**In the name of God**

# Network Flows

# 2. Search Algorithms

## 2.1 Algorithms

**Fall 2010**

*Instructor: Dr. Masoud Yaghini*

# Outline

- Introduction
- Breadth-First Search
- Depth-First Search
- Reverse Search Algorithm
- Determining Strong Connectivity
- Topological Ordering

# Introduction

# Search Algorithms

- ***Search algorithms***
  - fundamental graph techniques that attempt to find all the nodes in a network
  - Different variants of search algorithms lie at the heart of many ***maximum flow*** and ***minimum cost flow*** algorithms.

# Search Algorithms

- **The applications of search algorithms:**
  - to find all nodes in a network that are reachable by directed paths from a specific node,
  - to find all the nodes in a network that can reach a specific node t along directed paths,
  - To identify all connected components of a network
  - to identify a directed cycle in a network
  - to reorder the nodes 1, 2, ... , $n$ so that for each arc $(i, j) \in A$, $i < j$ (***topological ordering***), if the network is ***acyclic***

# Search Algorithms

- We wish to find all the nodes in a network $G = (N, A)$ that are reachable along directed paths from a distinguished node $s$, called the ***source***.
- A search algorithm fans out from the source and identifies an increasing number of nodes that are reachable from the source.
- ***Marked*** *vs*. ***unmarked node***
  - At every intermediate point in its execution, the search algorithm designates all the nodes in the network as being in one of the two states: ***marked*** or ***unmarked***.
  - The ***marked nodes*** are known to be reachable from the source, and the status of ***unmarked nodes*** has yet to be determined.

# Search Algorithms

- If node $i$ is marked, node $j$ is unmarked, and the network contains the arc $(i, j)$, we can mark node $j$
- It is reachable from ***source*** via a directed path to node $i$ plus arc $(i, j)$.
- ***Admissible*** *vs.* ***inadmissible arc***
  – Let us refer to arc $(i, j)$ as ***admissible arc*** if node $i$ is marked and node $j$ is unmarked,
  – and refer to it as ***inadmissible arc*** otherwise.
- Initially, we mark only the source node.
- Subsequently, by examining admissible arcs, the search algorithm will mark additional nodes.

# Search Algorithms

- ***Predecessor node***
  – Whenever the procedure marks a new node $j$ by examining an admissible arc $(i, j)$, we say that node $i$ is a ***predecessor*** of node $j$ [i.e., $\boldsymbol{pred(j) = i}$].
- The algorithm terminates when the network contains no admissible arcs.
- The search algorithm traverses the marked nodes in a certain order.
- We record this traversal order in an array ***order***:
  – the entry $\boldsymbol{order(i)}$ is the order of node $i$ in the traversal.

