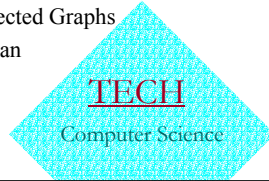
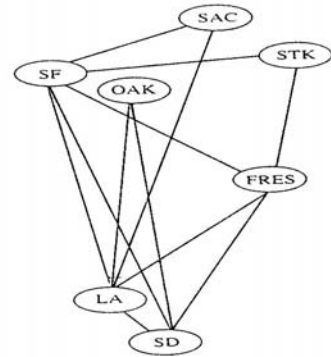


## Graphs and Graph Traversals

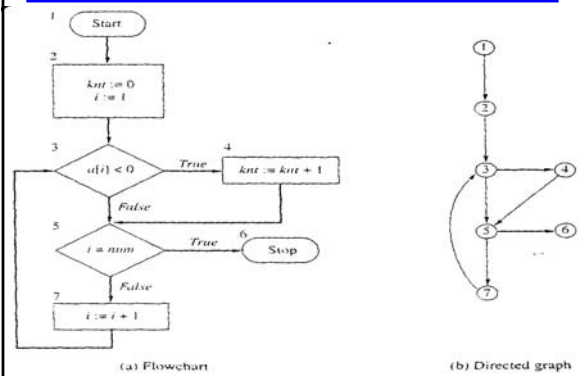
- // From Tree to Graph
- // Many programs can be cast as problems on graph
- Definitions and Representations
- Traversing Graphs
- Depth-First Search on Directed Graphs
- Strongly Connected Components of a Directed Graph
- Depth-First Search on Undirected Graphs
- Biconnected Components of an Undirected Graph



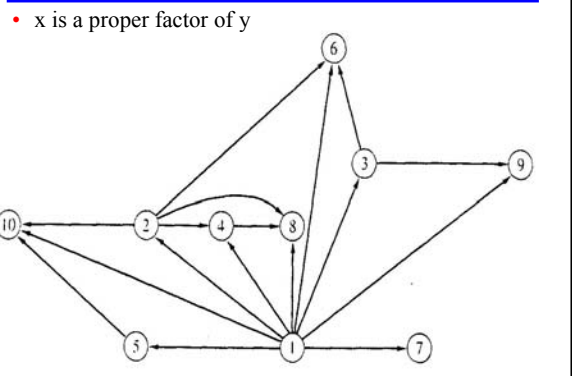
## Problems: e.g. Airline Routes



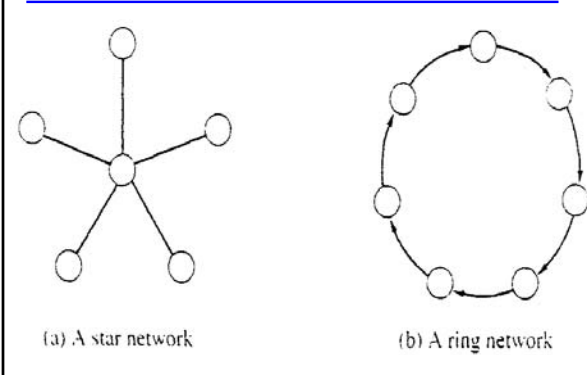
## Problems: e.g. Flowcharts



## Problems: e.g. Binary relation



## Problems: e.g. Computer Networks



## Definition: Directed graph

- Directed Graph
  - A directed graph, or digraph, is a pair
  - $G = (V, E)$
  - where  $V$  is a set whose elements are called vertices, and
  - $E$  is a set of ordered pairs of elements of  $V$ .
- Vertices are often also called nodes.
- Elements of  $E$  are called edges, or directed edges, or arcs.
- For directed edge  $(v, w)$  in  $E$ ,  $v$  is its tail and  $w$  its head;
- $(v, w)$  is represented in the diagrams as the arrow,  $v \rightarrow w$ .
- In text we simply write  $vw$ .

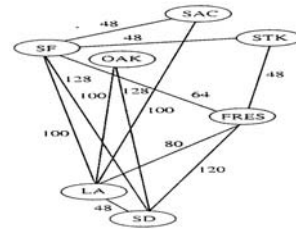
## Definition: Undirected graph

- Undirected Graph
  - A undirected graph is a pair
  - $G = (V, E)$
  - where  $V$  is a set whose elements are called vertices, and
  - $E$  is a set of *unordered* pairs of *distinct* elements of  $V$ .

