6. Graphs

© 2008, University of Colombo School of Computing

6.1. Definition of different Graphs

- A **graph** is the basic object of study in graph theory. Informally speaking, a graph is a set of objects called *points*, *nodes*, or *vertices* connected by links called *lines* or *edges*. In a proper graph, which is by default *undirected*, a line from point *A* to point *B* is considered to be the same thing as a line from point *B* to point *A*. In a *digraph*, short for *directed graph*, the two directions are counted as being distinct *arcs* or *directed edges*. Typically, a graph is depicted in diagrammatic form as a set of dots (for the points, vertices, or nodes), joined by curves (for the lines or edges).

6.1. Definition of different Graphs contd…

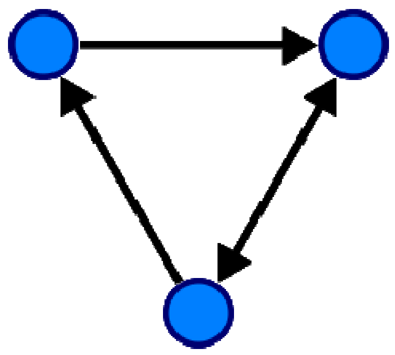
- A **graph** or **undirected graph** *G* is an ordered pair *G*: = (*V*,*E*) that is subject to the following conditions:
  - *V* is a set, whose elements are called **vertices** or **nodes**,
  - *E* is a multiset of unordered pairs of vertices (not necessarily distinct), called **edges** or **lines**.

(Note that this defines the most general type of graph. Some authors call this a multigraph and reserve the term "graph" for simple graphs.)

6.1. Definition of different Graphs contd…

- The vertices belonging to an edge are called the **ends**, **endpoints**, or **end vertices** of the edge.
- *V* (and hence *E*) are usually taken to be finite, and many of the well-known results are not true (or are rather different) for **infinite graphs** because many of the arguments fail in the infinite case. The **order** of a graph is | *V* | (the number of vertices). A graph's **size** is | *E* | , the number of edges. The **degree** of a vertex is the number of edges that connect to it, where an edge that connects to the vertex at both ends (a loop) is counted twice.
- The edges *E* induce a symmetric binary relation ~ on *V* which is called the **adjacency** relation of *G*. Specifically, for each edge {*u*,*v*} the vertices *u* and *v* are said to be **adjacent** to one another, which is denoted *u* ~ *v*.
- For an edge {*u*, *v*}, graph theorists usually use the somewhat shorter notation *uv*.

6.1. Definition of different Graphs contd…



- **Types of graphs**
  - **Directed graph**
    - A **directed graph** or **digraph** *G* is an ordered pair *G*: = (*V*,*A*) with
      - *V* is a set, whose elements are called **vertices** or **nodes**,
      - *A* is a set of ordered pairs of vertices, called **directed edges**, **arcs**, or **arrows**.
    - An arc *e* = (*x*,*y*) is considered to be directed **from** *x* **to** *y*; *y* is called the **head** and *x* is called the **tail** of the arc; *y* is said to be a **direct successor** of *x*, and *x* is said to be a **direct predecessor** of *y*. If a path leads from *x* to *y*, then *y* is said to be a **successor** of *x*, and *x* is said to be a **predecessor** of *y*. The arc (*y*,*x*) is called the arc (*x*,*y*) **inverted**.

6.1. Definition of different Graphs contd…

– **Directed graph contd..**
  - A directed graph *G* is called symmetric if, for every arc that belongs to *G*, the corresponding inverted arc also belongs to *G*. A symmetric loopless directed graph is equivalent to an undirected graph with the pairs of inverted arcs replaced with edges; thus the number of edges is equal to the number of arcs halved.
  - A variation on this definition is the oriented graph, which is a graph (or multigraph; see below) with an orientation or direction assigned to each of its edges. A distinction between a directed graph and an oriented *simple* graph is that if *x* and *y* are vertices, a directed graph allows both (*x*,*y*) and (*y*,*x*) as edges, while only one is permitted in an oriented graph. A more fundamental difference is that, in a directed graph (or multigraph), the directions are fixed, but in an oriented graph (or multigraph), only the underlying graph is fixed, while the orientation may vary.

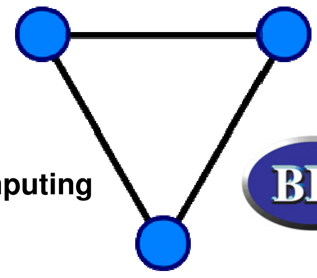
6.1. Definition of different Graphs contd…

- **Types of graphs**
  - **Undirected graph**

    A graph $G = \{V,E\}$ in which every edge is undirected. This is the same as a digraph (look above) where for an edge (v,u) there is an edge from v to u and u to v.
  - **Finite graph**

    A finite graph is a graph $G = <V,E>$ such that $V(G)$ and $E(G)$ are finite sets.
  - **Simple graph**

    A simple graph is an undirected graph that has no self-loops and no more than one edge between any two different vertices. In a simple graph the edges of the graph form a set (rather than a multiset) and each edge is a pair of *distinct* vertices. In a simple graph with p vertices every vertex has a degree that is less than p.



6.1. Definition of different Graphs contd…

- **Types of graphs**
  - **Regular graph**

    A regular graph is a graph where each vertex has the same number of neighbors, i.e., every vertex has the same degree or valency. A regular graph with vertices of degree *k* is called a *k*-regular graph or regular graph of degree *k*.
  - **Weighted graph**

    A graph is a weighted graph if a number (weight) is assigned to each edge. Such weights might represent, for example, costs, lengths or capacities, etc. depending on the problem.

    Weight of the graph is sum of the weights given to all edges.

6.1. Definition of different Graphs contd…

- **Types of graphs**
  - **Mixed graph**

    A **mixed graph** *G* is a graph in which some edges may be directed and some may be undirected. It is written as an ordered triple *G* := (*V, E, A*) with *V*, *E*, and *A* defined as above. Directed and undirected graphs are special cases.
  - **Complete graph**

    Complete graphs have the feature that each pair of vertices has an edge connecting them.
  - **Loop**

    A loop is an edge (directed or undirected) which starts and ends on the same vertex; these may be permitted or not permitted according to the application. In this context, an edge with two different ends is called a **link**.

6.1. Definition of different Graphs contd…

- **Types of graphs**
  - **Multi graph**

    The term "multigraph" is generally understood to mean that multiple edges (and sometimes loops) are allowed. Where graphs are defined so as to *allow* loops and multiple edges, a multigraph is often defined to mean a graph *without* loops,however, where graphs are defined so as to *disallow* loops and multiple edges, the term is often defined to mean a "graph" which can have both multiple edges *and* loops, although many use the term "pseudograph" for this meaning.
  - **Half-edges, loose edges**

    In exceptional situations it is even necessary to have edges with only one end, called **half-edges**, or no ends (**loose edges**).

6.2. Graph Representation

- Two common ways to represent graphs on a computer are as an adjacency list or as an adjacency matrix.
  - **Adjacency list:**
    Vertices are labelled (or re-labelled) from 0 to |V(G)|-1. Corresponding to each vertex is a list (either an array or linked list) of its neighbours.
  - **Adjacency matrix:**
    Vertices are labelled (or re-labelled) with integers from 0 to |V(G)|-1. A two-dimensional boolean array A with dimensions |V(G)| x |V(G)| contains a 1 at A[i][j]
  - if there is an edge from the vertex labelled i to the vertex labelled j,and a 0 otherwise.

Both representations allow us to represent directed graphs, since we can have an edge from v_i to v_j , but lack one from v_i to v_j . To represent undirected graphs, we simply make sure that are edges are listed twice: once from v_i to v_j , and once from v_i to v_j .

© 2008, University of Colombo School of Computing

6.3. Graph Traversals contd…

Breadth first search

- Given a graph G=(V,E) and a *source vertex s*, BFS explores the edges of G to "discover" (visit) each node of G reachable from *s*.
- Idea - expand a *frontier* one step at a time.
- *Frontier* is a FIFO queue (O(1) time to update)

6.3. Graph Traversals contd…

Breadth first search

- Computes the *shortest distance* (*dist*) from *s* to any reachable node.
- Computes a *breadth first tree* (of *parents*) with root *s* that contains all the reachable vertices from *s*.
- To get O(|V|+|E|) we use an adjacency list representation. If we used an adjacency matrix it would be $O(|V|^2)$

6.3. Graph Traversals contd…

Coloring the nodes

- We use colors (***white***, ***gray*** and ***black***) to denote the state of the node during the search.
- A node is ***white*** if it has not been reached (discovered).
- *Discovered* nodes are *gray* or *black*. ***Gray*** nodes are at the frontier of the search. ***Black*** nodes are fully explored nodes.