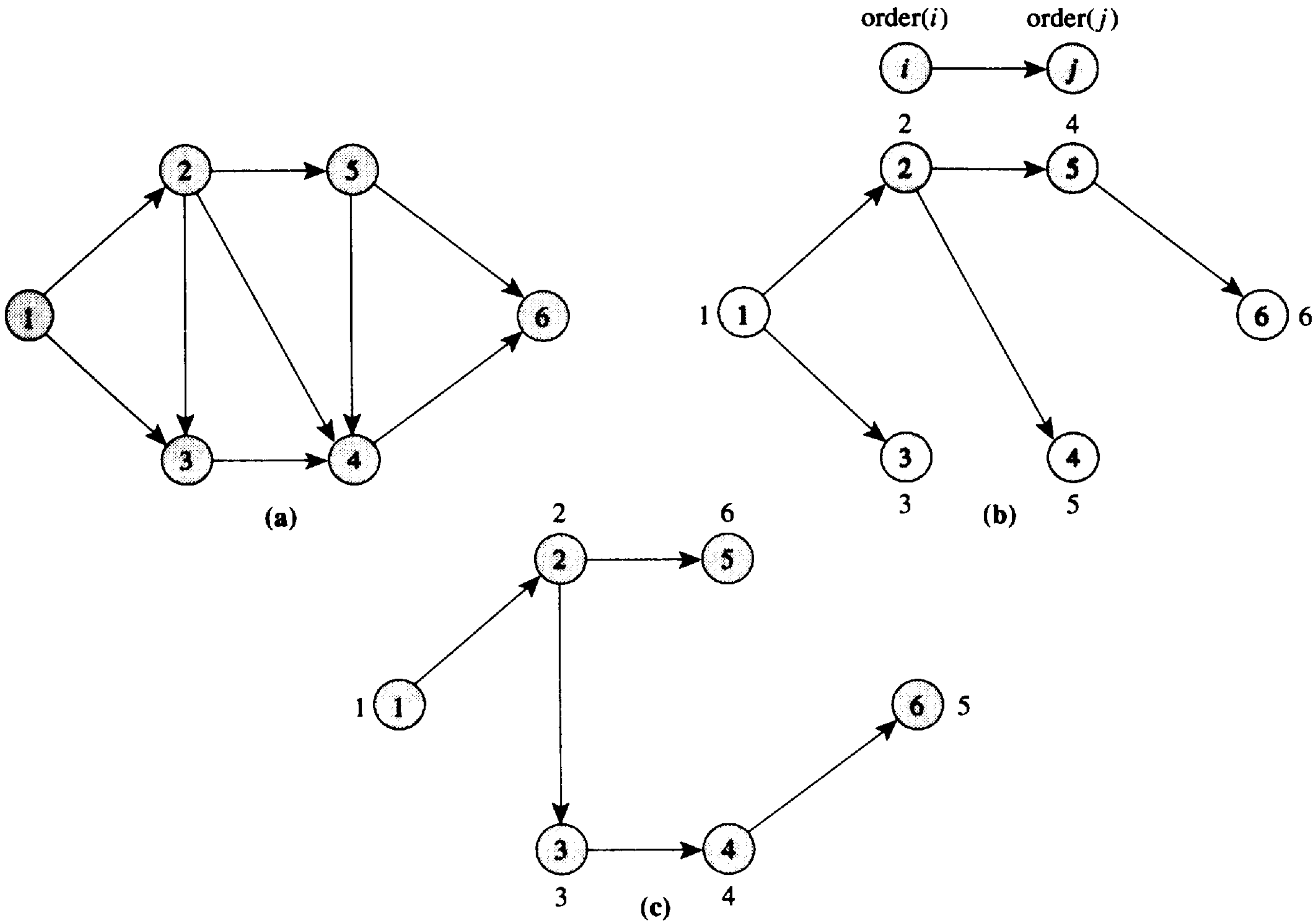
# Search Algorithms

```
algorithm search;
begin
    unmark all nodes in N;
    mark node s;
    pred(s) : = 0;
    next : = 1;
    order(s): = next;
    LIST : = {s}
    while LIST ≠ Ø do
    begin
        select a node i in LIST;
        if node i is incident to an admissible arc (i, j) then
        begin
            mark node j;
            pred( j) : = i;
            next : = next + 1;
            order( j) : = next;
            add node j to LIST;
        end
        else delete node i from LIST;
    end;
end;
```

# Search Algorithms

- **LIST** :
  - represents the set of marked nodes that the algorithm has yet to examine in the sense that some admissible arcs might emanate from them.
- When the algorithm terminates, it has marked all the nodes in $G$ that are reachable from $s$ via a directed path.
- ***Search tree***
  - The predecessor indices define a tree consisting of marked nodes.
  - We call this tree a ***search tree***.

# Search Algorithms




- (b) and (c), depict two search trees for the network shown in (a)

# Search Algorithms

- It is easy to show that the search algorithm runs in $O(m)$ time.
- Each iteration of the while loop either finds an admissible arc or does not.
  - If the loop finds an admissible arc, the algorithm marks a new node and adds it to LIST,
  - If the loop does not find an admissible arc it deletes a marked node from LIST.
- Since the algorithm marks any node at most once, it executes the while loop at most $2n$ times.

# Search Algorithms

- Now consider the effort spent in identifying the admissible arcs.
- For each node $i$, we scan the arcs in $A(i)$ at most once.
- Therefore, the search algorithm examines a total of

$$\sum_{i \in N} |A(i)| = m$$

- arcs, and thus terminates in $O(m)$ time.

