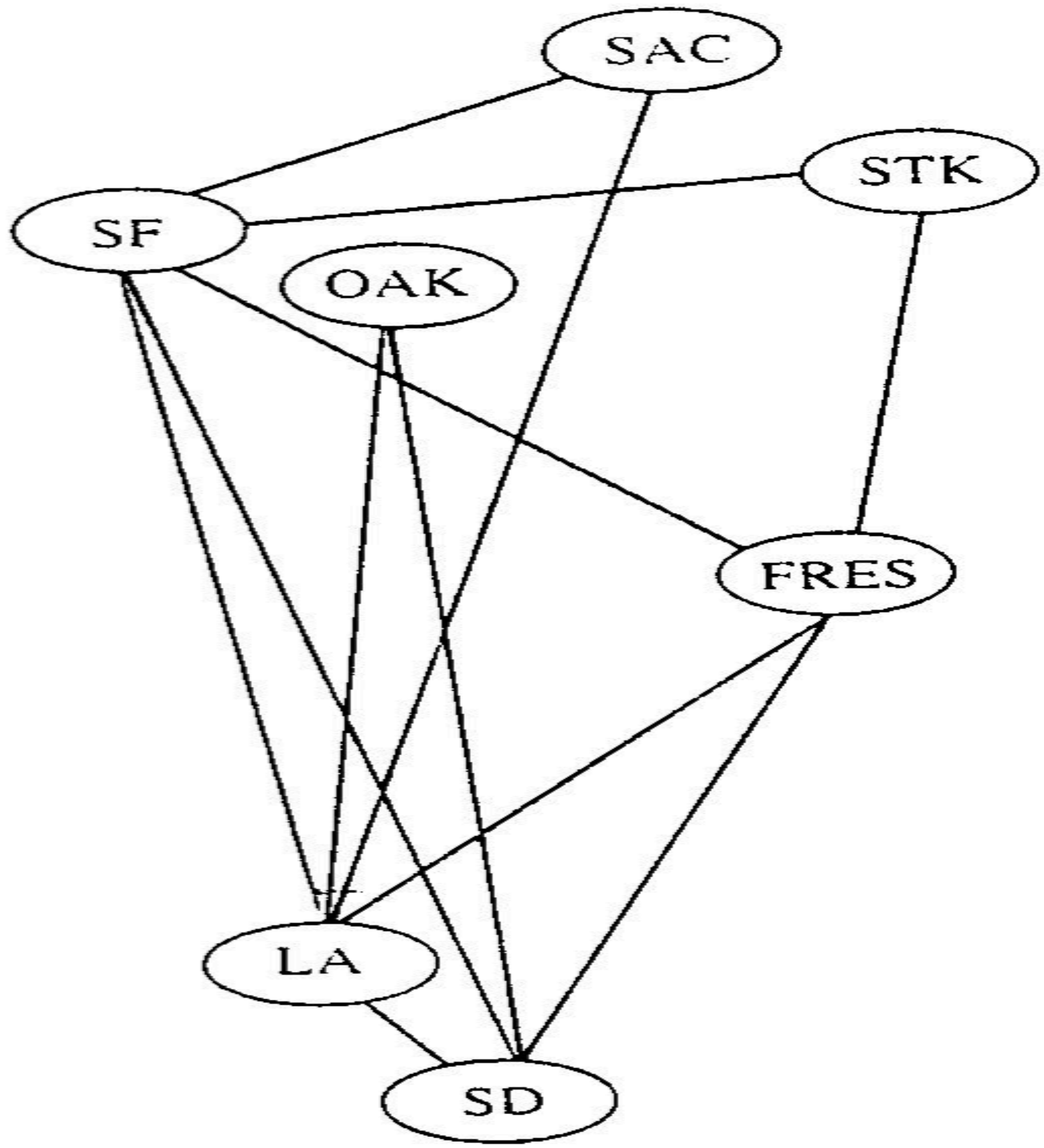
CS4102: Graph Traversals

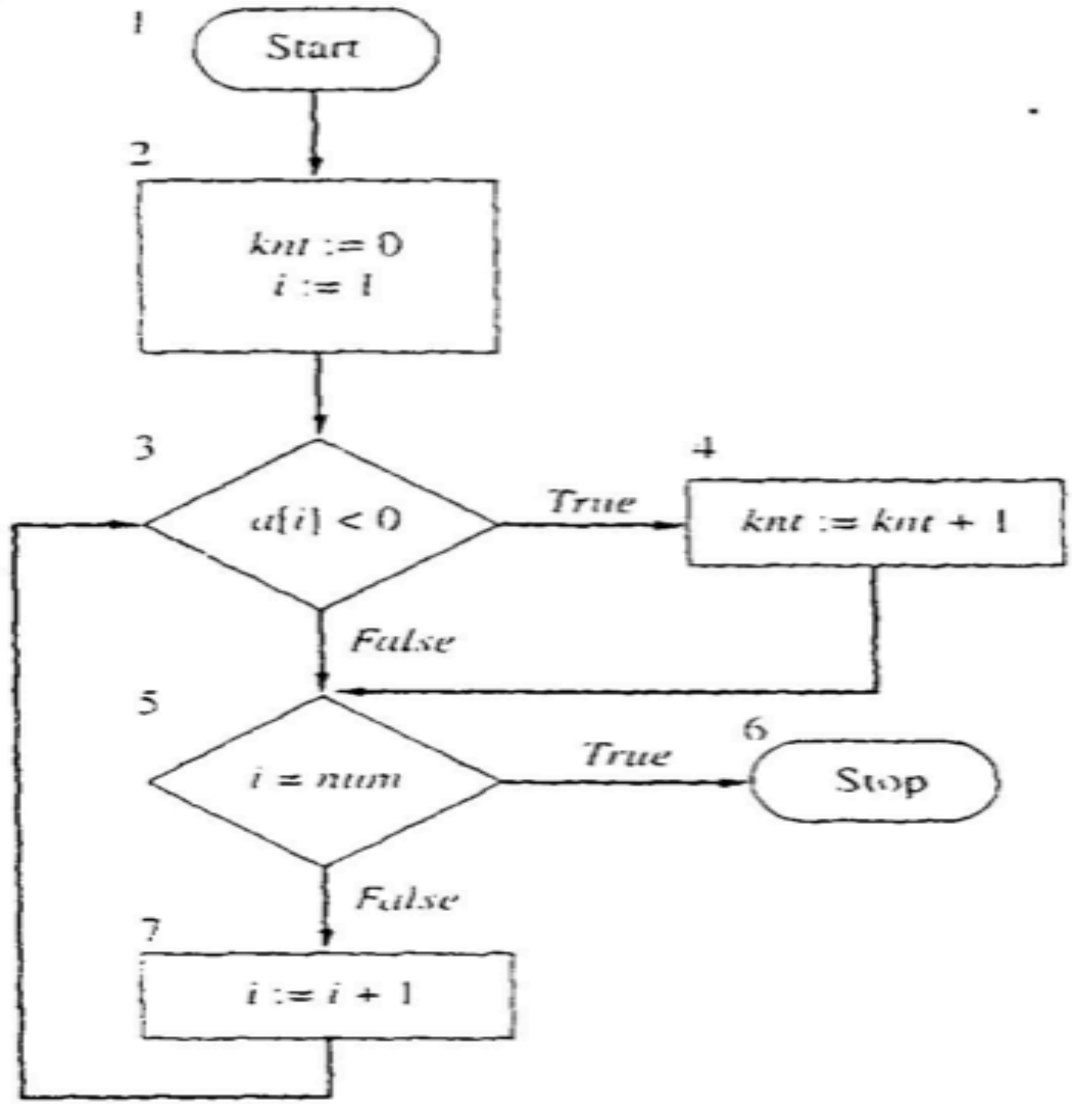
- Review: Section 2.5
  - Definitions, data structures
  - Note: review definitions, data structures, and BFS from CS216 slides from 4-16-03

    http://www.cs.virginia.edu/~cs216/notes/slides/graphs2.pdf
- Read: Chapter 4 (from 4.2 on)
  - Traversing Graphs
    - Depth-first Search (DFS)
    - Breadth-first Search (BFS)
  - Applications of DFS strategy (things not in text)
  - Backtracking, Exhaustive Search (handout)