- Vertices are often also called nodes.
- Elements of  $E$  are called edges, or undirected edges.
- Each edge may be considered as a subset of  $V$  containing two elements,
- $\{v, w\}$  denotes an undirected edge
- In diagrams this edge is the line  $v$ --- $w$ .
- In text we simple write  $vw$ , or  $wv$
- $vw$  is said to be *incident* upon the vertices  $v$  and  $w$

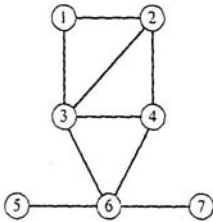
## Definitions: Weighted Graph

- A weighted graph is a triple  $(V, E, W)$ 
  - where  $(V, E)$  is a graph (directed or undirected) and
  - $W$  is a function from  $E$  into  $\mathbb{R}$ , the reals (integer or rationals).
  - For an edge  $e$ ,  $W(e)$  is called the weight of  $e$ .



## Graph Representations using Data Structures

- Adjacency Matrix Representation
  - Let  $G = (V, E)$ ,  $n = |V|$ ,  $m = |E|$ ,  $V = \{v_1, v_2, \dots, v_n\}$
  - $G$  can be represented by an  $n \times n$  matrix

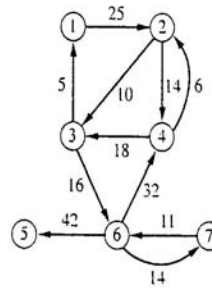


(a) An undirected graph

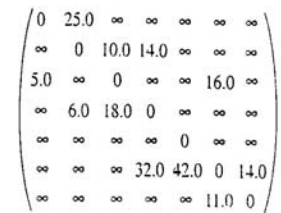


(b) Its adjacency matrix

## Adjacency Matrix for weight digraph



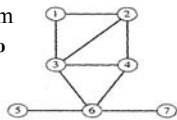
(a) A weighted digraph



(b) Its adjacency matrix

## Array of Adjacency Lists Representation

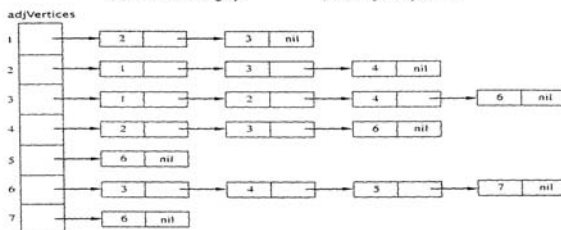
- From
  - to



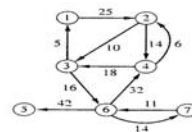
(a) An undirected graph



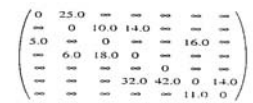
(b) Its adjacency matrix.



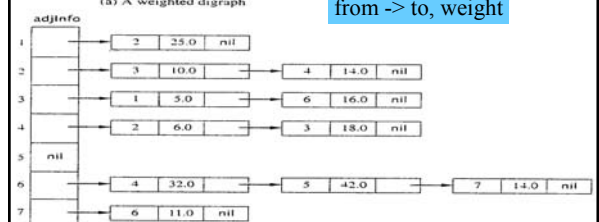
## Array of Adjacency Lists Representation



(a) A weighted digraph



from -> to, weight



## More Definitions

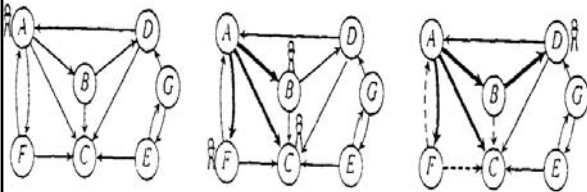
- Subgraph
- Symmetric digraph
- complete graph
- Adjacency relation
- Path, simple path, reachable
- Connected, Strongly Connected
- Cycle, simple cycle
- acyclic
- undirected forest
- free tree, undirected tree
- rooted tree
- Connected component