# Search Algorithms

- The algorithm, as described, does not specify the manner for examining the nodes or for adding the nodes to **LIST**.
- Different rules give rise to different search techniques.
- Two data structures have proven to be the most popular for maintaining LIST ***a queue*** and ***a stack*** and they give rise to two fundamental search strategies:
  - ***Breadth-first search***
  - ***Depth-first search***

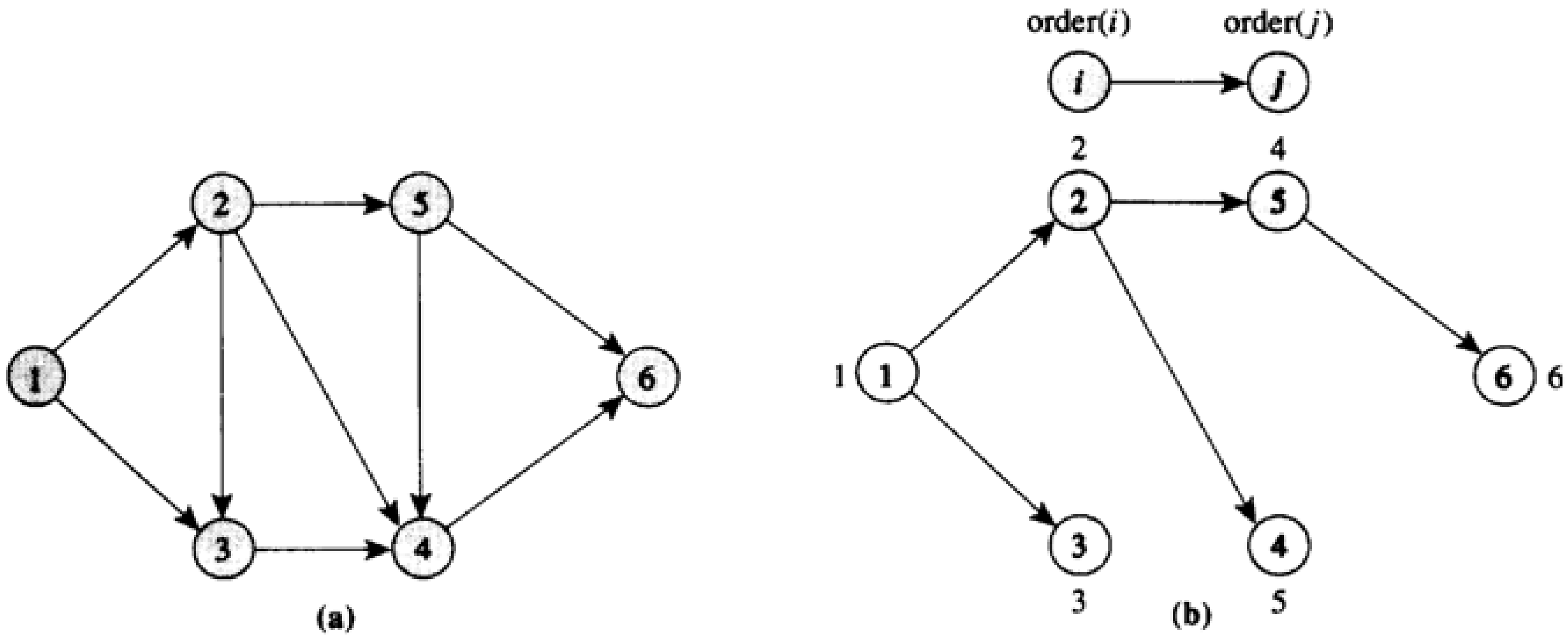
# Breadth-First Search

# Breadth-First Search

- If we maintain the set **LIST** as a ***queue***, we always select nodes from the front of LIST and add them to the rear.
- In this case the search algorithm selects the marked nodes in a first-in, first-out order.
- If we define the distance of a node $i$ as the minimum number of arcs in a directed path from node $s$ to node $i$, this kind of search first marks nodes with distance 1, then those with distance 2, and so on.
- Therefore, this version of search is called a ***breadth-first search*** and the resulting search tree is a breadth-first search tree.

# Breadth-First Search

- *The breadth-first search tree*



# Breadth-First Search

- Property
  - *In the breadth-first search tree, the tree path from the source node s to any node i is a shortest path*
  - *i.e., contains the fewest number of arcs among all paths joining these two nodes*