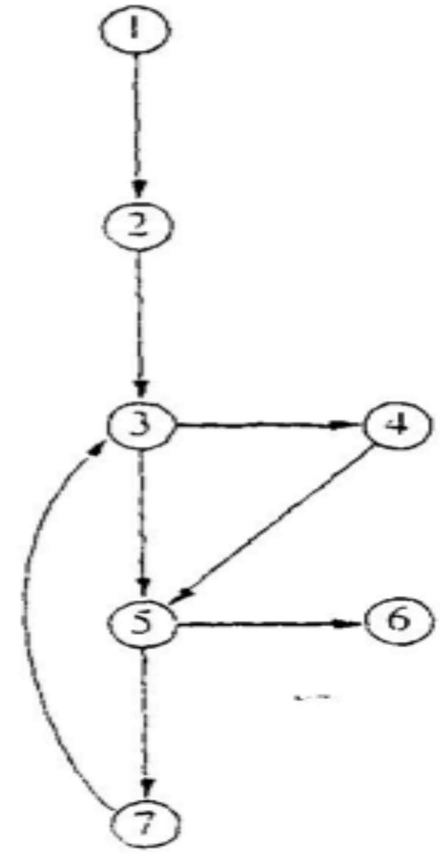
Problems: e.g. Airline Routes



Problems: e.g. Flowcharts



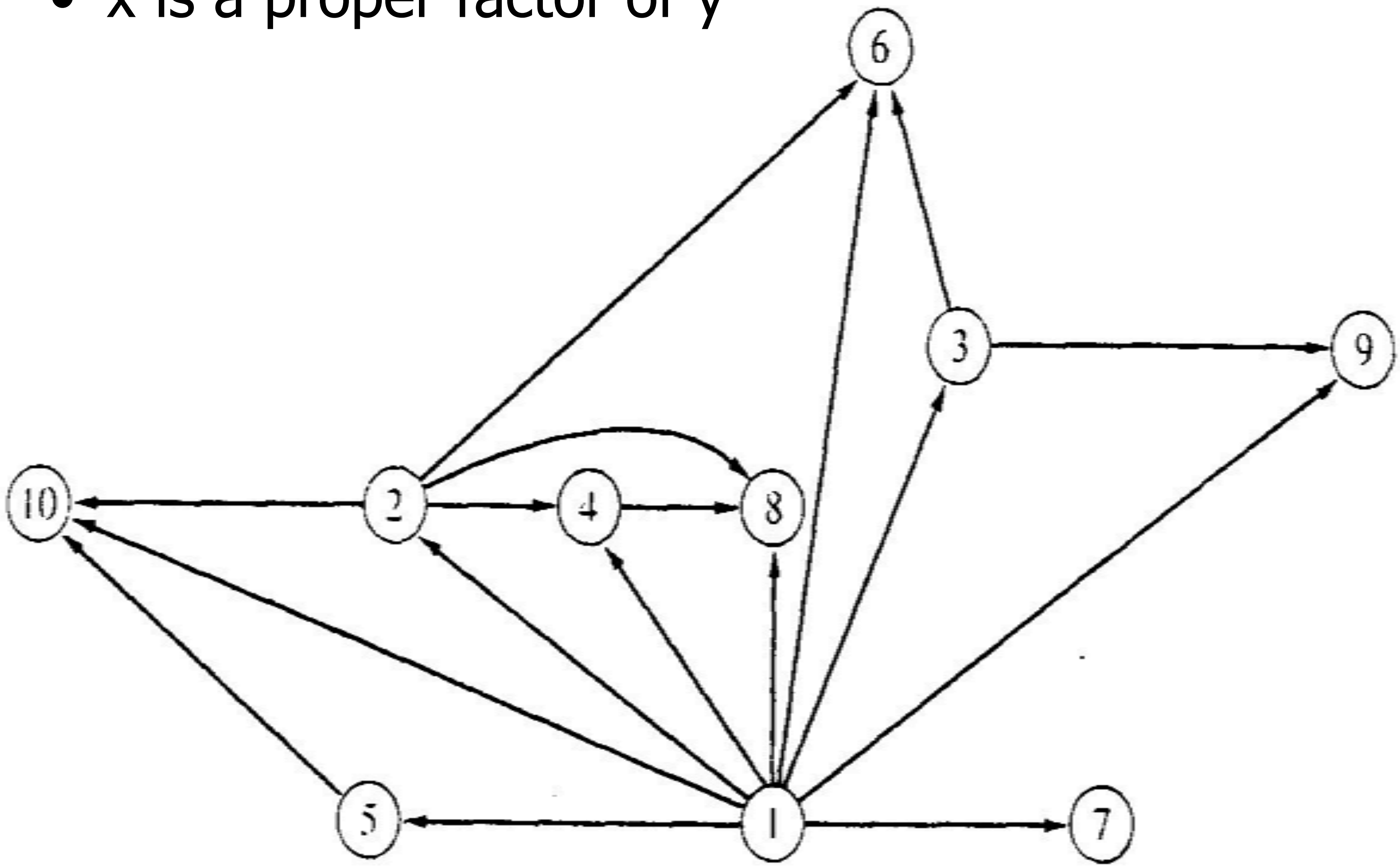
(a) Flowchart



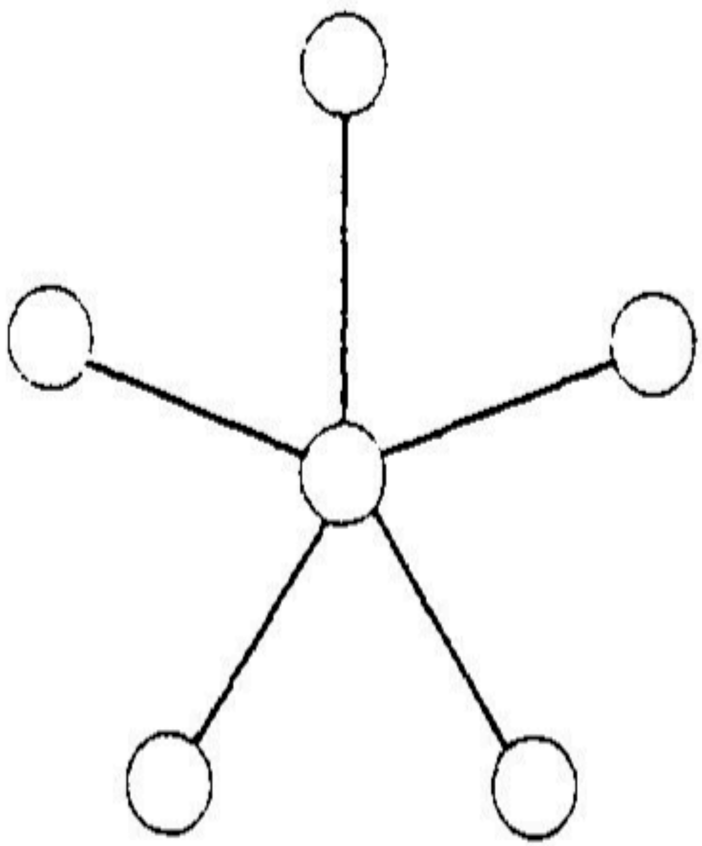
(b) Directed graph

Problems: e.g. Binary relation

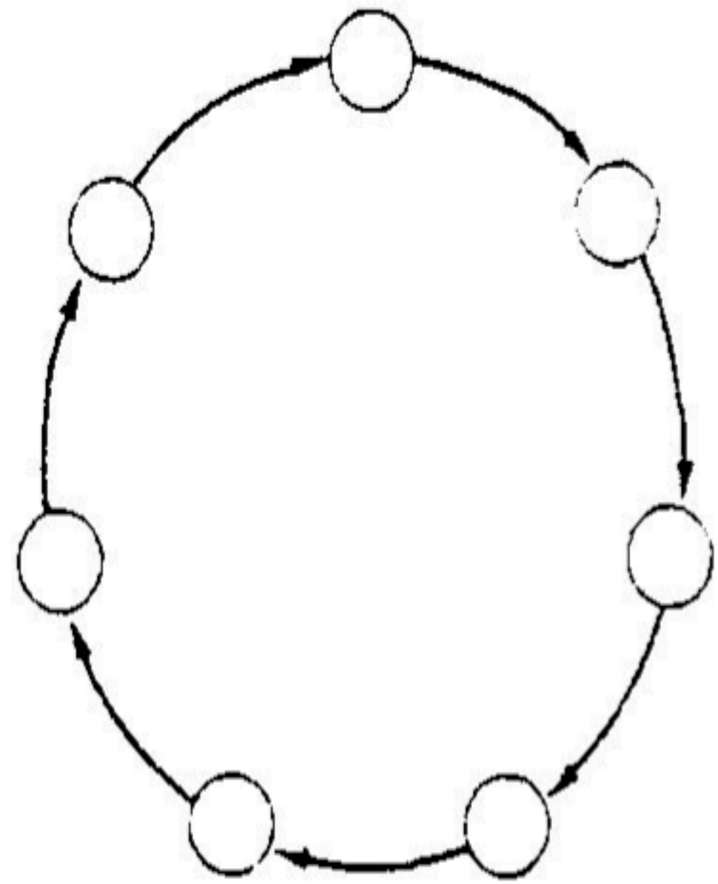
- x is a proper factor of y



Problems: e.g. Computer Networks



(a) A star network



(b) A ring network

Terms You Should Know or Learn Now

- Vertex (plural *vertices*) or Node
- Edge (sometimes referred to as an *arc*)
  - Note the meaning of *incident*
- Degree of a vertex: how many adjacent vertices
  - Digraph: in-degree (num. of incoming edges) vs. out-degree
- Graphs can be:
  - Directed or undirected
  - Weighted or not weighted
    - weights can be reals, integers, etc.
    - weight also known as: cost, length, distance, capacity,…
- Undirected graphs:
  - Normally an edge can't connect a vertex to itself
- A directed graph (also known as a *digraph*)
  - "Originating" node is the *head*, the target the *tail*
  - An edge may connect a vertex to itself