6.3. Graph Traversals contd…

BFS - initialize

```
procedure BFS(G:graph; s:node; var
  color:carray; dist:iarray; parent:parray);
  for each vertex u do
    color[u]:=white; dist[u]:=∞;          Θ(V)
    parent[u]:=nil; end for
  color[s]:=gray; dist[s]:=0;
  init(Q); enqueue(Q, s);
```

© 2008, University of Colombo School of Computing

6.3. Graph Traversals contd…

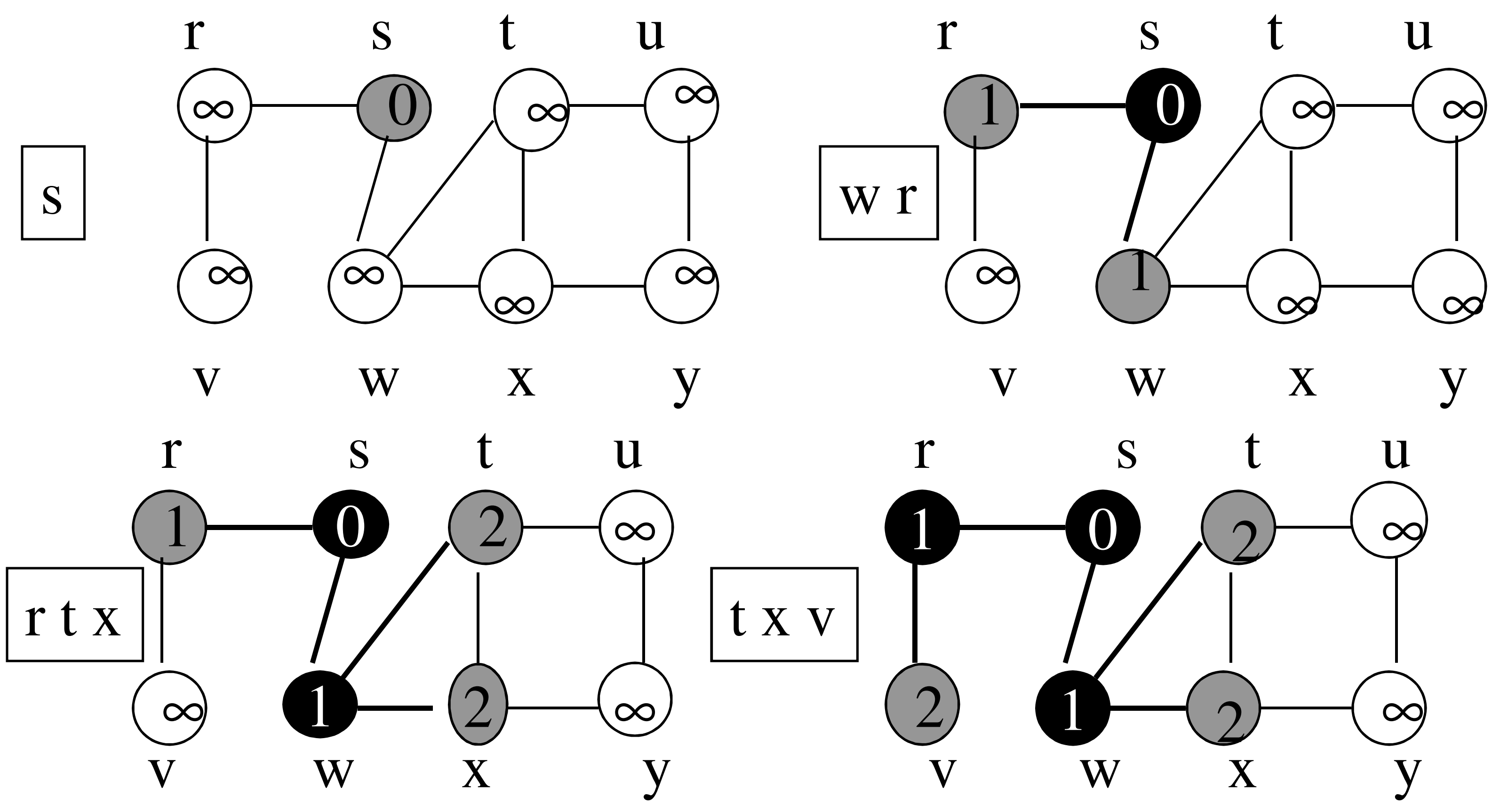
BFS - main

```
while not (empty(Q)) do
  u:=head(Q);
  for each v in adj[u] do
    if color[v]=white then                    O(E)
      color[v]:=gray; dist[v]:=dist[u]+1;
      parent[v]:=u; enqueue(Q, v);
  dequeue(Q); color[u]:=black;
end BFS
```

$$\sum_{u \in V} |ADJ[u]| = \sum_{u \in V} \mathrm{degree}[u] = O(E)$$

6.3. Graph Traversals contd…

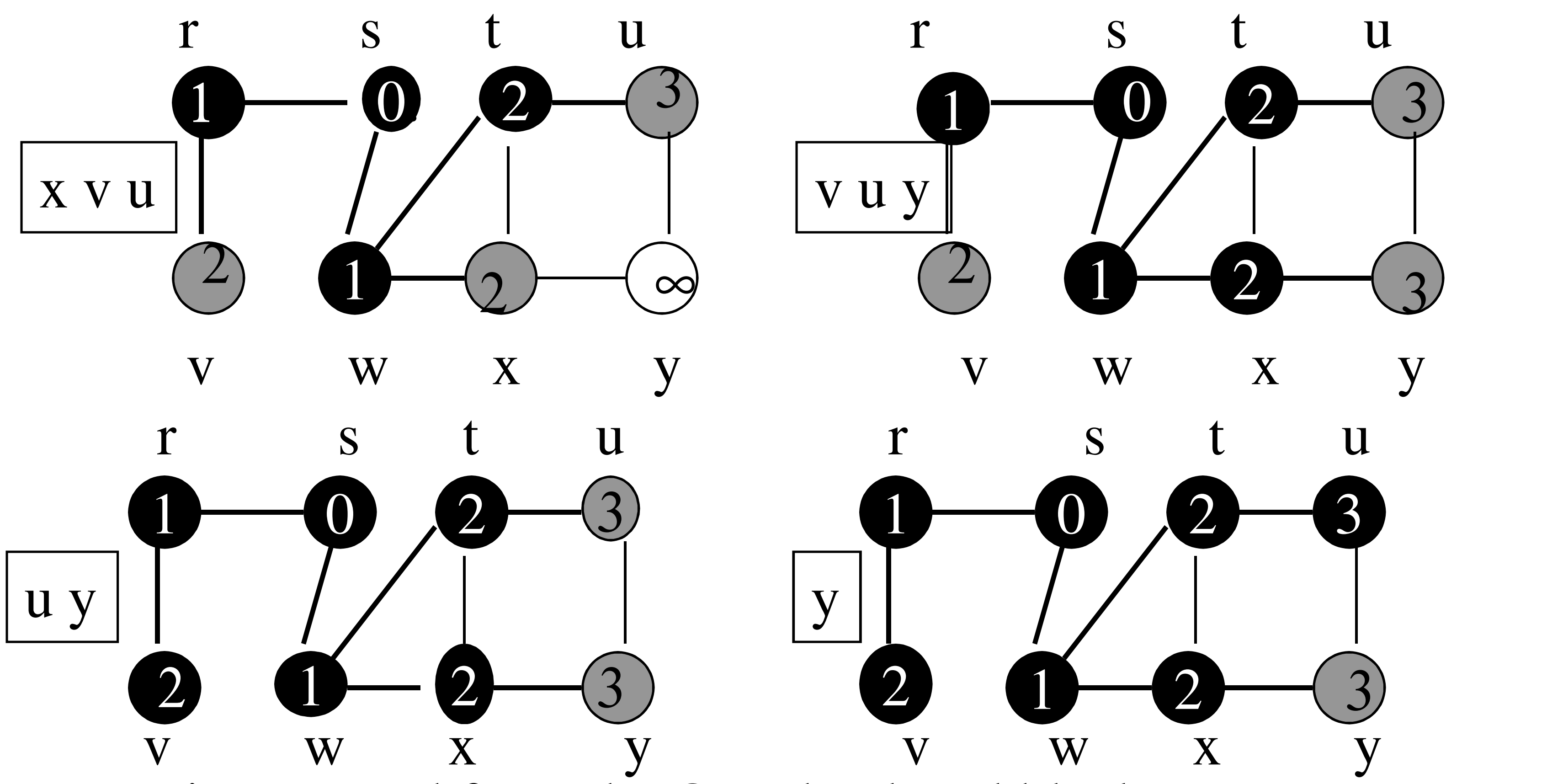
BFS example



© 2008, University of Colombo School of Computing

6.3. Graph Traversals contd…

BFS example



now y is removed from the Q and colored black

6.3. Graph Traversals contd…

Analysis of BFS

- Initialization is Θ(|V|).
- Each node can be added to the queue at most once (it needs to be white), and its adjacency list is searched only once. At most all adjacency lists are searched.
- If graph is undirected each edge is reached twice, so loop repeated at most 2|E| times.
- If graph is directed each edge is reached exactly once. So the loop repeated at most |E| times.
- Worst case time O(|V|+|E|)