- *<Breadth-First Search Animation>*

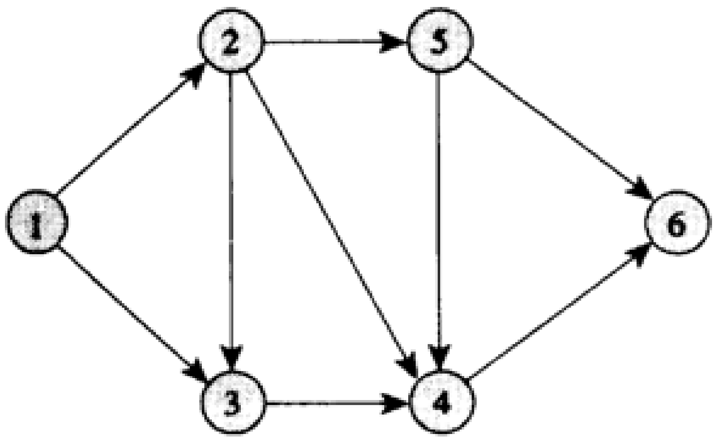
# Depth-First Search

# Depth-First Search

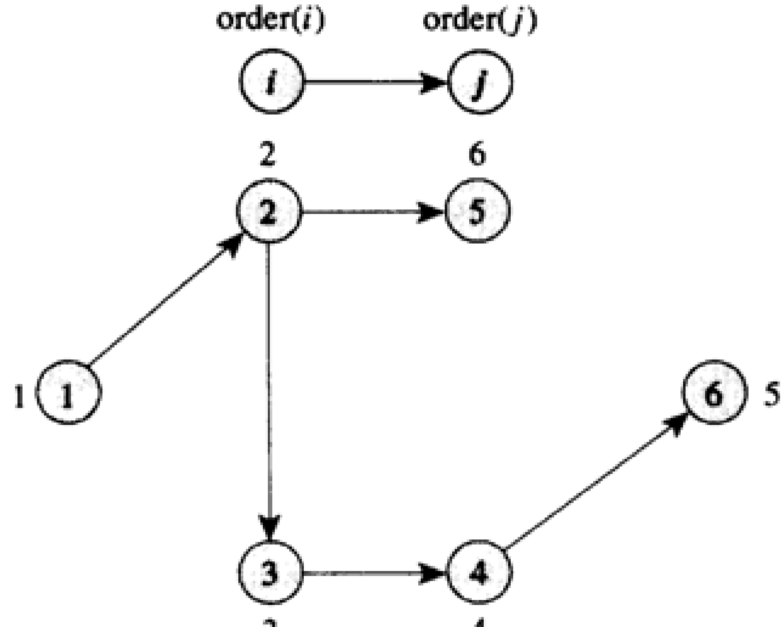
- If we maintain the set LIST as a ***stack***, we always select the nodes from the front of LIST and also add them to the front.
- In this case the search algorithm selects the marked node in a ***last-in, first-out order***.
- This algorithm performs a deep probe, creating a path as long as possible, and backs up one node to initiate a new probe when it can mark no new node from the tip of the path.
- We call this version of search a ***depth-first search***
- The depth-first traversal of a network is also called its ***preorder traversal***.

# Depth-First Search

- *A depth-first search tree*




(a)




(c)

# Depth-First Search

- Property
  - (a) If node $j$ is a descendant of node $i$, then $order(j) > order(i)$.
  - (b) All the descendants of any node are ordered consecutively in sequence.

- ***<Depth-First Search Animation>***

# Reverse Search Algorithm

# Reverse Search Algorithm

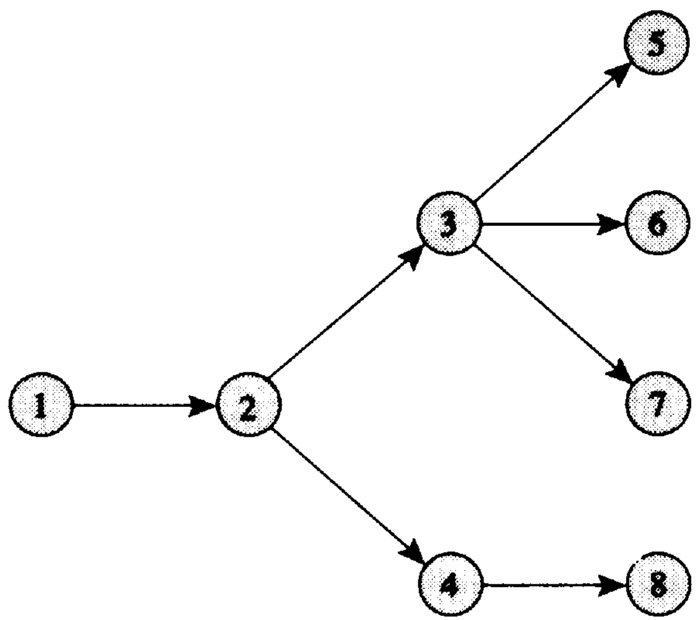
- The algorithm were described allows us to identify all the nodes in a network that are reachable from a given node $s$ by directed paths.
- Suppose that we wish to identify all the nodes in a network from which we can reach a given node $t$ along directed paths.