Terms You Should Know or Learn Now

- Size of graph? Two measures:
  - Number of nodes. Usually n
  - Number of edges: usually m
- Dense graph: many edges
  - Maximally dense?
  - Undirected: each node connects to all others, so
    $m = n(n-1)/2$
    Called a *complete graph*
  - Directed: $m = n(n-1)$ *why?*
- Sparse graph: fewer edges
  - Could be zero edges…

Terms You Should Know or Learn Now

- Path vs. simple path
  - One vertex is *reachable* from another vertex
- A *connected graph*
  - undirected graph, where each vertex is reachable from all others
- A *strongly connected digraph:*
  - direction affects this!
  - node u may be reachable from v, but not v from u
  - Strongly connected means both directions
- Connected components for undirected graphs

Terms You Should Know or Learn Now

- Cycle
  - Directed graph: non-empty path with same starting and ending node
  - An edge may appear more than once (but why?)
    - **Simple cycle**: no node repeated except start and end
  - Undirected graph: same idea
    - If an edge appears more than once (I.e. non-simple) then we traverse it in the same direction
- Acyclic: no-cycles
- A connected, acyclic undirected graph: *free tree*
  - If we specificy a root, it's a *rooted tree*
  - Acyclic but not connected? a undirected *forest*
- Directed acyclic graph: a DAG

Self-test: Understand these Terms?

- 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

Definition: Directed graph

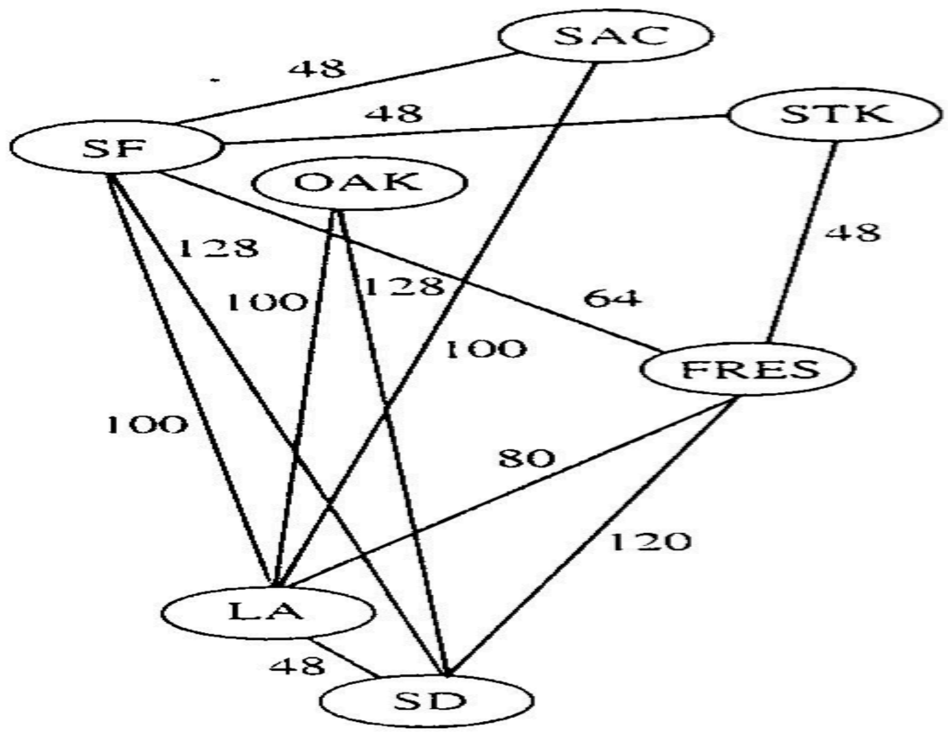
- Directed Graph
  - A directed graph, or digraph, is a pair
  - G = (V, E)
  - where V is a set whose elements are called vertices, and
  - E is a set of ordered pairs of elements of V.
    - Vertices are often also called nodes.
    - Elements of E are called edges, or directed edges, or arcs.
    - 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 -> w.
    - In text we simple 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

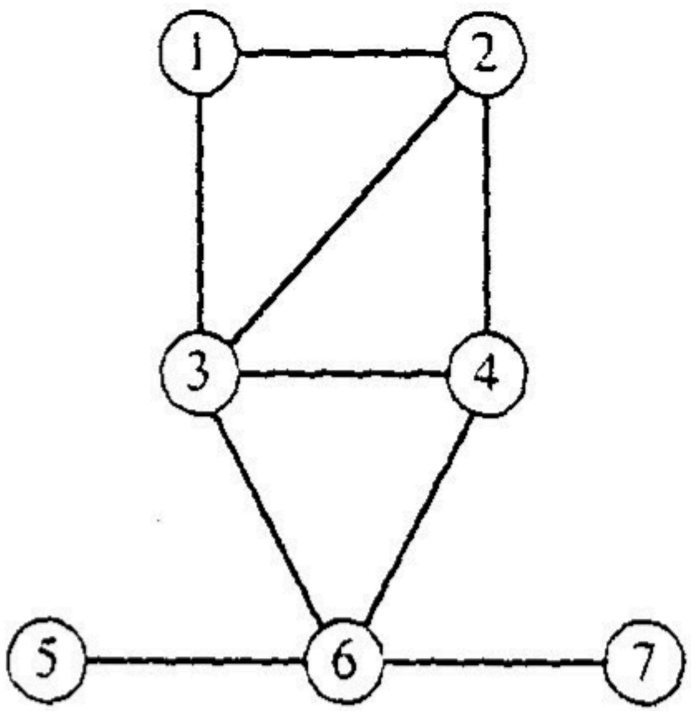
Definitions: Weighted Graph

- 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 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 = {v1, v2, …, vn)
  - G can be represented by an n × n matrix