6.3. Graph Traversals contd…

Depth First Search

- Goal - explore every vertex and edge of G
- We go "deeper" whenever possible.
- *Directed* or *undirected* graph $G = (V, E)$.
- To get worst case time $\Theta(|V|+|E|)$ we use an adjacency list representation. If we used an adjacency matrix it would be $\Theta(|V|^2)$

6.3. Graph Traversals contd…

Depth First Search

- Until there are no more undiscovered nodes.
  - Picks an undiscovered node and starts a depth first search from it.
  - The search proceeds from the *most recently discovered* node to discover new nodes.
  - When the last discovered node *v* is fully explored, backtracks to the node used to discover *v*. Eventually, the start node is fully explored.

© 2008, University of Colombo School of Computing

6.3. Graph Traversals contd…

Depth First Search

- In this version *all* nodes are discovered even if the graph is directed, or undirected and not connected
- The algorithm saves:
  - A depth first *forest* of the edges used to discover new nodes.
  - Timestamps for the first time a node *u* is discovered *d*[*u*] and the time when the node is fully explored *f*[*u*]

6.3. Graph Traversals contd…

Depth First Search

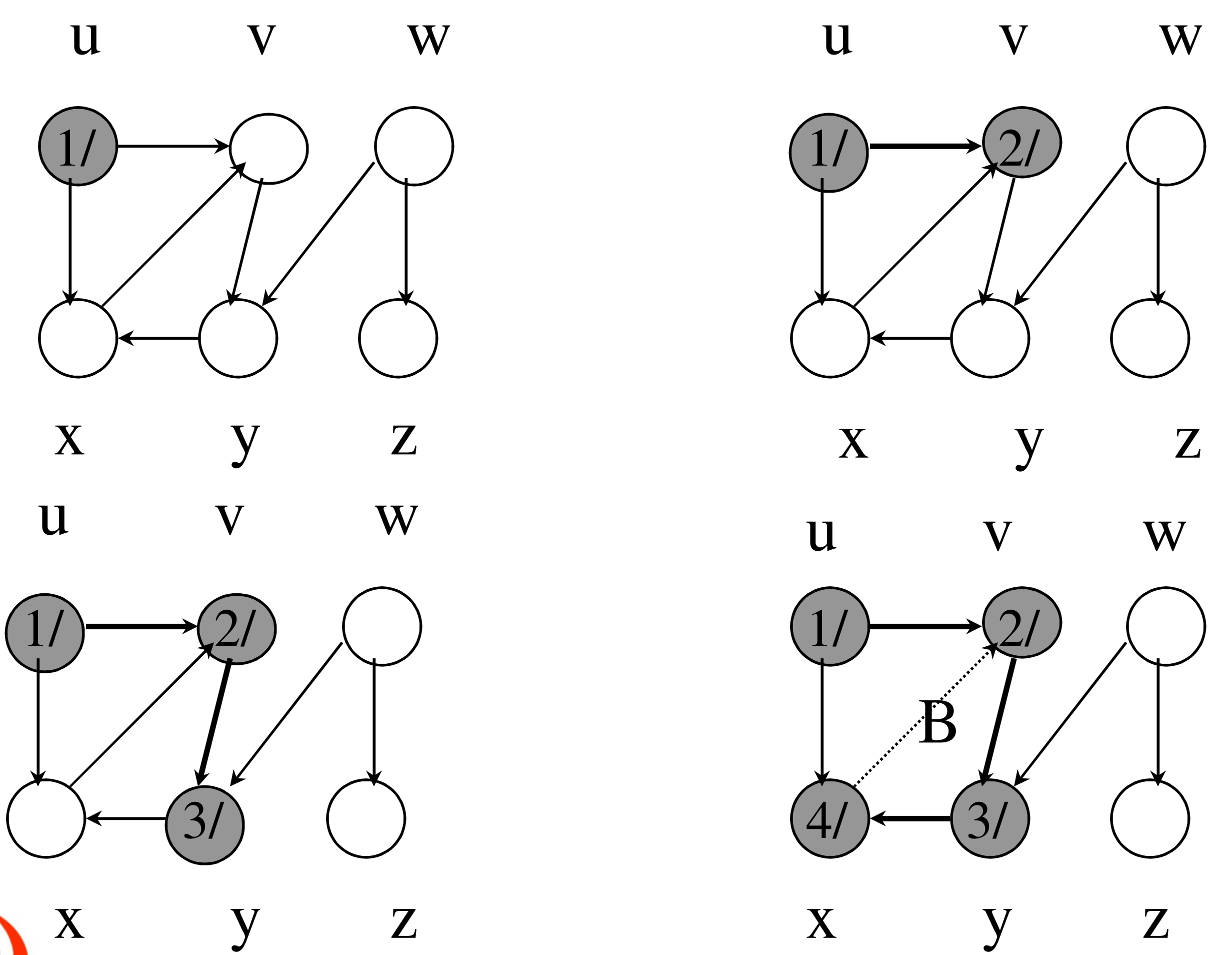
```
procedure DFS(G:graph; var color:carray; d, f:iarray;
    parent:parray);
    for each vertex u do
        color[u]:=white;  parent[u]:=nil;               $\Theta(V)$
    end for
    time:=0;
    for each vertex u do
        if color[u]=white then
                DFS-Visit(u); end if; end for
end DFS
```

6.3. Graph Traversals contd…

```
DFS-Visit(u)
    color[u]=:gray; time:=time+1; d[u]:=time
    for each v in adj[u] do
        if color[v]=white then
            parent[v]:=u;  DFS-Visit(v);
        end if; end for;
    color[u]:=black; time:=time+1; f[u]:=time;
    end DFS-Visit
```

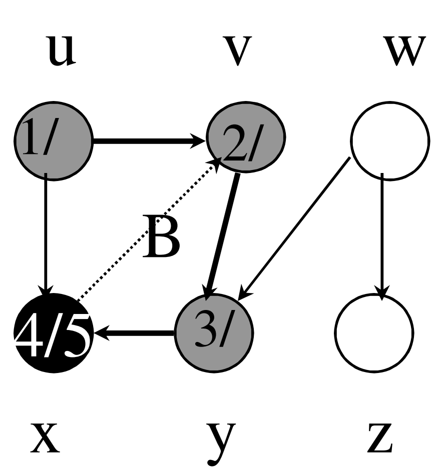
6.3. Graph Traversals contd…

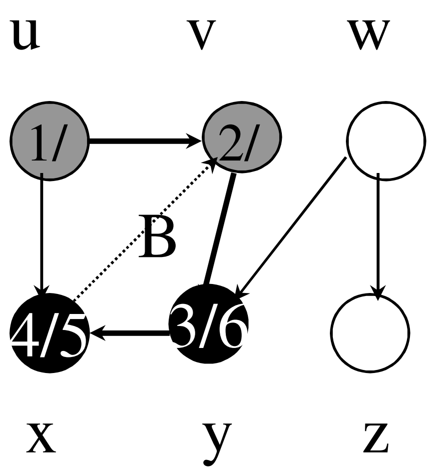
DFS example (1)

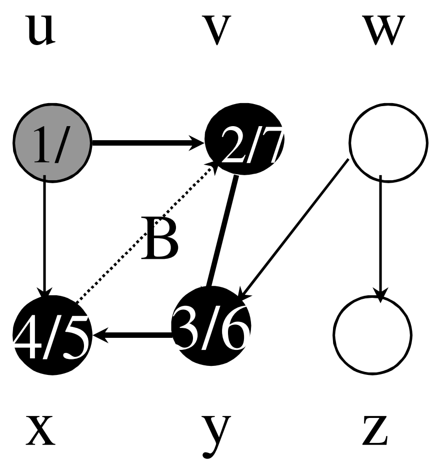


6.3. Graph Traversals contd…

DFS example (2)