# Reverse Search Algorithm

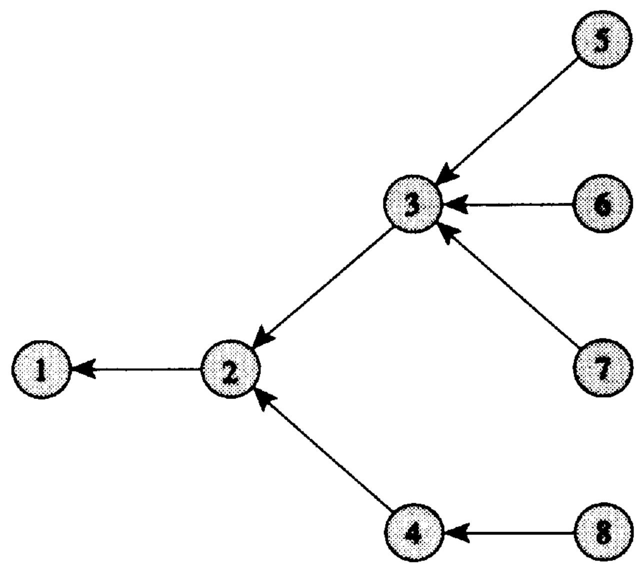
- We can solve this problem by using the algorithm we have just described with three slight changes:
  - (1) we initialize LIST as LIST = {t};
  - (2) while examining a node, we scan the **incoming arcs** of the node instead of its **outgoing arcs**; and
  - (3) we designate an arc $(i, j)$ as admissible if $i$ is unmarked and $j$ is marked.
- We subsequently refer to this algorithm as a ***reverse search algorithm***.

# Reverse Search Algorithm

- Whereas the (forward) search algorithm gives us a ***directed out-tree rooted*** at node *s*, the reverse search algorithm gives us ***a directed in-tree rooted*** at node *t*.