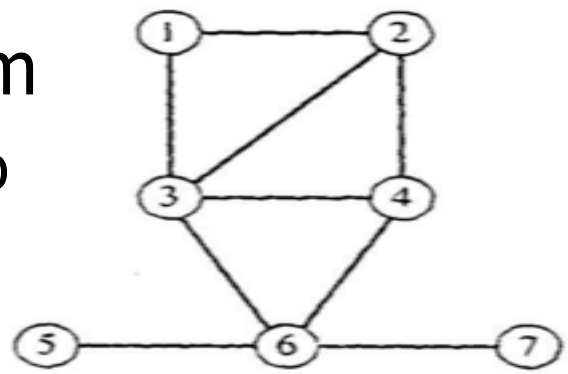
(a) An undirected graph

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) Its adjacency matrix

Array of Adjacency Lists Representation

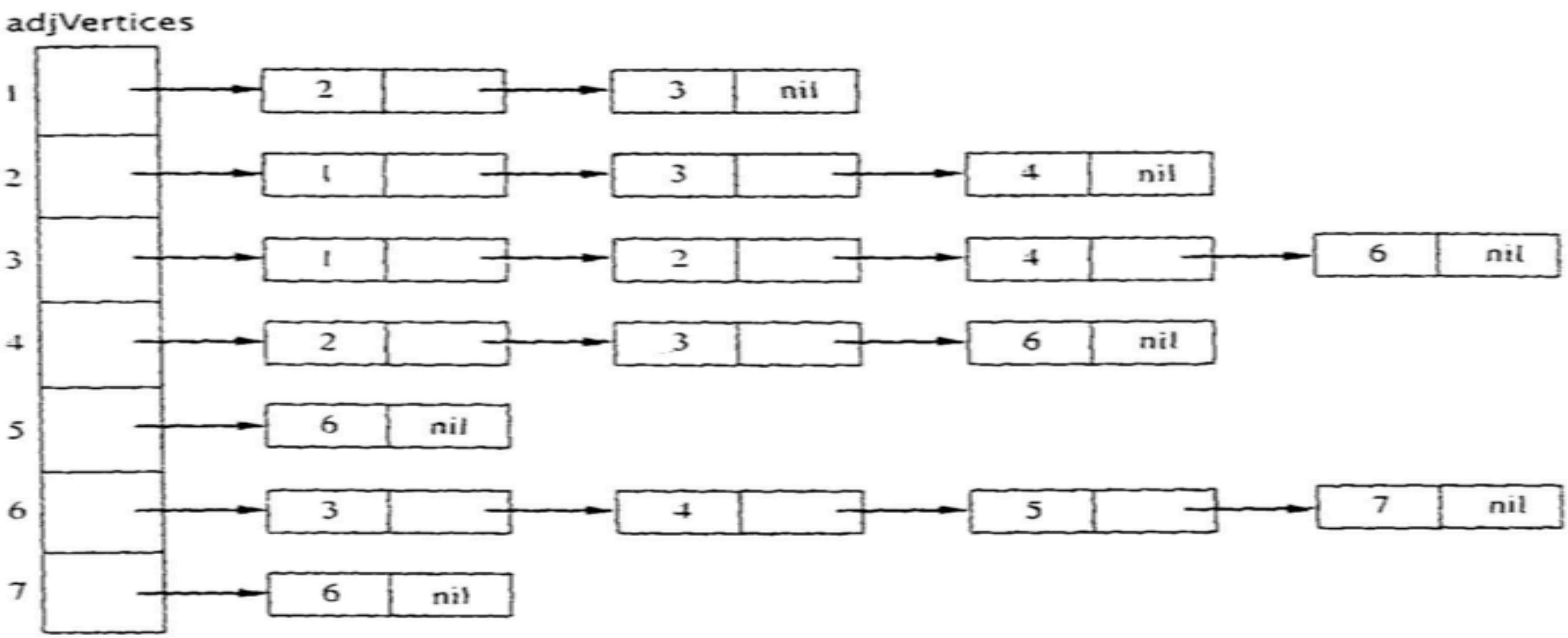
- From
  - to



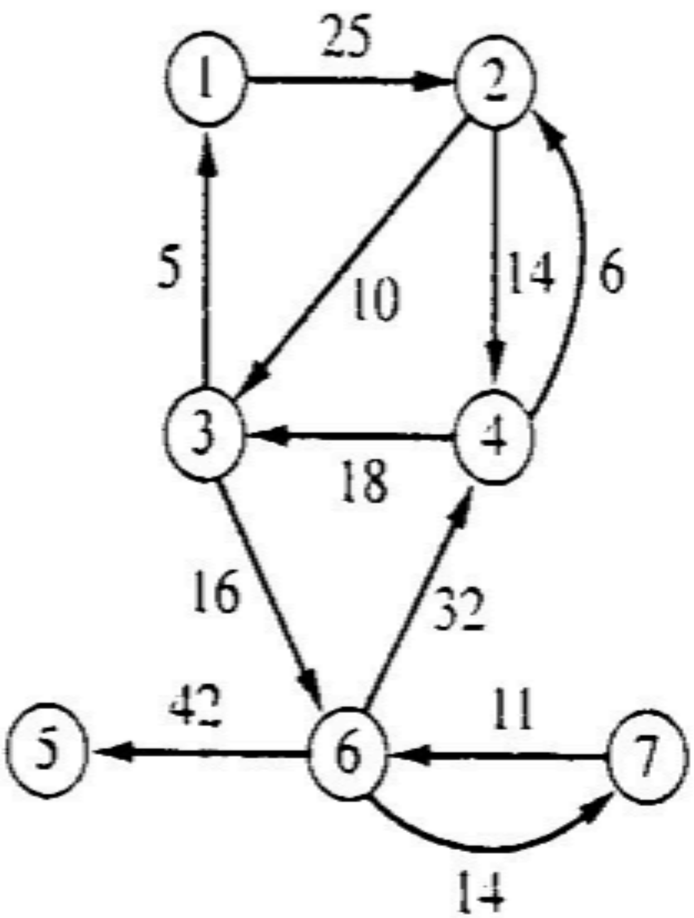
(a) An undirected graph

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) Its adjacency matrix



Adjacency Matrix for weight digraph

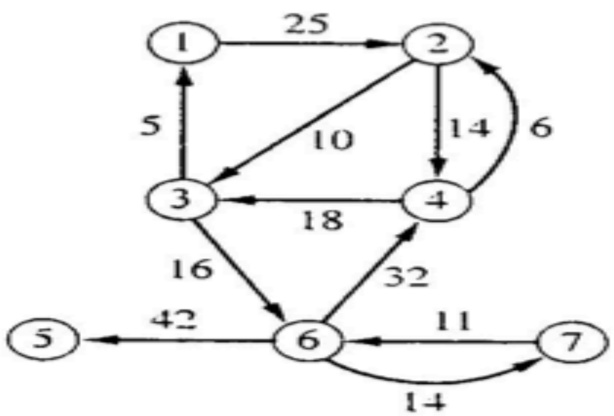


(a) A weighted digraph

$$
\begin{pmatrix}
0 & 25.0 & \infty & \infty & \infty & \infty & \infty \\
\infty & 0 & 10.0 & 14.0 & \infty & \infty & \infty \\
5.0 & \infty & 0 & \infty & \infty & 16.0 & \infty \\
\infty & 6.0 & 18.0 & 0 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty \\
\infty & \infty & \infty & 32.0 & 42.0 & 0 & 14.0 \\
\infty & \infty & \infty & \infty & \infty & 11.0 & 0
\end{pmatrix}
$$

(b) Its adjacency matrix

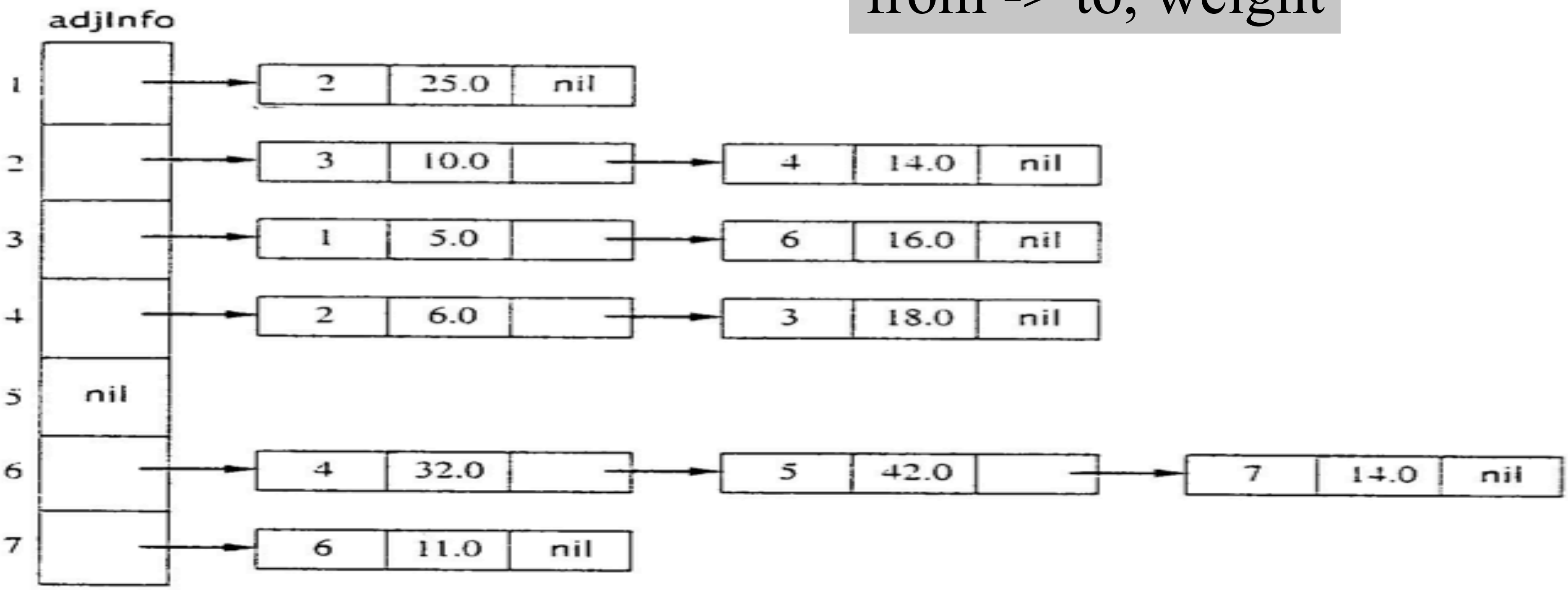
Array of Adjacency Lists Representation



(a) A weighted digraph

$$
\begin{pmatrix}
0 & 25.0 & \infty & \infty & \infty & \infty & \infty \\
\infty & 0 & 10.0 & 14.0 & \infty & \infty & \infty \\
5.0 & \infty & 0 & \infty & \infty & 16.0 & \infty \\
\infty & 6.0 & 18.0 & 0 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty \\
\infty & \infty & \infty & 32.0 & 42.0 & 0 & 14.0 \\
\infty & \infty & \infty & \infty & \infty & 11.0 & 0
\end{pmatrix}
$$

from -> to, weight



Traversing Graphs

- "Traversing" means processing each vertex edge in some organized fashion by following edges between vertices
  - We speak of *visiting* a vertex. Might do something while there.
- Recall traversal of binary trees:
  - Several strategies: In-order, pre-order, post-order
  - Traversal strategy implies an <u>order</u> of visits
  - We used recursion to describe and implement these
- Graphs can be used to model interesting, complex relationships
  - Often traversal used just to process the set of vertices or edges
  - Sometimes traversal can identify interesting properties of the graph
  - Sometimes traversal (perhaps modified, enhanced) can answer interesting questions about the problem-instance that the graph models