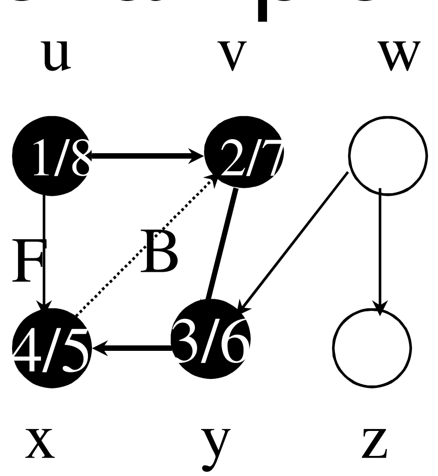


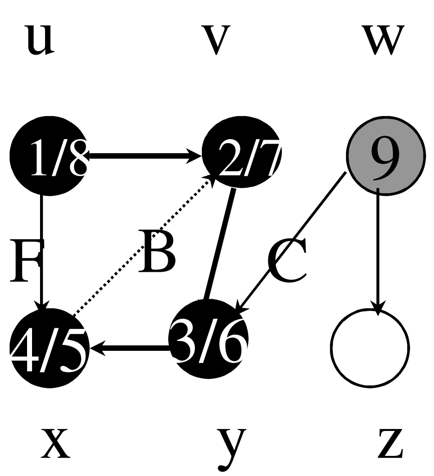


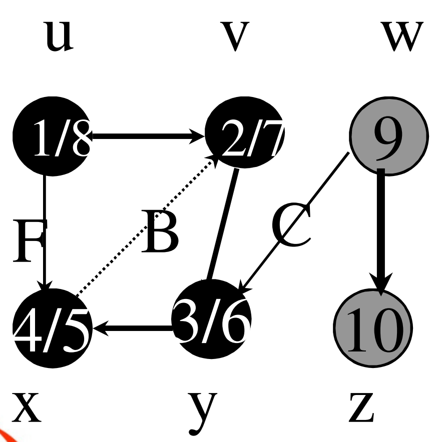


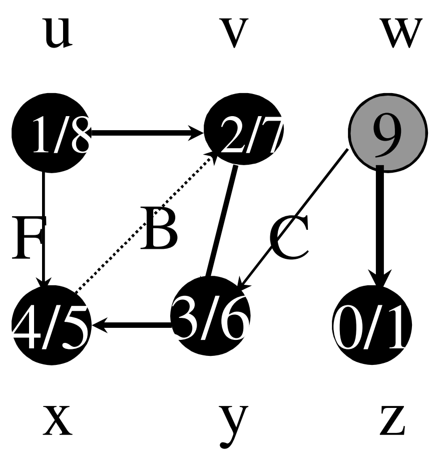
6.3. Graph Traversals contd…

DFS example (3)



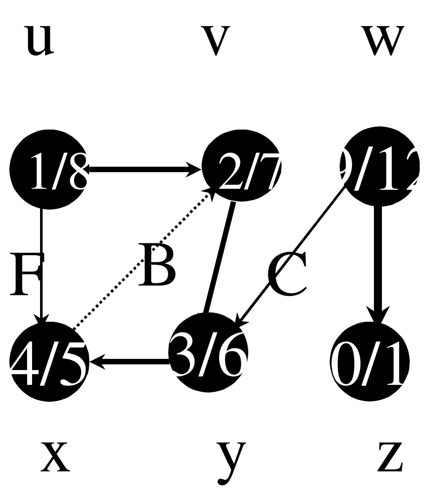






6.3. Graph Traversals contd…

DFS example (4)



© 2008, University of Colombo School of Computing

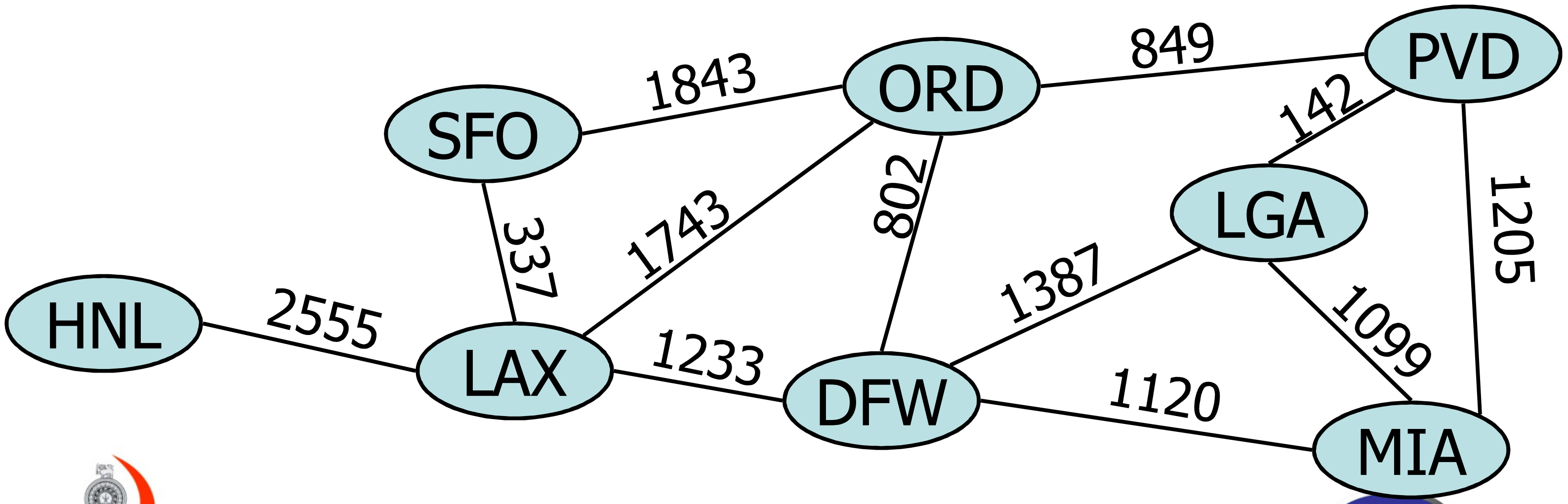
6.3. Graph Traversals contd…

Analysis

- DFS is $\Theta(|V|)$ (excluding the time taken by the DFS-Visits).
- DFS-Visit is called once for each node v. Its *for* loop is executed |adj(v)| times. The DFS-Visit calls for all the nodes take $\Theta(|E|)$.
- Worst case time $\Theta(|V|+|E|)$

6.4. Shortest Paths

- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



6.4 Shortest Paths contd…

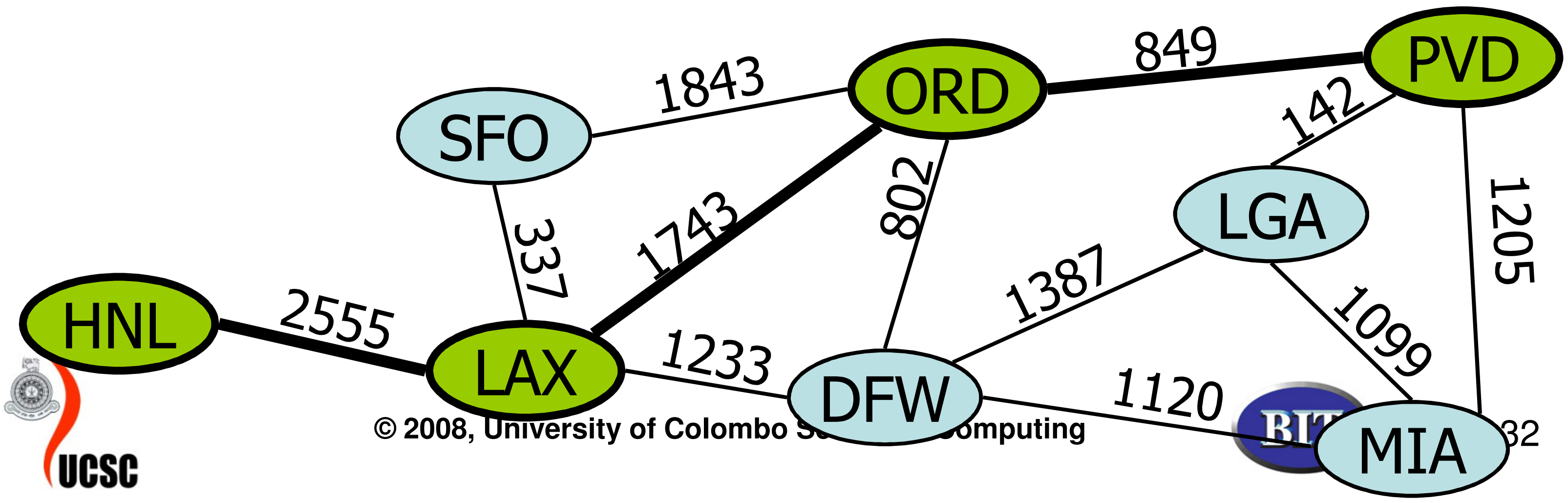
- The weight of path p = < v_0, v_1,….v_k> is the sum of the weights of its constituent edges.