***directed out-tree rooted***




***directed in-tree rooted***

# Determining Strong Connectivity

# Determining Strong Connectivity

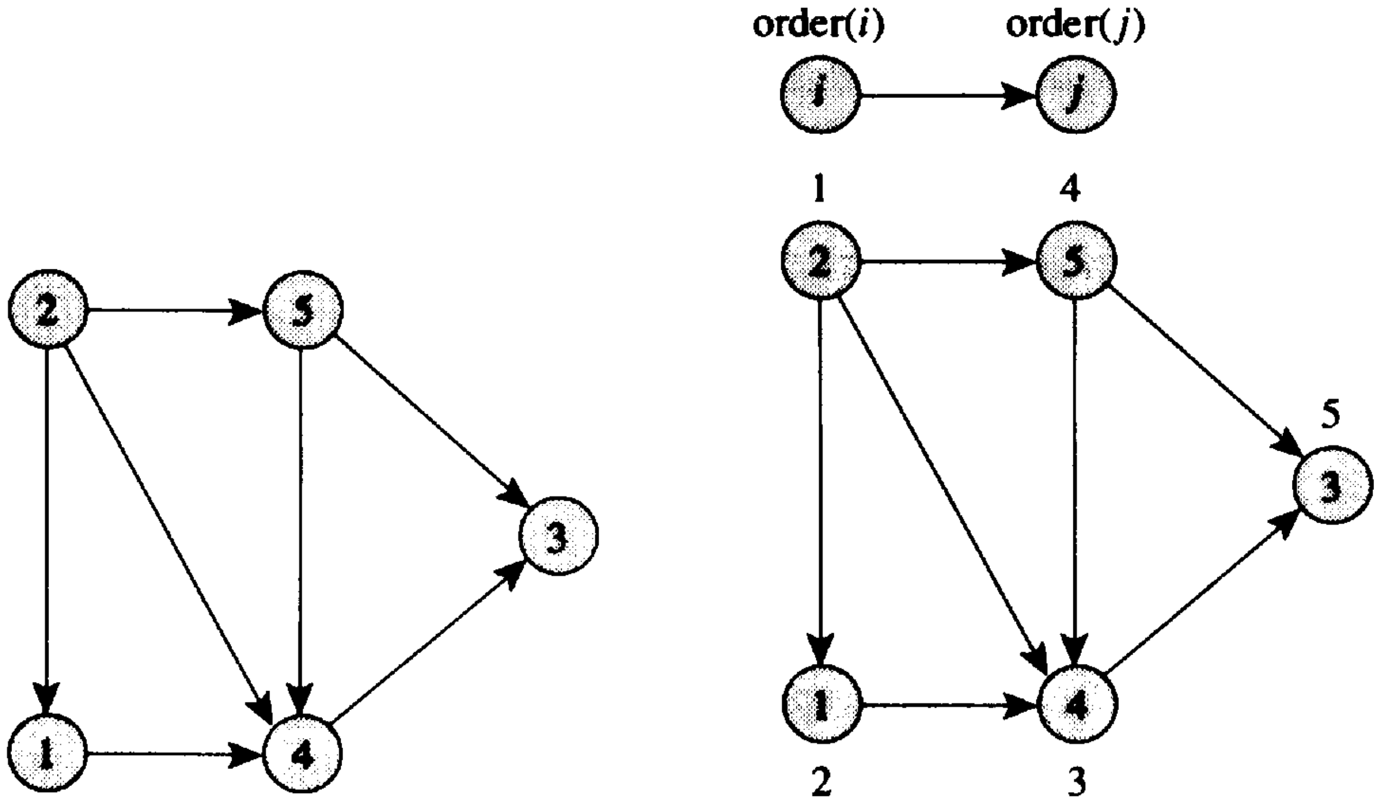
- A network is strongly connected if for every pair of nodes $i$ and $j$, the network contains a directed path from node $i$ to node $j$.
- This definition implies that a network is strongly connected if and only if for any arbitrary node $s$, every node in $G$ is reachable from $s$ along a directed path
- Conversely, node $s$ is reachable from every other node in $G$ along a directed path.
- Clearly, we can determine the strong connectivity of a network by two applications of the search algorithm, once applying the (forward) search algorithm and then the reverse search algorithm.

# Topological Ordering

# Topological Ordering

- ***Node labeling***
  - Let us label the nodes of a network $G = (N, A)$ by distinct numbers from 1 through $n$ and represent the labeling by an array order [i.e., $order(i)$ gives the label of node $i$].
- ***Topological ordering***
  - We say that this labeling is a ***topological ordering*** of nodes if every arc joins ***a lower-labeled node to a higher-labeled*** node.
  - That is, for every arc $(i, j) \in A$, $order(i) < order(j)$.

# Topological Ordering

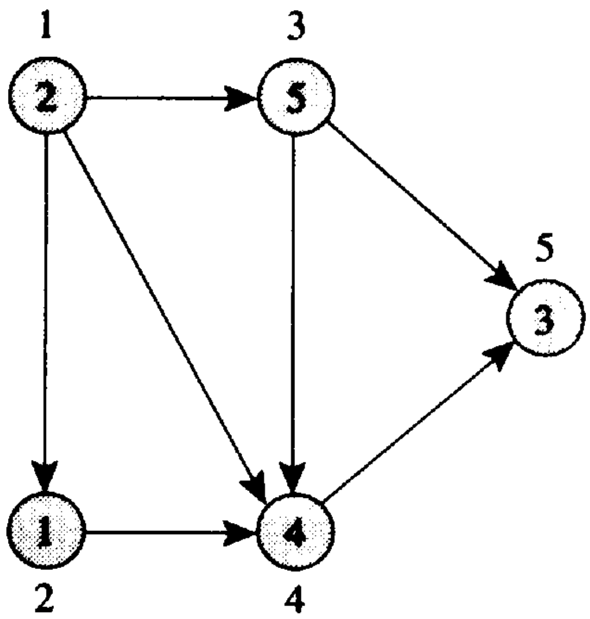



(a)