Traversal Strategies

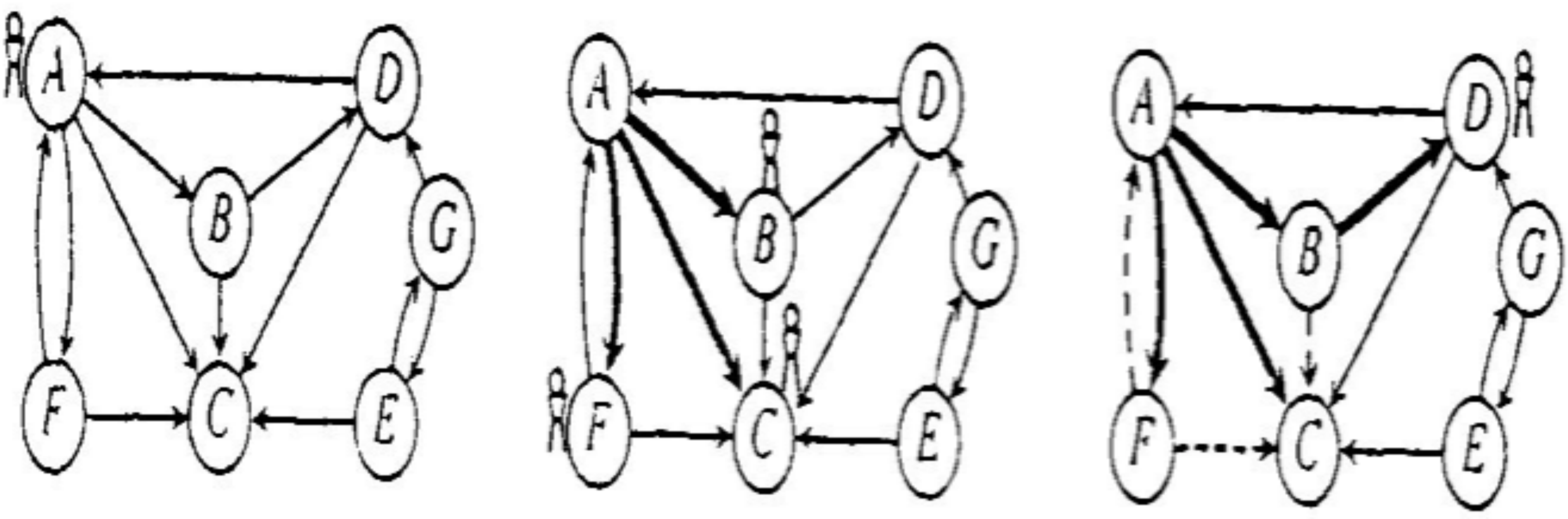
- Note: traversal algorithms start at some vertex
  - Which? Trees have a root, but graphs don't.
  - Might matter, might not.
- Breadth-first search and depth-first search
  - efficient way to "visit" each vertex and edge exactly once.
- Later we'll see exhaustive search
  - Can visit vertices and edges more than once
  - Exhaustively finds... (wait and see!)
- We'll see that BFS will tell us something about distances between a vertex and other vertices
- We'll see that DFS will be a generally useful approach for solving many graph problems.

BFS Strategy

- 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

Breath-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 $V = \{1, \ldots, n\}$; $s \in V$, 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

DFS: the Strategy in Words

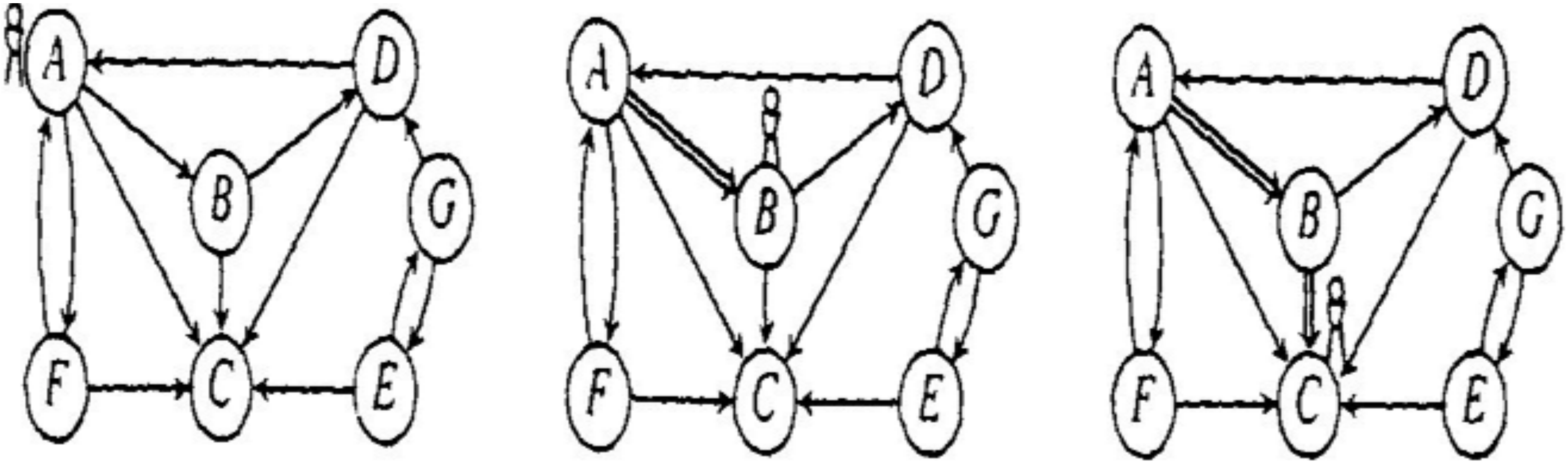
- Depth-first search: Strategy
  - Go as deep as can visiting un-visited nodes
    - Choose any un-visited vertex when you have a choice
  - When stuck at a dead-end, backtrack as little as possible
    - Back up to where you could go to another unvisited vertex
  - Then continue to go on from that point
  - Eventually you'll return to where you started
    - Reach all vertices? Maybe, maybe not

- Things are a bit different for directed vs. undirected graphs
  - It's not really that different, until you get interested in using DFS to find cycles

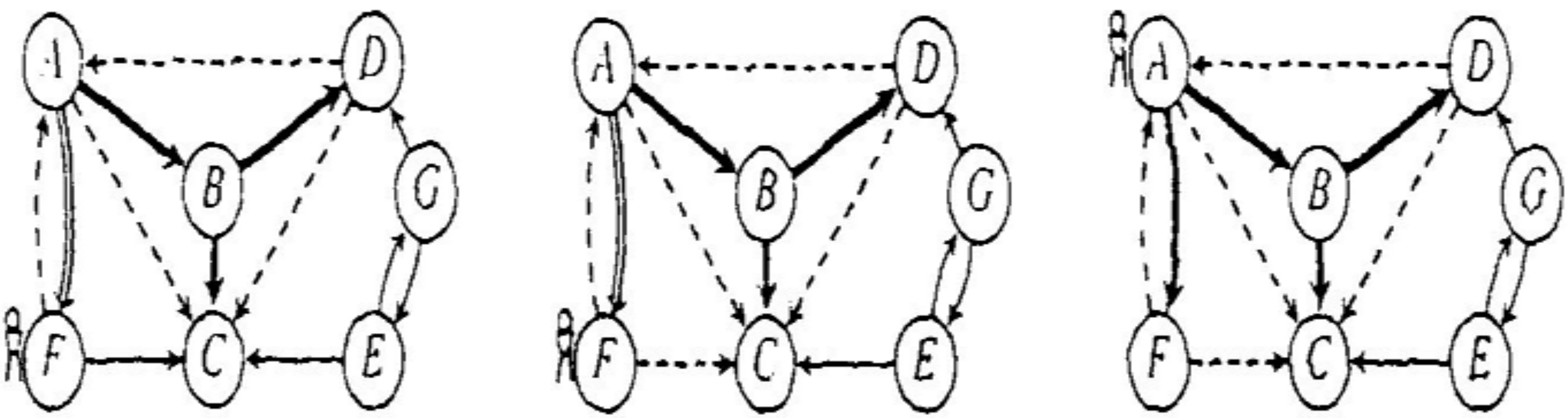
Observations about the DFS Strategy

- Note: we must keep track of what nodes we've visited
- DFS traverses a subset of E (the set of edges)
  - Creates a tree, rooted at the starting point:
    the Depth-first Search Tree (DFS tree)
  - Each node in the DFS tree has a distance from the start. (We often don't care about this, but we could.)
- At any point, all nodes are either:
  - Un-discovered
  - Finished (you backed up from it), or
  - Discovered (I.e. visited) but not finished
    - On the path from the current node back to the root
    - We might back up to it
  - (Later we'll call these states: white, black and gray)