- Given a weighted graph and two vertices ***u*** and ***v***, we want to find a path of minimum total weight between ***u*** and ***v.***
  - Length of a path is the sum of the weights of its edges.

6.4 Shortest Paths contd…

Example: Shortest path between *Providence* and *Honolulu*

- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



6.4 Shortest Paths contd…

- We will focus on **Single source shortest paths problem**: given a graph G = (V,E), we want to find a shortest path from a given source vertex s Є V to each vertex v Є V.

Shortest Path Properties

Property 1: A sub path of a shortest path is itself a shortest path.

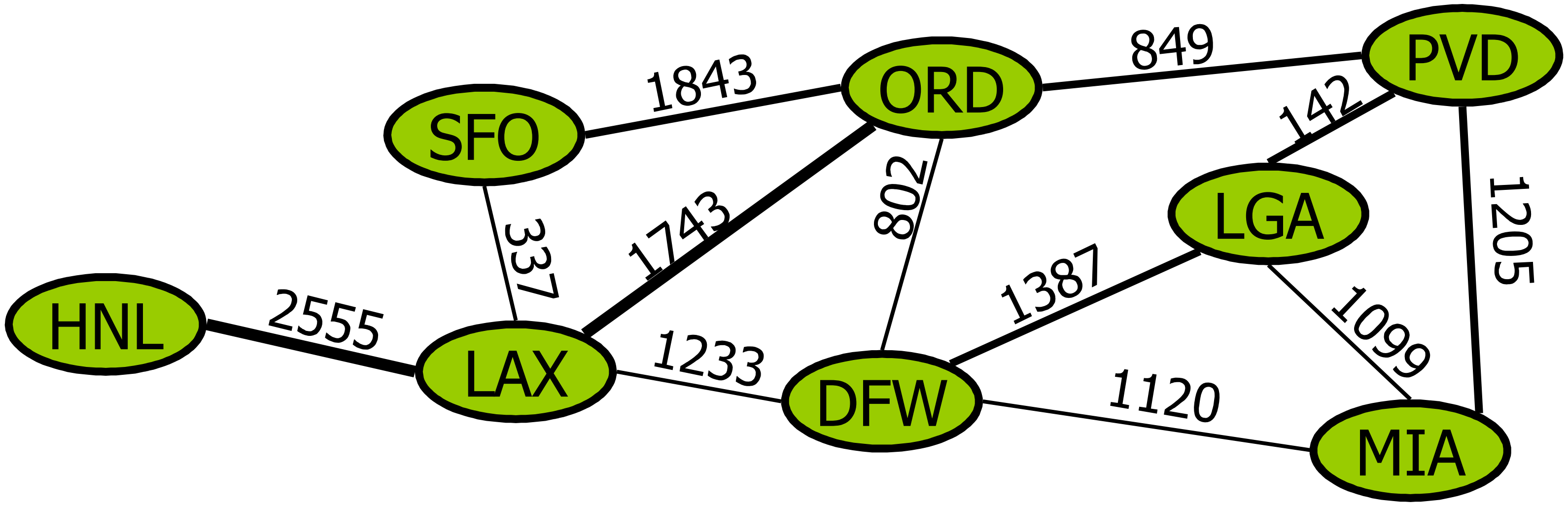
Property 2: There is a tree of shortest paths from a start vertex to all the other vertices.

© 2008, University of Colombo School of Computing

6.4.1. Shortest Path Problem

Example:

Tree of shortest paths from *Providence*



6.4.1. Shortest Path Problem contd…

- The shortest path algorithms use the technique of **relaxation**.

- For each vertex v Є V, an attribute d[v] is maintained which is an upper bound on the weight of a shortest path from source s to v.
- d[v] – shortest path estimate

6.4.2. Shortest Path Algorithms

- The shortest path estimates and predecessors are initialized by the following O(V) time procedure.

```
INITIALIZE-SINGLE-SOURCE (G,s)
    for each vertex v Є V[G]
        do d[v] ← ∞
           π[v] ← NIL
    d[s] ← 0
```

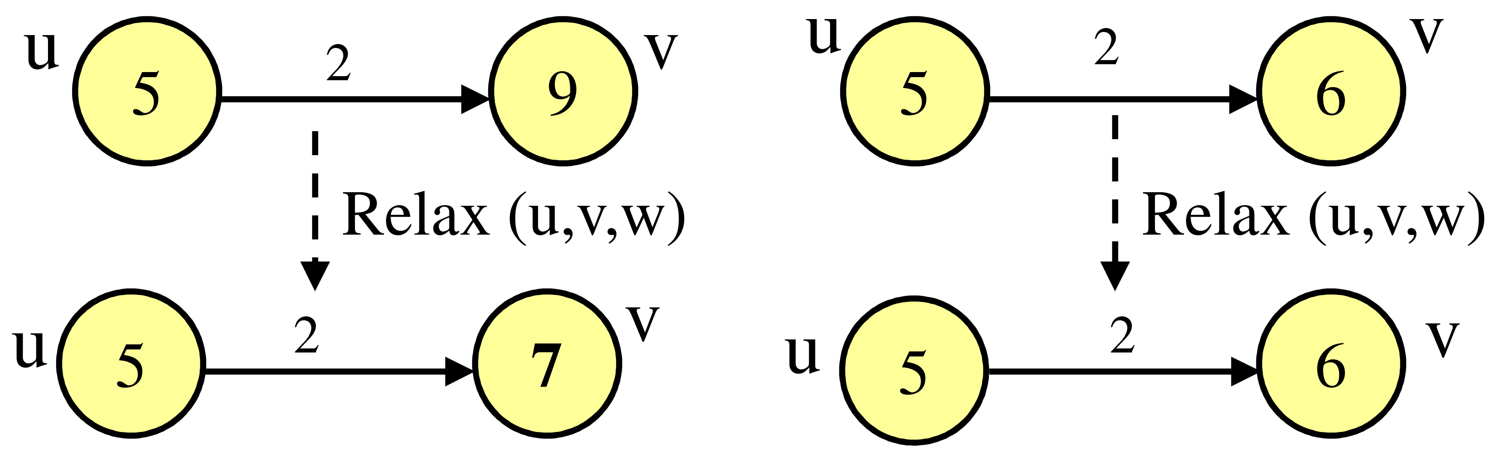
6.4.2. Shortest Path Algorithms contd…

- Relaxing and edge (u,v) – consists of testing whether the shortest path to v found so far can be improved by going through u. If so d[v] and π[v] values should be updated.

- A relaxation step may decrease the value of the shortest path estimate d[v].

6.4.3. Relaxation

- Relaxation of an edge (u,v) with weight w(u,v) = 2.



$d[v] > d[u] + w(u,v)$

d[v] is changed by relaxation

$d[v] \leq d[u] + w(u,v)$

d[v] is unchanged by relaxation

6.4.3. Relaxation contd…

```
Relax (u,v,w)
 if $d[v] > d[u] + w(u,v)$
   then $d[v] \leftarrow d[u] + w(u,v)$
    $\pi[v] \leftarrow u$
```

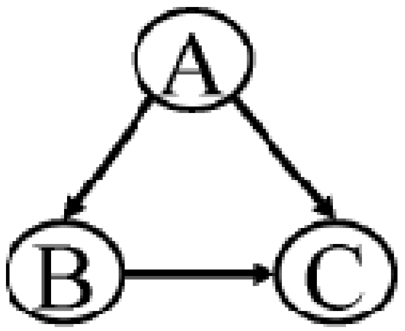
6.5. Cycle Detection

- Cycle detection on a graph is a bit different than on a tree due to the fact that a graph node can have multiple parents. On a tree, the algorithm for detecting a cycle is to do a depth first search, marking nodes as they are encountered. If a previously marked node is seen again, then a cycle exists. This won’t work on a graph.

© 2008, University of Colombo School of Computing

6.5. Cycle Detection contd…

- The graph in figure will be falsely reported to have a cycle, since node C will be seen twice in a DFS starting at node A.



An acyclic graph on which the tree cycle detection algorithm would fail.