## Traversing Graphs

- Most algorithms for solving problems on a graph examine or process each vertex and each edge.
- Breadth-first search and depth-first search
  - are two traversal strategies that provide an efficient way to “visit” each vertex and edge exactly once.
- Breadth-first search: Strategy (for digraph)
  - choose a starting vertex, distance  $d = 0$
  - vertices are visited in order of increasing distance from the starting vertex,
  - examine all edges leading from vertices (at distance  $d$ ) to adjacent vertices (at distance  $d+1$ )
  - then, examine all edges leading from vertices at distance  $d+1$  to distance  $d+2$ , and so on,
  - until no new vertex is discovered

## Breadth-first search, e.g.

- e.g. Start from vertex A, at  $d = 0$ 
  - visit B, C, F; at  $d = 1$
  - visit D; at  $d = 2$
- e.g. Start from vertex E, at  $d = 0$ 
  - visit G; at  $d = 1$



## Breadth-first search: I/O Data Structures

*Input:*  $G = (V, E)$ , a graph represented by an adjacency list structure, `adjVertices`, as described in Section 7.2.3, where  $V = \{1, \dots, n\}$ ;  $s \in V$ , the vertex from which the search begins.

*Output:* A breadth-first spanning tree, stored in the parent array. The parent array is passed in and the algorithm fills it.

*Remarks:* For a queue  $Q$ , we assume operations of the Queue abstract data type (Section 2.4.2) are used. The array `color[1], ..., color[n]` denotes the current search status of all vertices. Undiscovered vertices are white; those that are discovered but not yet processed (in the queue) are gray; those that are processed are black.

## Breadth-first search: Algorithm

```
void breadthFirstSearch(intList[] adjVertices, int n, int s, int[] parent)
int[] color = new int[n+1];
Queue pending = create(n);
Initialize color[1], ..., color[n] to white.

parent[s] = -1;
color[s] = gray;
enqueue(pending, s);
while (pending is nonempty)
    v = front(pending);
    dequeue(pending);
    For each vertex w in the list adjVertices[v]:
        if (color[w] == white)
            color[w] = gray;
            enqueue(pending, w);
            parent[w] = v; // Process tree edge vw.
            // Continue through list.
        // Process vertex v here.
    color[v] = black;
return;
```

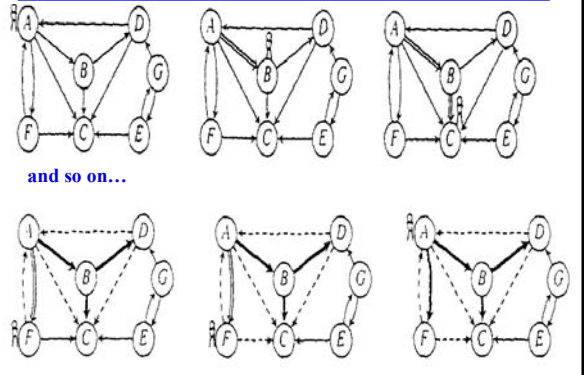
## Breadth-first search: Analysis

- For a digraph having  $n$  vertices and  $m$  edges
  - Each edge is processed once in the while loop for a cost of  $\theta(m)$
  - Each vertex is put into the queue once and removed from the queue and processed once, for a cost  $\theta(n)$
  - Extra space is used for color array and queue, there are  $\theta(n)$
- From a tree (breadth-first spanning tree)
  - the path in the tree from start vertex to any vertex contains the *minimum* possible number of edges
- Not all vertices are necessarily reachable from a selected starting vertex

## Depth-first search for Digraph

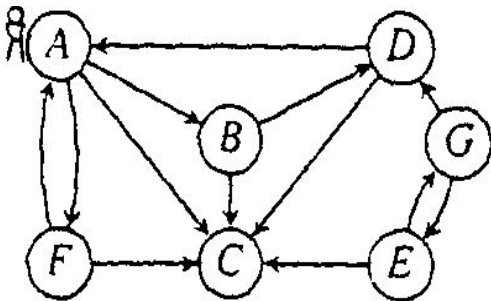
- Depth-first search: Strategy (for digraph)
  - choose a starting vertex, distance  $d = 0$
  - vertices are visited in order of increasing distance from the starting vertex,
  - examine One edges leading from vertices (at distance  $d$ ) to adjacent vertices (at distance  $d+1$ )
  - then, examine One edges leading from vertices at distance  $d+1$  to distance  $d+2$ , and so on,
  - until no new vertex is discovered, or dead end
  - then, backtrack one distance back up, and try other edges, and so on
  - until finally backtrack to starting vertex, with no more new vertex to be discovered.