An Example of DFS

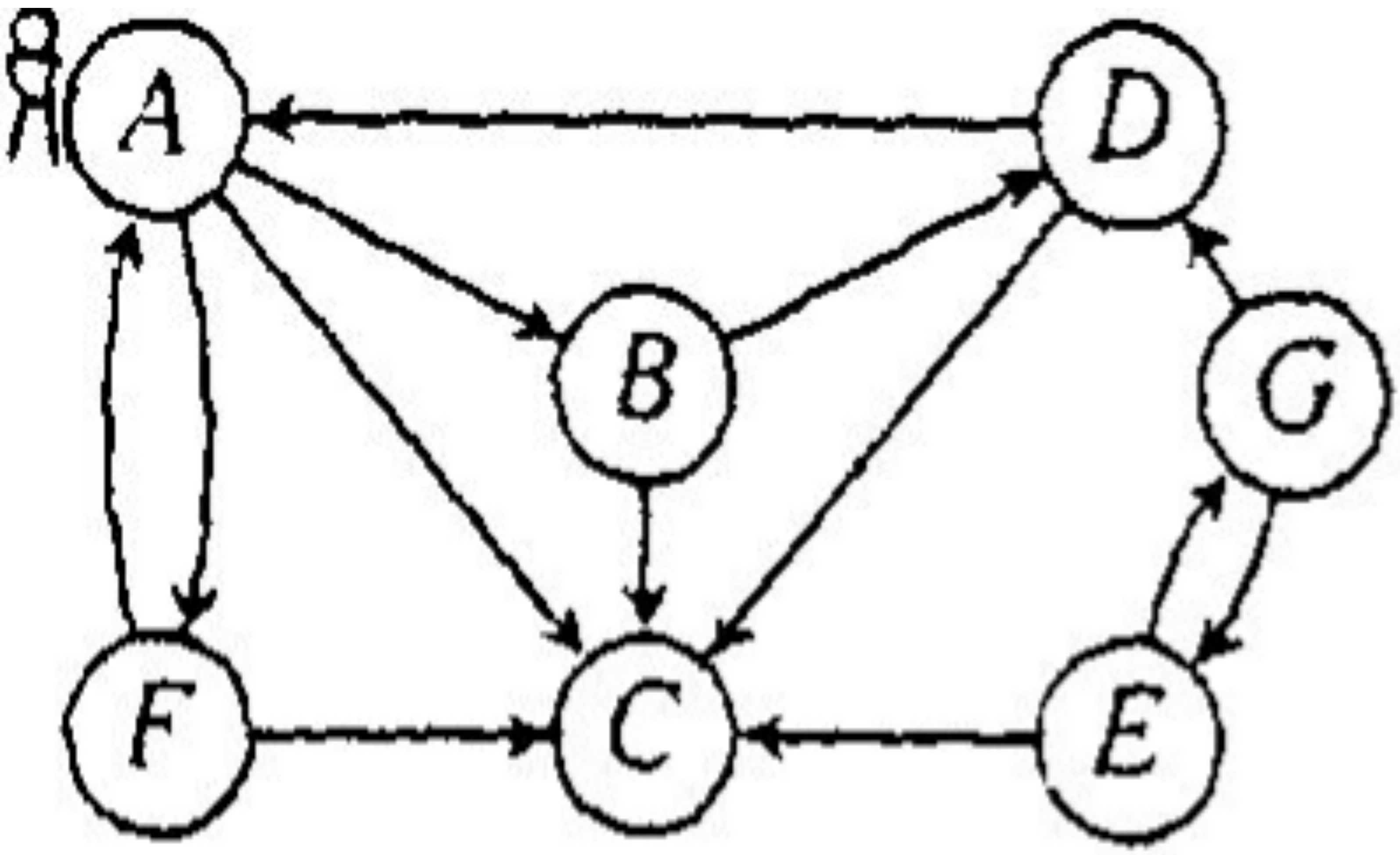


and so on…



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

- Vertex status: undiscovered, discovered, finished
- Edge status: part of DFS tree or not?



Recursive DFS visit function

```
dfs_recurs(adj,start) {
   // reached node “start”; do something?
   visit[start] = true
   trav = adj[start]
   while (trav != null) {
      v = trav.ver
      if (!visit[v])
           dfs_recurs(adj,v)
      trav = trav.next
   }
  // about to leave “start”; do something?
}
```

- Sometimes called dfs_visit().

Calling Function for DFS

- Purpose: do all required initializations, then call dfs_recurs() at a given node (just one call)

```
Input Parameters: adj,start
Output Parameters: None

dfs(adj,start)  {
  // do any initializations
   n = adj.last
   for i = 1 to n
      visit[i] = false

   // one call to recursive function at start
   dfs_recurs(adj,start)
}
```

DFS to Process all Vertices in a Graph

- Purpose: do all required initializations, then call dfs_recurs() as many times as needed to visit all nodes. May create a DFS forest.

```
dfs_sweep(adj)  {
   n = adj.last
  // do any initializations
   for i = 1 to n
      visit[i] = false

   // loop called on any unvisited node
  for i = 1 to n
      if (!visit[i]) dfs_recurs(adj, i)
}
```

Notes on dfs_recurs() function

- Often called “dfsVisit” (or something like that)
- Creates one DFS tree from a given start node
  - Must be called by some caller function
  - May not visit all nodes in a the graph G
- Assumes that all nodes have been initialized as “undiscovered”
- Sometimes an “else” clause that does something to nodes not visited (or edges to those)

General Skeleton Similar to DFS_recurs (Cormen)

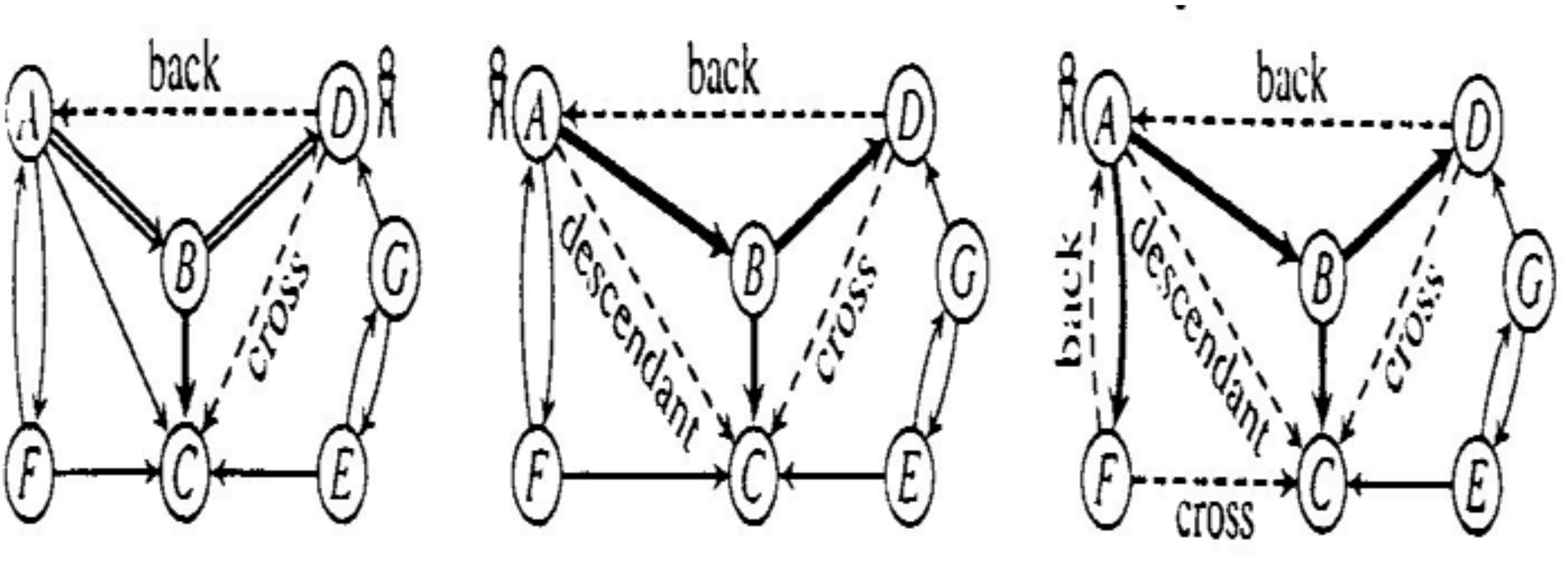
```
int  dfs(IntList[] adjVertices, int[] color, int v, . . .)
     int  w;
     IntList  remAdj;
     int  ans;
     color[v] = gray;
     Preorder processing of vertex v
     remAdj = adjVertices[v];
     while (remAdj ≠ nil)
          w = first(remAdj);
          if (color[w] == white)
               Exploratory processing for tree edge vw
               int  wAns = dfs(adjVertices, color, w, . . .);
               Backtrack processing for tree edge vw. using wAns (like inorder)
          else
               Checking (i.e., processing) for nontree edge vw
          remAdj = rest(remAdj)
     Postorder processing of vertex v. including final computation of ans
     color[v] = black;
     return ans;
```