6.5. Cycle Detection contd…

- The cycle detection algorithm for trees can easily be modified to work for graphs. The key is that in a DFS of an acyclic graph, a node whose descendants have all been visited can be seen again without implying a cycle. However, if a node is seen a second time before all of its descendants have been visited, then there must be a cycle. Can you see why this is? Suppose there is a cycle containing node A. Then this means that A must be reachable from one of its descendants. So when the DFS is visiting that descendant, it will see A again, before it has finished visiting all of A’s descendants. So there is a cycle.

6.5. Cycle Detection contd…

- In order to detect cycles, we use a modified depth first search called a colored DFS. All nodes are initially marked white. When a node is encountered, it is marked grey, and when its descendants are completely visited, it is marked black. If a grey node is ever encountered, then there is a cycle.

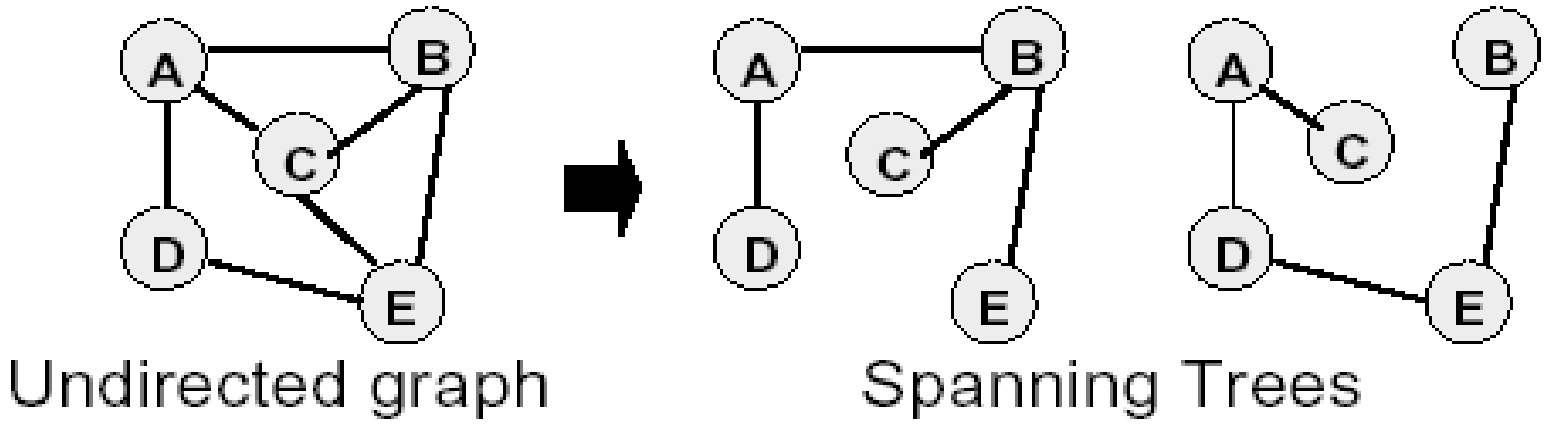
© 2008, University of Colombo School of Computing

Cycle detection algorithm.

```
boolean containsCycle(Graph g):
    for each vertex v in g do:
        v.mark = WHITE;
    od;
    for each vertex v in g do:
        if v.mark == WHITE then:
            if visit(g, v) then:
                return TRUE;
            fi;
        fi;
    od;
    return FALSE;
boolean visit(Graph g, Vertex v):
    v.mark = GREY;
    for each edge (v, u) in g do:
        if u.mark == GREY then:
            return TRUE;
        else if u.mark == WHITE then:
            if visit(g, u) then:
                return TRUE;
            fi;
        fi;
    od;
    v.mark = BLACK;
    return FALSE;
```

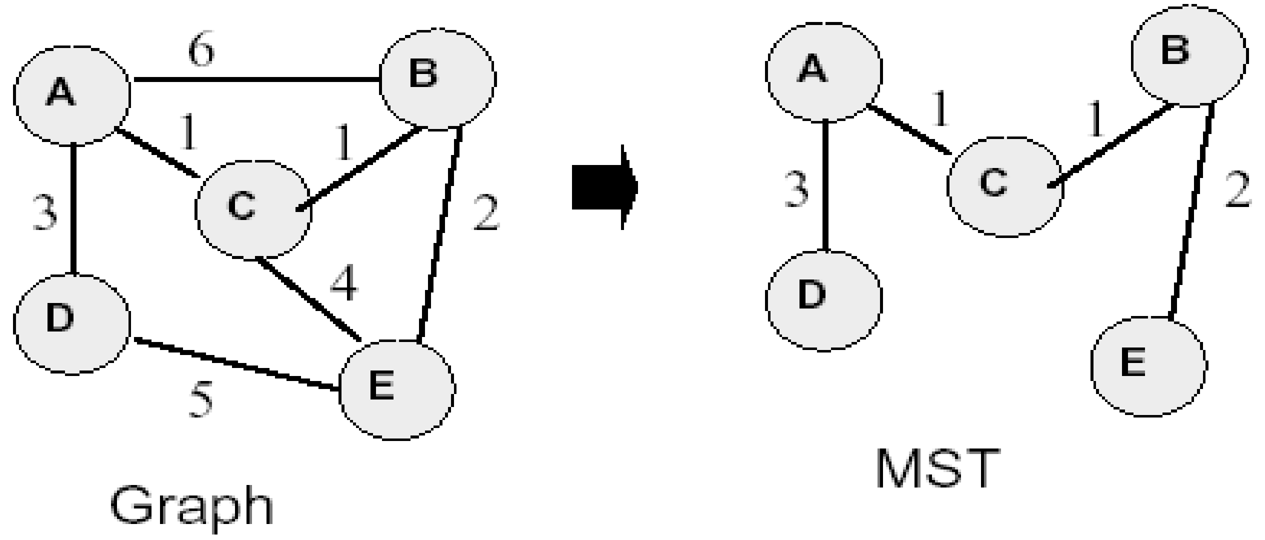
6.6. Spanning Tree

A spanning tree of a graph is a subgraph that is a tree containing all the vertices.



6.6.1. Minimum Spanning Tree (MST)

The spanning tree among all spanning trees with the lowest total edge weight.



6.6.2. Applications of MST Problem

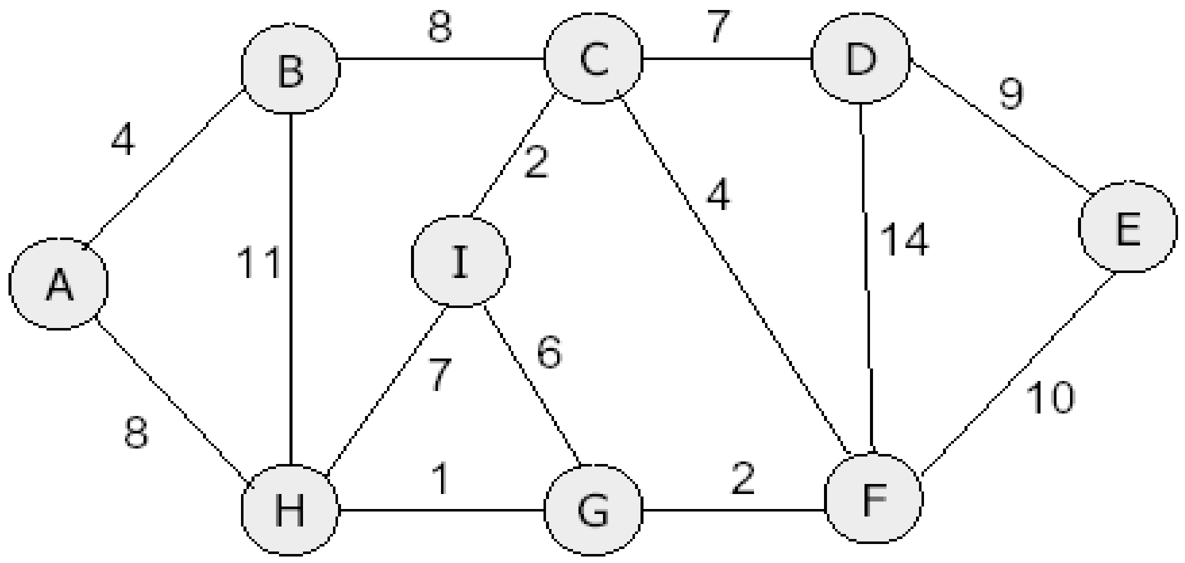
- Computer networks

  - How to connect a set of computers using the minimum amount of wire.

