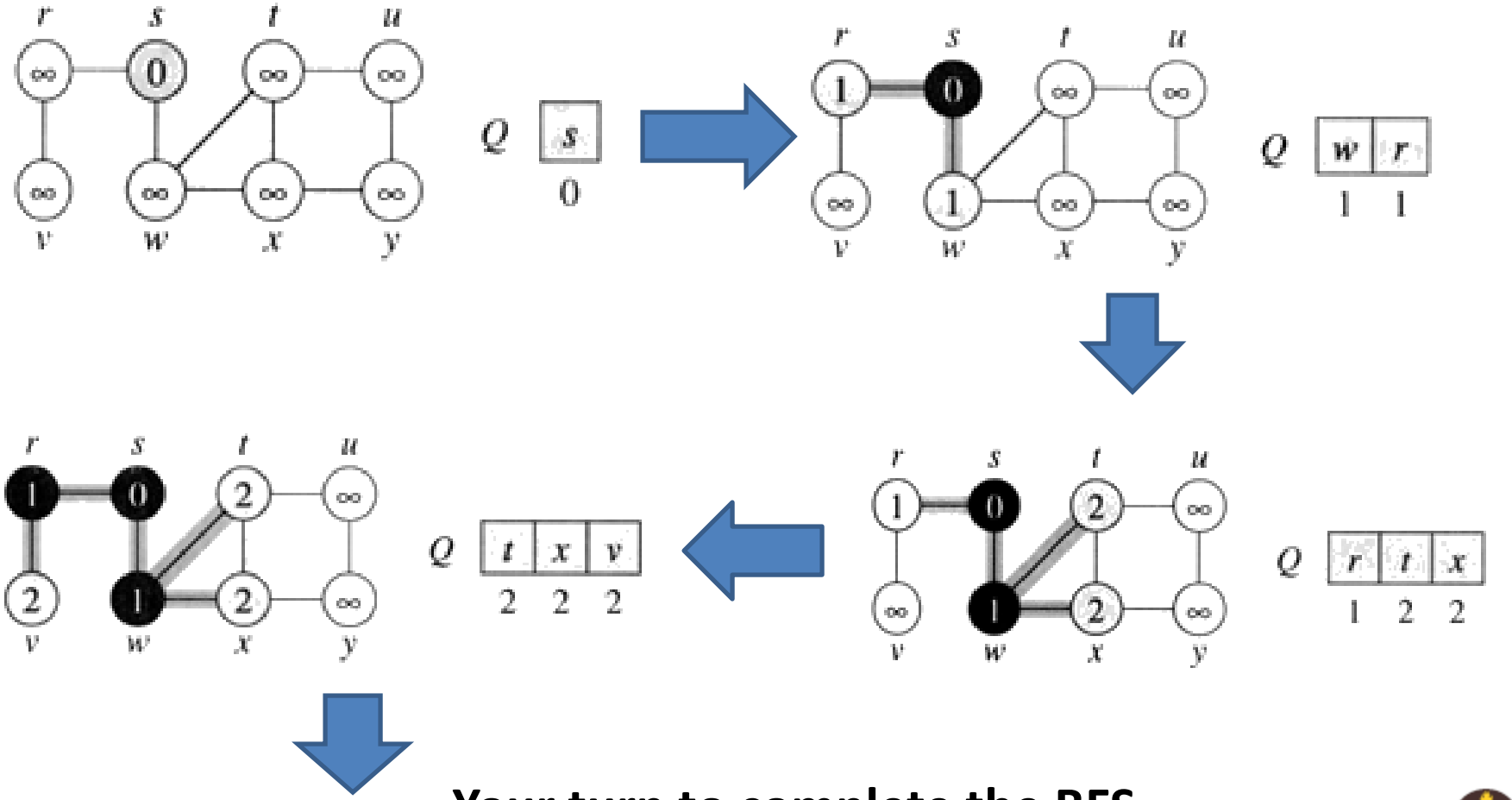
Iteration:

1. As long as $R \neq \emptyset$, we search from the first vertex u of R
2. The neighbors of u not in $S \cup R$ are added to the back of R and assigned distance $d(u)+1$
3. Then u is removed from the front of R and placed in S

d(v) contains the number of tree edges on the path from s to v

Algorithms for Trees and Graphs

Tree/Graph Traversal –BFS



Your turn to complete the BFS…

Rowan University