Using DFS to Find if a Graphic is Acyclic

- Does a graph have a cycle?
  - DFS is great for this
  - But, slightly harder if graph is undirected
- Use DFS tree: classify edges and nodes as you process them
  - Nodes:
    - White: unvisited
    - Black: done with it, backed up from it (never to return)
    - Gray: Have reached it; exploring it's adjacent nodes; but not done with it
  - Also, have a "time counter", say, ctr
    - Set d[v] = ctr++ as discovery time
    - Set f[v] = ctr++ as finish time

Depth-first search tree

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



Using Non-Tree Edges to Identify Cycles

- From the previous graph, note that:
- Back edges (indicates a cycle)
  - dfs_recurs() sees a vertex that is gray
  - This back edge goes back up the DFS tree to a vertex that is on the path from the current node to the root
- Cross Edges and Descendant Edges (not cycles)
  - dfs_recurs() sees a vertex that is black
  - Descendant edge: connects current node to a descendant in the DFS tree
  - Cross edge: connects current node to a node in another subtree – not a descendant of current node

Non-tree Edges in DFS

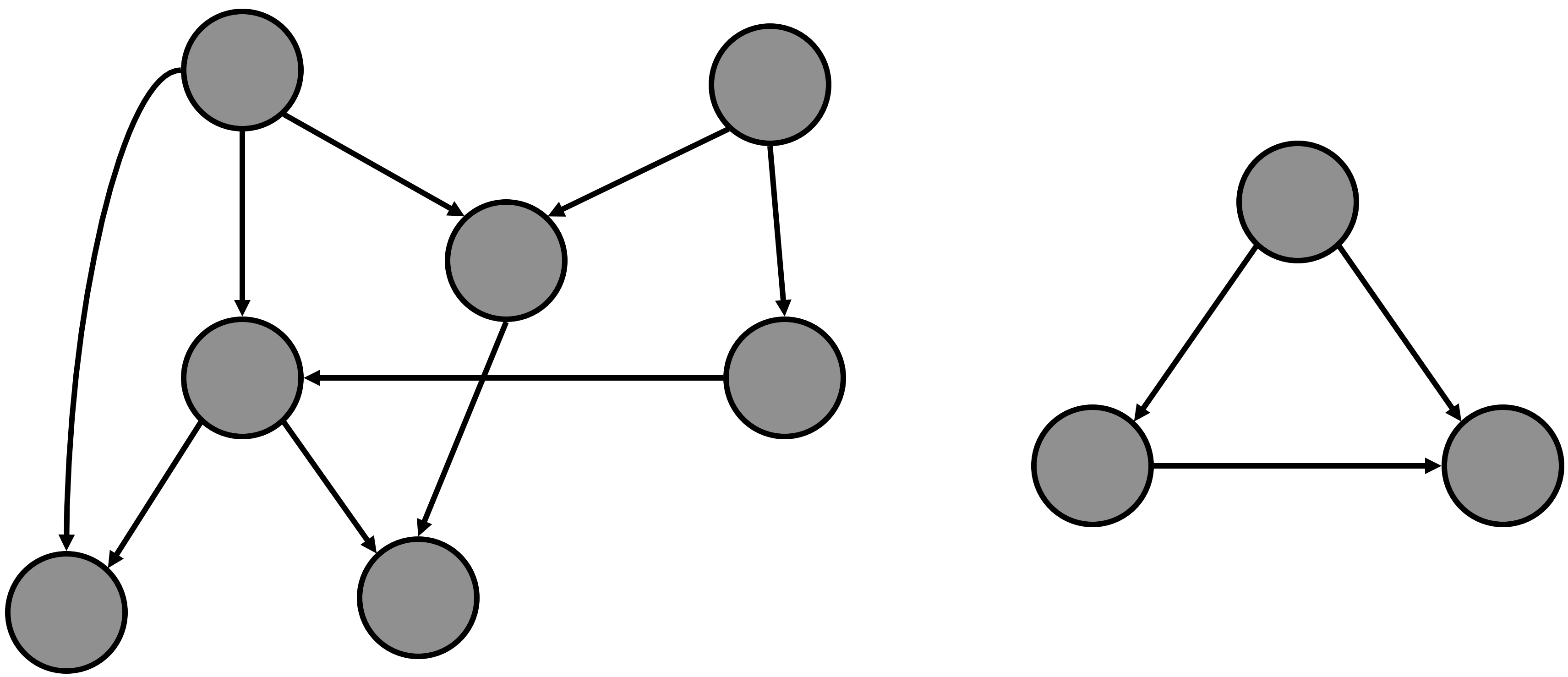
- Question 1: Finding back edges for an undirected tree is not **quite** this simple:
  - The parent node of the current node is gray
  - Not a cycle, is it? It's the same edge you just traversed
  - Question: how would you modify our code to recognize this?
- Question 2:
  - How could you modify the code to distinguish cross edges from descendant edges?
  - Hint: use discovery and finish times

Time Complexity of DFS

- For a digraph having n vertices and m edges
  - Each edge is processed once in the while loop of dfs_recurs() for a cost of $\theta(m)$
    - Think about adjacency list data structure.
    - Traverse each list exactly once. (Never back up)
    - There are a total of 2m nodes in all the lists
  - The dfs_sweep() algorithm will do $\theta(n)$ work even if there are no edges in the graph
  - Thus over all time-complexity is $\theta(n+m)$
    - Remember: this means the larger of the two values
    - Note: This is considered "linear" for graphs since there are two size parameters for graphs.
  - Extra space is used for color array.
    Space complexity is $\theta(n)$

Directed Acyclic Graphs

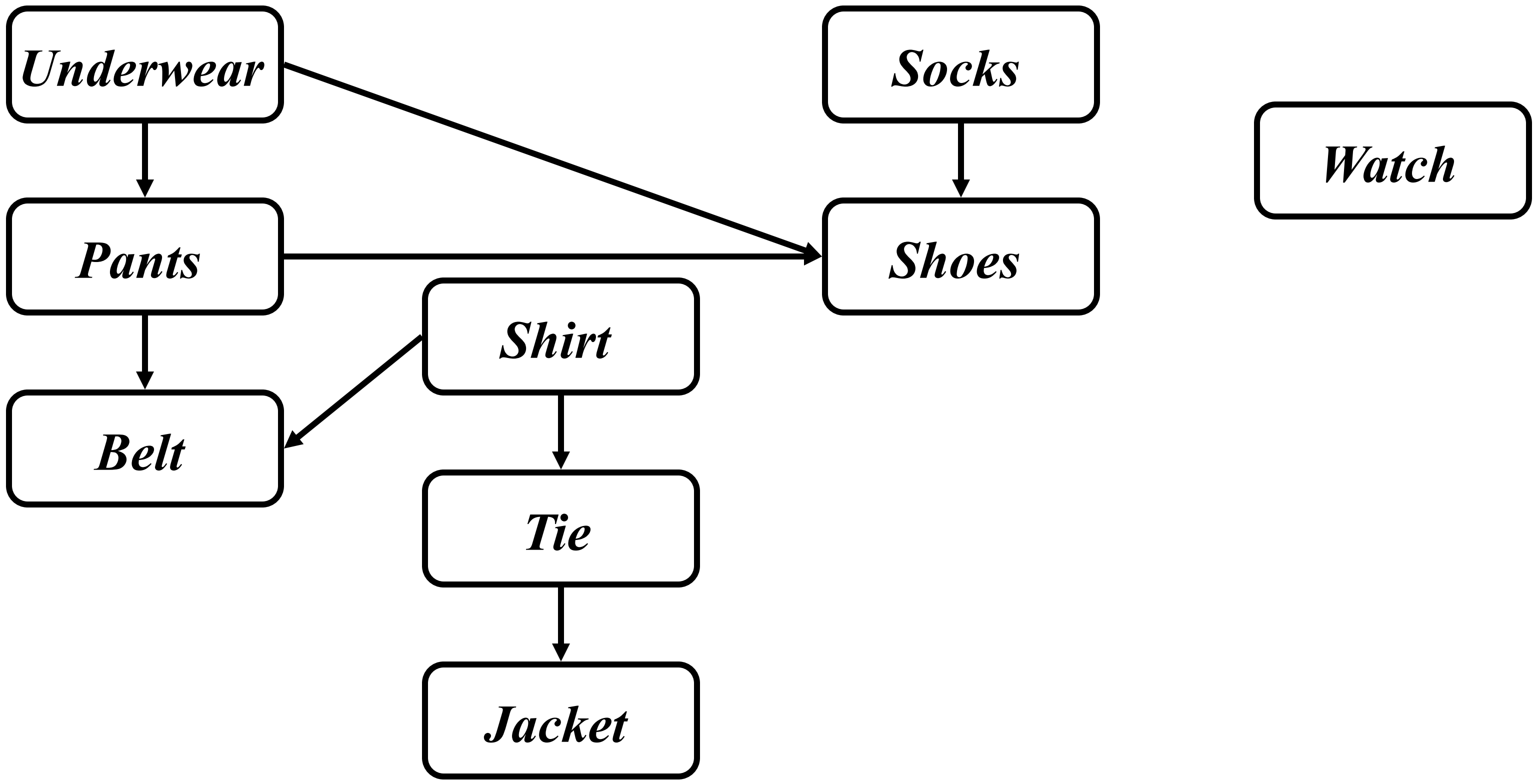
- A *directed acyclic graph* or *DAG* is a directed graph with no directed cycles:



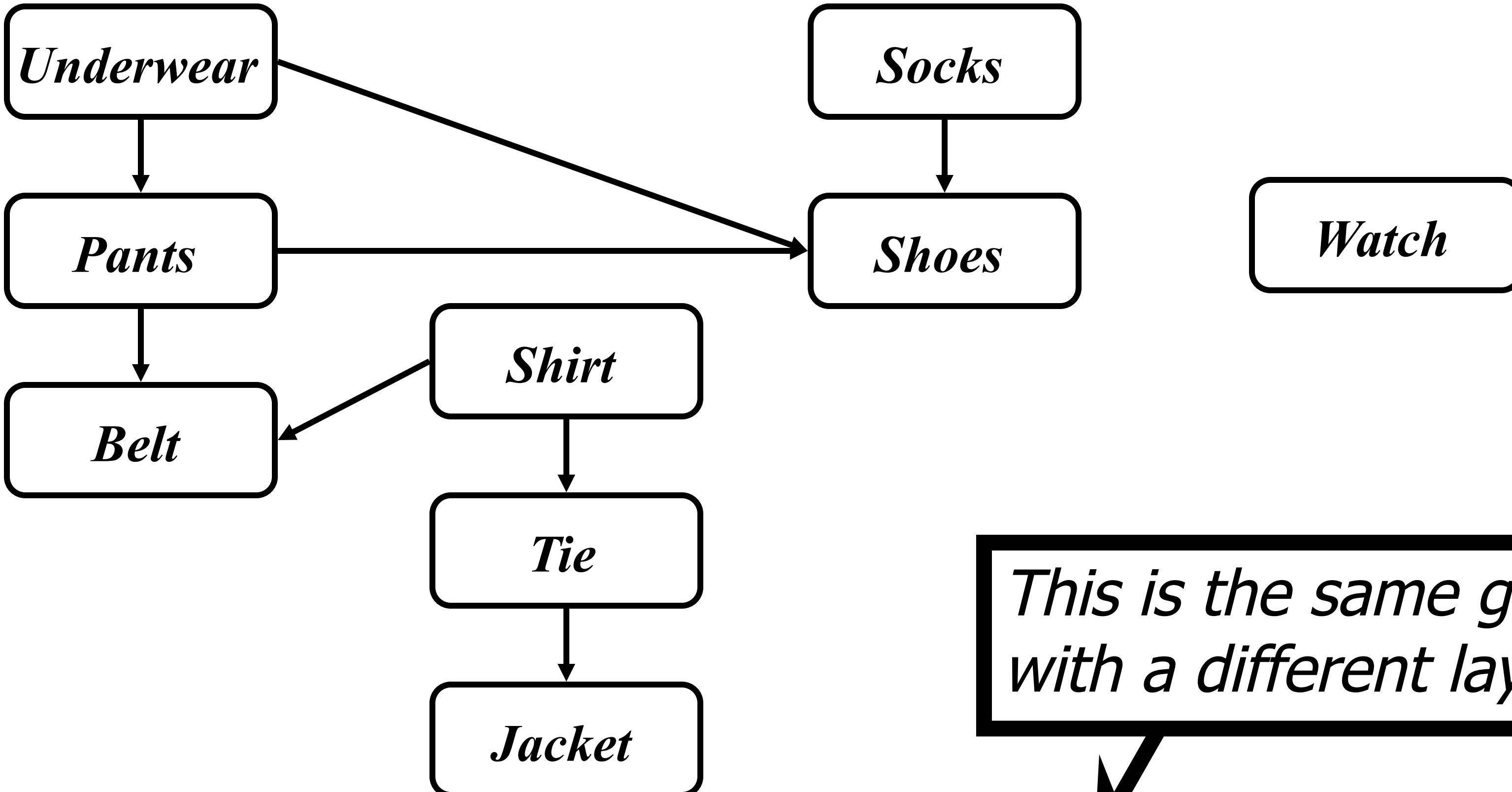
Topological Sort

- *Topological sort* of a DAG:
  - Linear ordering of all vertices in graph G such that vertex *u* comes before vertex *v* if edge $(u, v) \in G$
- Real-world example: getting dressed

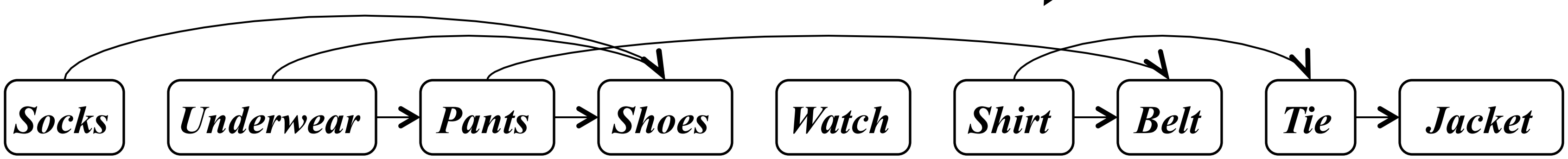
Getting Dressed



Getting Dressed



This is the same graph with a different layout.



Topological Sort Algorithm

```
Topological-Sort()
{
   Run DFS_recurs()
   When a vertex is finished, output it
   Vertices are output in reverse
     topological order
        (or add to stack/list)
}
```

- Can stack/store vertices as found to store them in topologically sorted order
- Time: O(V+E)

Topologoical Sort, Recursive Function

```
top_sort_recurs(adj, start, ts) {
        visit[start] = true
        trav = adj[start]
        while (trav != null) {
                v = trav.ver
                if (!visit[v])
                     top_sort_recurs(adj,v,ts)
                trav = trav.next
        }
        ts[k] = start
        k = k – 1
}
```

Topological Sort: Driver

```
top_sort(adj, ts) {
    n = adj.last
    // k is the index in ts where the next vertex is to be
    // stored in topological sort. k is assumed global.
    k = n
    for i = 1 to n
        visit[i] = false
    for i = 1 to n
        if (!visit[v])
          top_sort_recurs(adj, i, ts)
    }
}
```

Forward vs. Reverse

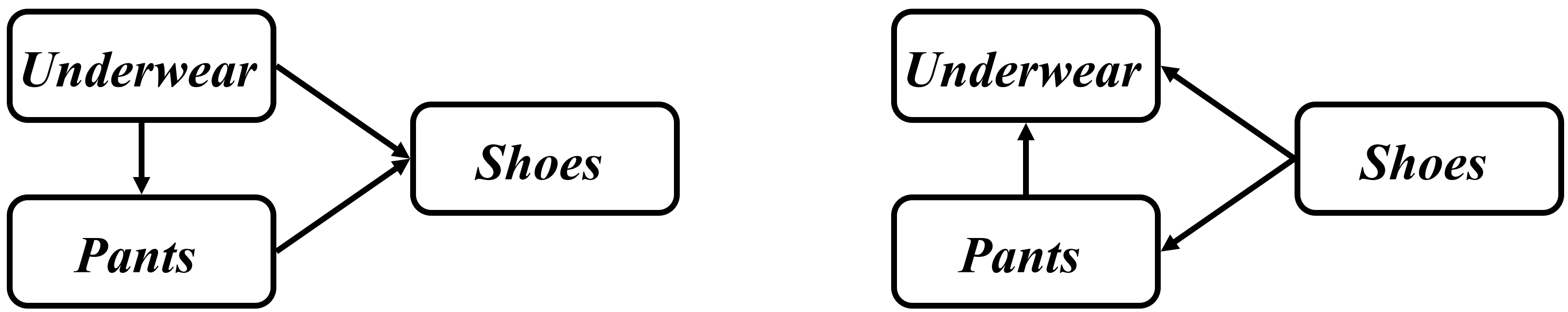
- Topological sort is a type of sort
  - Implies an ordering
  - Can sort backwards, of course

- Forward topological order
  - If edge **vw** in graph, then topo[**v**] < topo[**w**]
- Reverse topological order
  - If edge **vw** in graph, then topo[**v**] > topo[**w**]

- And, every directed graph has a transpose, which means… (see next slide)

What's an Edge Mean?

- What's our graph model?
  - Edge **uv** means do **u** first, then **v**. Or, …
  - Edge **uv** means task **u** depends on v (I.e. **v** must be done first)



- The latter called a dependency graph
- "forward in time" vs. "depend on this one"

- Big deal? No, we can order vertices in reverse topological order if needed

Sort this!