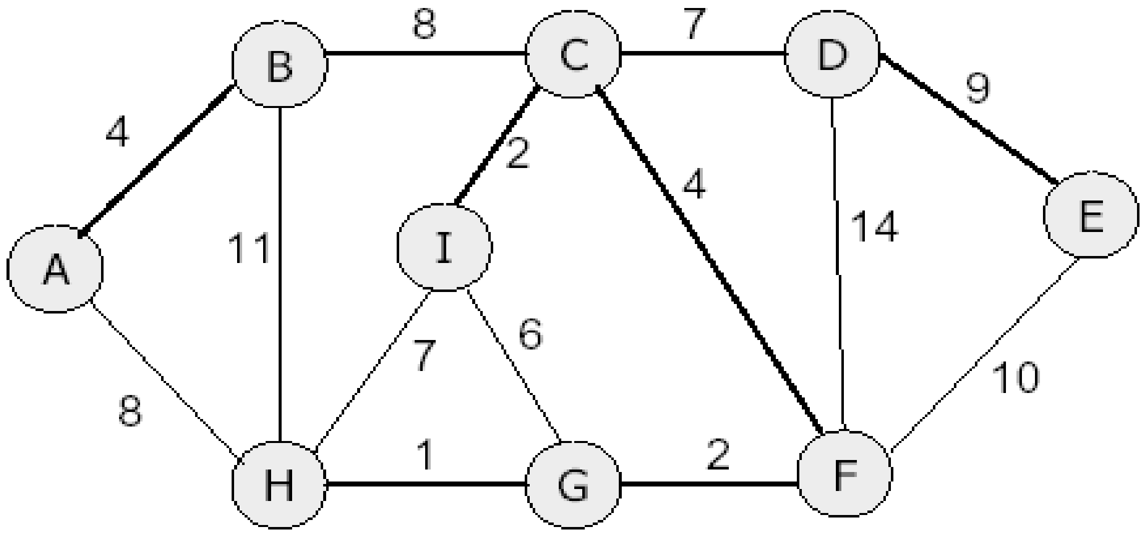
- Electronic circuits

6.6.3. Minimum Spanning Tree (MST)

Find the MST.



6.6.3.Solution



© 2008, University of Colombo School of Computing

6.6.4. Generic Algorithm for MST

Input : connected weighted graph, G

Output : MST, T, for graph G

Greedy strategy in the generic algorithm

- Grow the MST one edge at a time.
- Manage a set of edges A, that is prior to each iteration, A is a subset of some MST
  - At each step determine an edge (u,v) that can be added to A without violating this invariant.
  - We call such an edge a **safe edge** for A, since it can be safely added to A while maintaining the invariant.

6.6.4. Generic Algorithm for MST

Generic-MST(G,w)

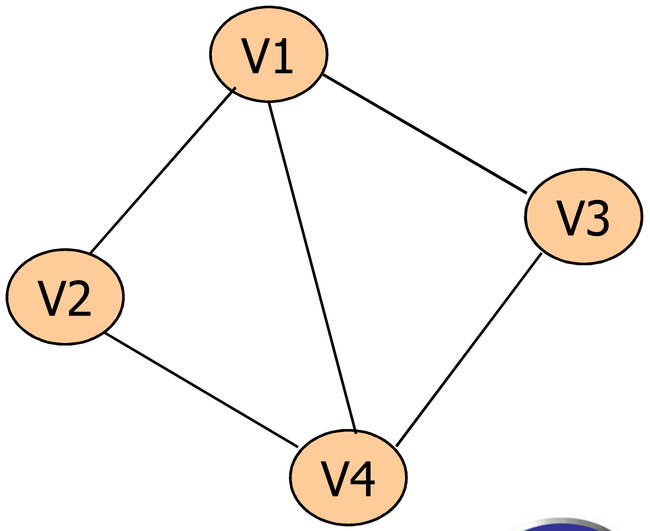
```
1. A ← 0
2. while A does not form a spanning tree
3.   do find an edge (u,v) that is safe for A
4.        A ← A U { (u,v)}
5. return A
```

6.7. Connectivity of graphs

Undirected Graph

- An *Undirected Graph* is a graph where the edges have no directions.
- The edges in an undirected graph are called *Undirected Edges.*

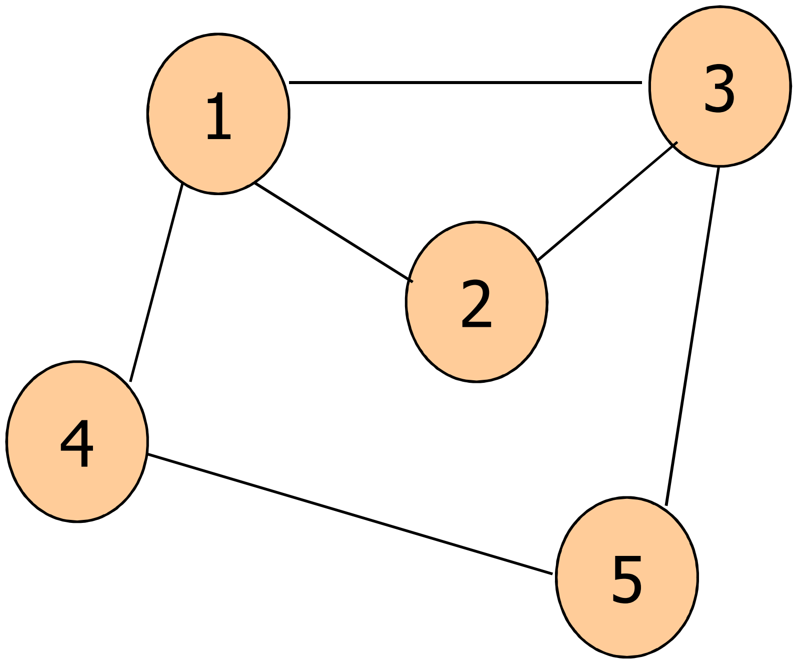
$\{vi,vj\} = \{vj, vi\}$



© 2008, University of Colombo School of Computing

6.7. Connectivity of graphs contd…

- Example (Undirected Graph)



G = (V, E)

V = {1, 2, 3, 4, 5}

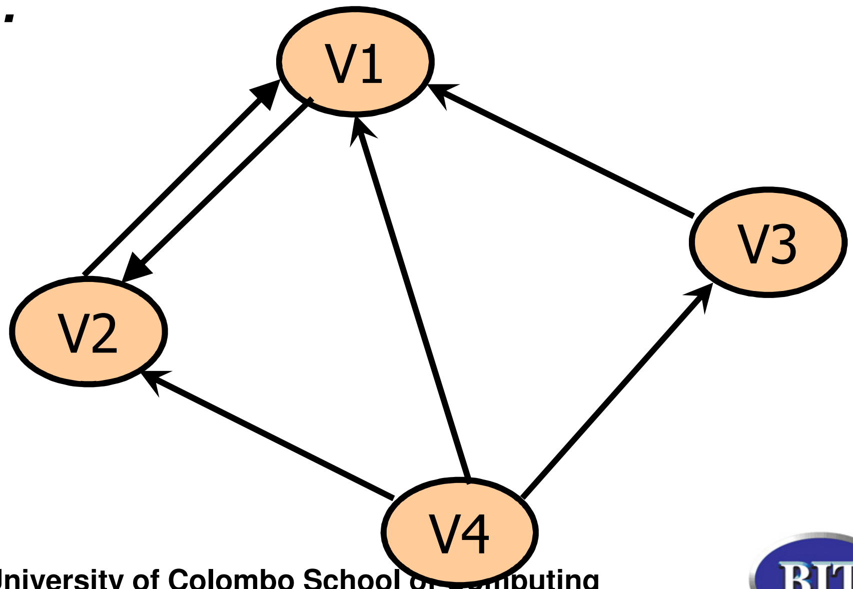
E = {(1,2), (1,3), (1,4), (2,3), (3,5), (4,5)}