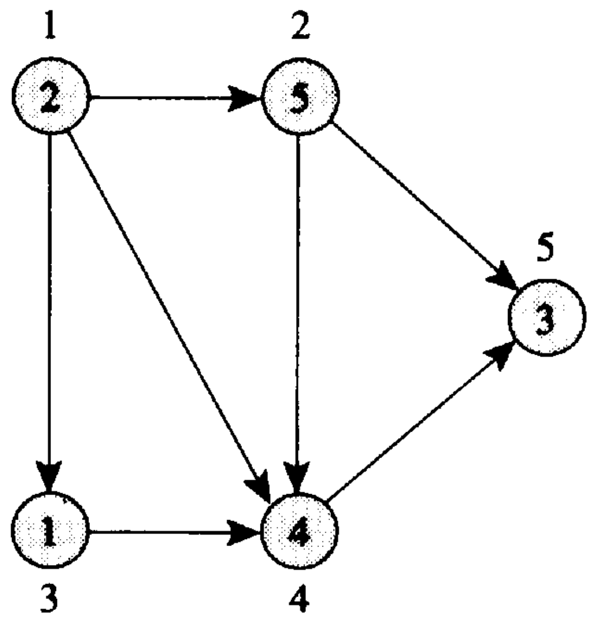
(b)

- (a) a sample network, (b) the labeling shown is not a topological ordering because (5, 4) is an arc and *order*(5) > *order*(4).

# Topological Ordering




(c)

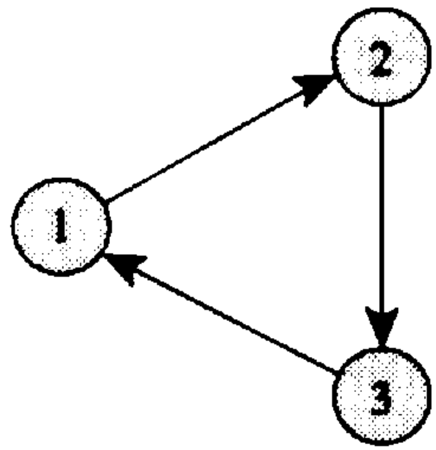



(d)

- (c) and (d) : the labelings shown are topological orderings.
- A network might have several topological orderings.

# Topological Ordering

- A network that contains a directed cycle has no topological ordering




  - This network is cyclic because it contains a directed cycle and for any directed cycle $W$
  - we can never satisfy the condition $order(i) < order(j)$ for each $(i, j) \in W$.

# Topological Ordering

- we shall show next that a network that does not contain any negative cycle can be topologically ordered.
- This observation shows that a network is acyclic if and only if it possesses a topological ordering of its nodes.

# Topological Ordering

- By using a search algorithm, we can either detect the presence of a directed cycle or produce a topological ordering of the nodes.
- The algorithm is fairly easy to describe.

# Topological Ordering

- In the network $G$,
  - select any node of zero indegree.
  - Give it a label of 1, and then delete it and all the arcs emanating from it.
  - In the remaining subnetwork select any node of zero indegree, give it a label of 2, and then delete it and all arcs emanating from it.
  - Repeat this process until no node has a zero indegree.
- At this point,
  - if the remaining subnetwork contains some nodes and arcs, the network $G$ contains a directed cycle.
  - Otherwise, the network is acyclic and this labeling gives a topological ordering of nodes.

# Topological Ordering

```
algorithm topological ordering;
begin
    for all i ∈ N do indegree(i) : = 0;
    for all (i, j) ∈ A do indegree( j) : = indegree( j) + 1;
    LIST : = Ø;
    next : = 0;
    for all i ∈ N do
        if indegree(i) = 0 then LIST : = LIST ∪ {i};
    while LIST ≠ Ø do
    begin
        select a node i from LIST and delete it;
        next : = next + 1;
        order(i) : = next;
        for all (i, j) ∈ A(i) do
        begin
            indegree( j) : = indegree( j) - 1;
            if indegree( j) = 0 then LIST : = LIST ∪ { j};
        end;
    end;
    if next < n then the network contains a directed cycle
    else the network is acyclic and the array order gives a topological order of nodes;
end;
```

# Topological Ordering

- This algorithm
  - first computes the indegrees of all nodes and forms a set **LIST** consisting of all nodes with zero indegrees.
  - At every iteration we select a node $i$ from **LIST**, for every arc $(i, j) \in A(i)$ we reduce the indegree of node $j$ by 1 unit, and if indegree of node $j$ becomes zero, we add node $j$ to the set **LIST**.
  - Observe that deleting the arc $(i, j)$ from the network is quivalent to decreasing the indegree of node $j$ by 1 unit.
- Since the algorithm examines each node and each arc of the network $O(1)$ times, it runs in $O(m)$ time.
- ***<Topological Ordering Animation>***

The End