## Depth-first search for digraph, e.g. from a tree, remember where we have been



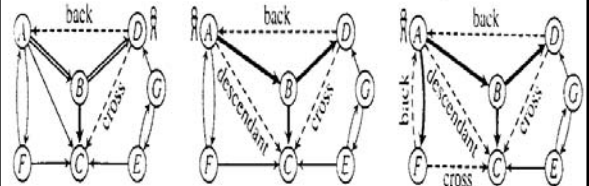
## Depth-first Search, e.g. trace it, in order

- Vertex status: undiscovered, discovered, finished
- Edge status: exploring, backtrack, checked



## Depth-first search tree

- edges classified:
  - tree edge, back edge, descendant edge, and cross edge



## Depth-first search algorithm: outline

```
dfs(G, v) // OUTLINE
  Mark v as "discovered."
  For each vertex w such that edge vw is in G:
    If w is undiscovered:
      dfs(G, w); that is, explore vw, visit w, explore from there
      as much as possible, and backtrack from w to v.
    Otherwise:
      "Check" vw without visiting w.
  Mark v as "finished."
```

## Reaching all vertices

```
dfsSweep(G) // OUTLINE
  Initialize all vertices of G to "undiscovered."
  For each vertex v in G, in some order:
    If v is undiscovered:
      dfs(G, v); that is, perform a depth-first search beginning
      (and ending) at v; any vertices discovered during an earlier
      depth-first search visit are not revisited: all vertices visited
      during this dfs are now classified as "discovered."
```

## Depth-first search algorithm

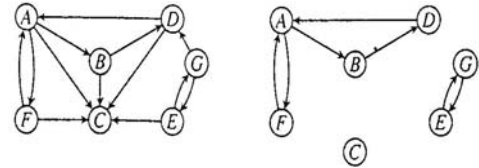
```

int dfs(IntList[] adjVertices, int[] color, int v, ...)
int w;
IntList remAdj;
int ans;
1. color[v] = gray;
2. Preorder processing of vertex v
3. remAdj = adjVertices[v];
4. while (remAdj ≠ nil)
5.   w = first(remAdj);
6.   if (color[w] == white)
7.     Exploratory processing for tree edge vw
8.     int wAns = dfs(adjVertices, color, w, ...);
9.     Backtrack processing for tree edge vw, using wAns (like inorder)
10.  else
11.    Checking (i.e., processing) for nontree edge vw
12.    remAdj = rest(remAdj)
13. Postorder processing of vertex v, including final computation of ans
14. color[v] = black;
15. return ans;

```

## Strongly Connected Components of a Digraph

- Strongly connected:
  - A directed graph is strongly connected if and only if, for each pair of vertices  $v$  and  $w$ , there is a path from  $v$  to  $w$ .
- Strongly connected component:
  - A strongly connected component of a digraph  $G$  is a maximal strongly connected subgraph of  $G$ .



## Strongly connected Components and Equivalence Relations

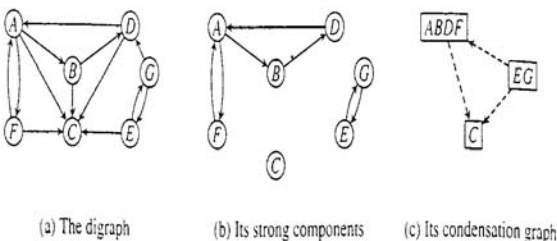
- Strongly Connected Components may be defined in terms of an equivalence relation,  $S$ , on the vertices
  - $vSw$  iff there is a path from  $v$  to  $w$  and
  - a path from  $w$  to  $v$
- Then, a strongly connected component consists of one equivalence class,  $C$ , along with all edges  $vw$  such that  $v$  and  $w$  are in  $C$ .