6.7. Connectivity of graphs contd…

Directed Graphs

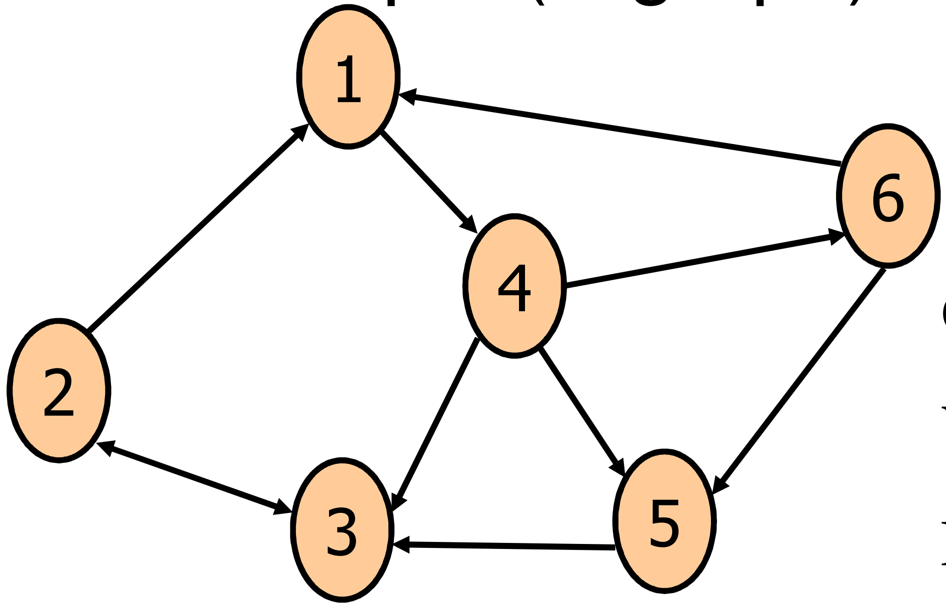
- A *Directed Graph* or *Digraph* is a graph where each edge has a direction.
- The edges in a digraph are called *Arcs* or *Directed Edges.*

$\{vi,vj\} \neq \{vj, vi\}$



6.7. Connectivity of graphs contd…

- Example (Digraph)



G = (V, E)

V = {1, 2, 3, 4, 5, 6}

E = {(1,4), (2,1), (2,3), (3,2), (4,3), (4,5), (4,6), (5,3), (6,1), (6,5)}

(1, 4) = 1→4 where 1 is the *tail*

and 4 is the *head*

6.8. Topological Sort

- Graphs are sometimes used to represent "before and after" relationships.
  For example, you need to think through a design for a program before you start coding.
- These two steps can be represented as vertices, and the relationship between them as a directed edge from the first to the second.

6.8. Topological Sort contd…

- On such graphs, it is useful to determine which steps must come before others. The topological sort algorithm computes an ordering on a graph such that if vertex β is earlier than vertex in the ordering, there is no path from β to α. In other words, you cannot get from a vertex later in the ordering to a vertex
- earlier in the ordering. Of course, topological sort works only on directed acyclic graphs.

© 2008, University of Colombo School of Computing

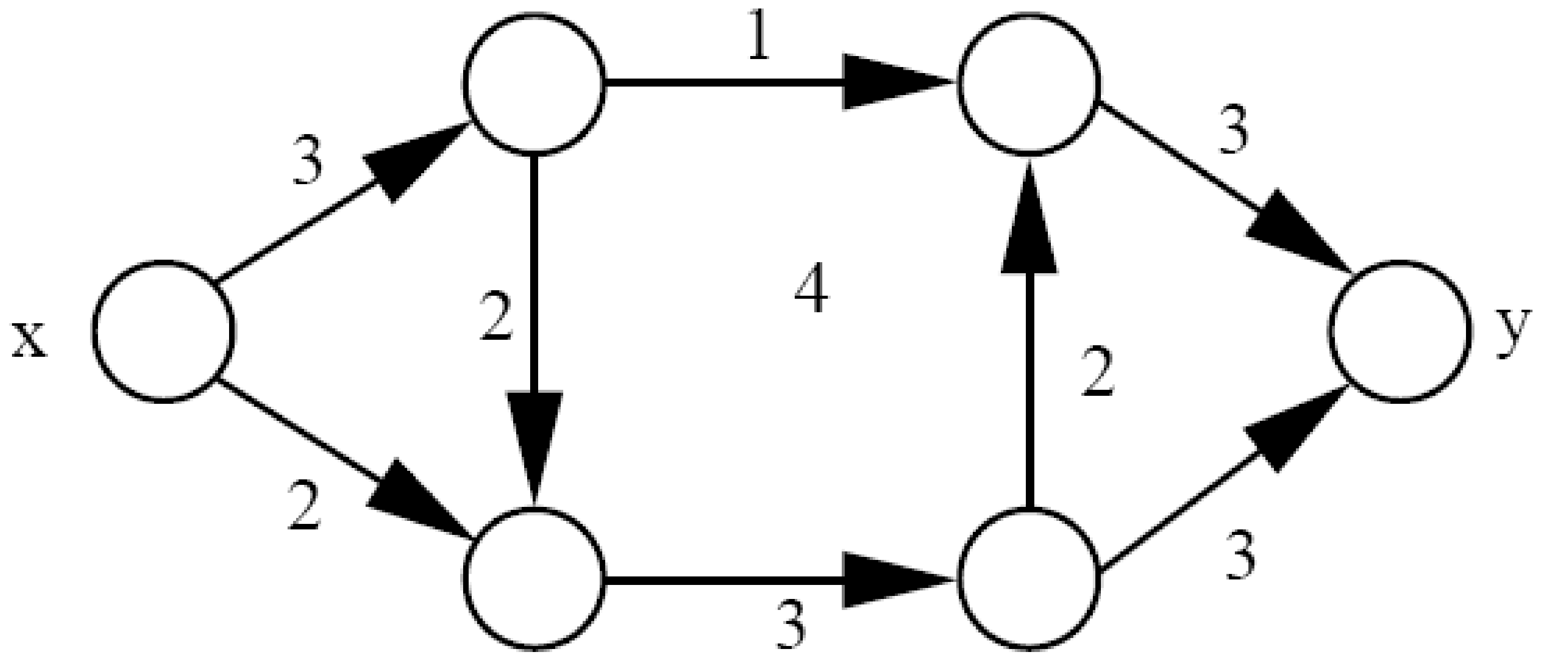
6.8. Topological Sort contd…

- The simplest topological sort algorithm is to repeatedly remove vertices with in-degree of 0 from the graph. The edges belonging to the vertex are also removed, reducing the in-degree of adjacent vertices. This is done until the graph is empty, or until no vertex without incoming edges exists, in which case the sort fails.

6.9. Networks

- Networks can be used to represent the transportation of some commodity through a system of delivery channels.
- There are sources (x) and sinks (y).
- The network is a directed graph, where each arc a is associated with a capacity, c(a).

6.9. Networks contd…



© 2008, University of Colombo School of Computing

6.9.1. Flow

- A flow in a network is a set of numbers associated with each arc, f (a).
- This indicates how much of a channel's capacity is being used.
- $0 \leq f(a) \leq c(a)$.
- For a vertex v, the flow into and out of the vertex is
- denoted by $f^{-}(v)$ and $f^{+}(v)$ respectively.
- For intermediate vertices (not sources or sinks) the flow in is the same as the flow out. This is called the conservation condition.

6.9.2. Resultant flow

- For some set of vertices S, the resultant flow out of S is given by $f^+(S) - f^-(S)$.
- We are often interested in the resultant flow out of the source x. (Or the set of sources X if there is more than one).
- In particular, we usually want to find a maximum flow, so that as much of the capacity is used as possible in transporting out of the sources to the sinks.
- It is straightforward to extend a network with multiple sources and sinks to one with just one source and sink in order to analyse the maximum flow.

6.9.3. Cuts

- A cut is a division of the vertices into two sets S and $\overline{S}$, so that the source is in S and the sink is in $\overline{S}$.
- The capacity of a cut is the sum of all the edges which cross between S and $\overline{S}$.
- How many cuts are possible in a network with vertices?
- What are the different cuts of the network on the board, and what are their capacities?

6.9.4. Max-flow min-cut

- In all the examples we have seen, the minimum capacity cut is the same as the maximum flow.
- Intuitively we can think of saturating the bottlenecks.
- To prove, we can show first that max flow $\leq$ min cut (no augmenting paths).
- Then show max flow $\geq$ min cut (removing edges changes capacity).

6.9.5. The Ford-Fulkerson Algorithm

- An algorithm for finding the maximum flow in a network.

1. Set the flow to zero for all arcs.
2. Calcuate the residual network G_f . While there is a path p from x to y in G_f :
   - Find $c_f(p) = \min\{c_f(u, v) | (u, v) \in p\}$
   - For each edge in p, add cf (p) to the flow.

   (Subtract cf (p) from the flow if the edge is a reverse arc in the network).

– Repeat step 2 until there is no augmenting path.

6.9.5. The Ford-Fulkerson Algorithm contd…

The residual network G_f has:

- $c_f(u, v) = c(u, v) - f(u, v)$.
- No flow on any edges.

6.9.6. Other problems regarding network flow

- Multi commodity flow: a number of sources produce different products that are to be transported to different sinks using the same network.
- Mimimum cost flow: each arc has an associated cost, and we want to find the cheapest mode of transportation.
- Circulation: there is a lower bound on the flow as well as an upper bound.