## Condensation graph

- The strongly connected components of a digraph can each be collapsed to a single vertex yielding a new digraph that has no cycles.
- Condensation graph:
  - Let  $S_1, S_2, \dots, S_p$  be the strong components of  $G$ .
  - The condensation graph of  $G$  denoted as  $G\downarrow$ , is the digraph  $G\downarrow = (V', E')$ ,
  - where  $V'$  has  $p$  elements  $s_1, s_2, \dots, s_p$  and
  - $s_i, s_j$  is in  $E'$  if and only if  $i \neq j$  and
  - there is an edge in  $E$  from some vertex in  $S_i$  to some vertex in  $S_j$ .

## Condensation graph and its strongly connected components

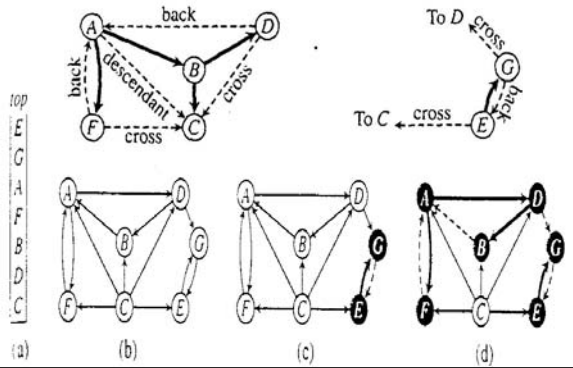
- Condensation Graph is acyclic.



## Algorithm to Find Strongly Connected Component

- Strategy:
  - Phase 1:
    - A standard depth-first search on  $G$  is performed,
    - and the vertices are put in a stack at their finishing times
  - Phase 2:
    - A depth-first search is performed on  $G^T$ , the transpose graph.
    - To start a search, vertices are popped off the stack.
    - A strongly connected component in the graph is identified by the name of its starting vertex (call leader).

## The strategy in Action, e.g.



## Depth-First Search on Undirected Graphs

- Depth-first search on an undirected graph is complicated by the fact that edges should be *explored in one direction only*,
- but they are represented twice in the data structure (*symmetric digraph equivalence*)
- Depth-first search provides an orientation for each of its edges
  - they are oriented in the direction in which they are first encountered (during exploring)
  - the reverse direction is then ignored.

## Algorithm for depth-first search on undirected graph

```

int dfs(int l, int[] adjVertices, int[] color, int v, int p, ...)
int w;
int l, int remAdj;
int ans;

1. color[v]=gray;
2. Preorder processing of vertex v
3. remAdj=adjVertices[v];
4. while(remAdj<>nil)
5. w=first(remAdj);
6. if(color[w]==white)
7. Exploratory processing for tree edge vw.
8. int wAns=dfs(adjVertices,color,w,v,...)
9. BackTrack processing for tree edge vw using wAns(like in order)
10. else if(color[w]==gray && w<>p)
11. Checking back edge vw
    //else vw was traversed, so ignore vw.
12. remAdj=rest(remAdj)
13. Postorder processing of vertex v, including final computation of ans
14. color[v]=black;
15. return ans;
    
```

## Breadth-first Search on Undirected Graph

- Simply treat the undirected graph as symmetric digraph
  - in fact undirected graph is represented in adjacency list as symmetric digraph
- Each edge is processed once in the “forward” direction
  - whichever direction is encountered (explored) first is considered “forward” for the duration of the search

## Bi-connected components of an Undirected graph

- Problem:
  - If any *one* vertex (and the edges incident upon it) are removed from a connected graph,
  - is the remaining subgraph still connected?
- Biconnected graph:
  - A connected undirected graph G is said to be biconnected if it remains connected after removal of any one vertex and the edges that are incident upon that vertex.
- Biconnected component:
  - A biconnected component of a undirected graph is a maximal biconnected subgraph, that is, a biconnected subgraph not contained in any larger biconnected subgraph.
- Articulation point:
  - A vertex v is an articulation point for an undirected graph G if there are distinct vertices w and x (distinct from v also) such that v is in every path from w to x.

## Bi-connected components, e.g.

- Some vertices are in more than one component
- (which vertices are these?)

