Algorithms Design
Chap03-Graphs

College of Computer Science

Nankai University

Tianjin, P.R.China

Chap03-Graphs Outline

3.1 Basic Definitions and Applications

3.2 Graph Traversal and Graph Connectivity

3.3 Testing Bipartiteness

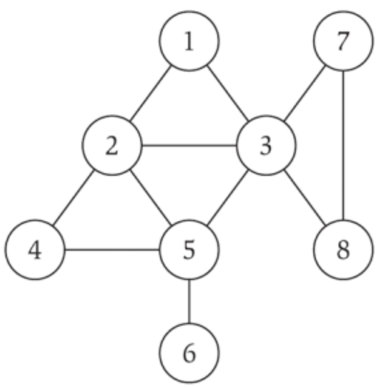
3.4 Connectivity in Directed Graphs

3.5 DAGs and Topological Ordering

3.1 Basic Definitions and Applications

Undirected graph. $G = (V, E)$

- V = nodes.
- E = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|$, $m = |E|$.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

m = 11

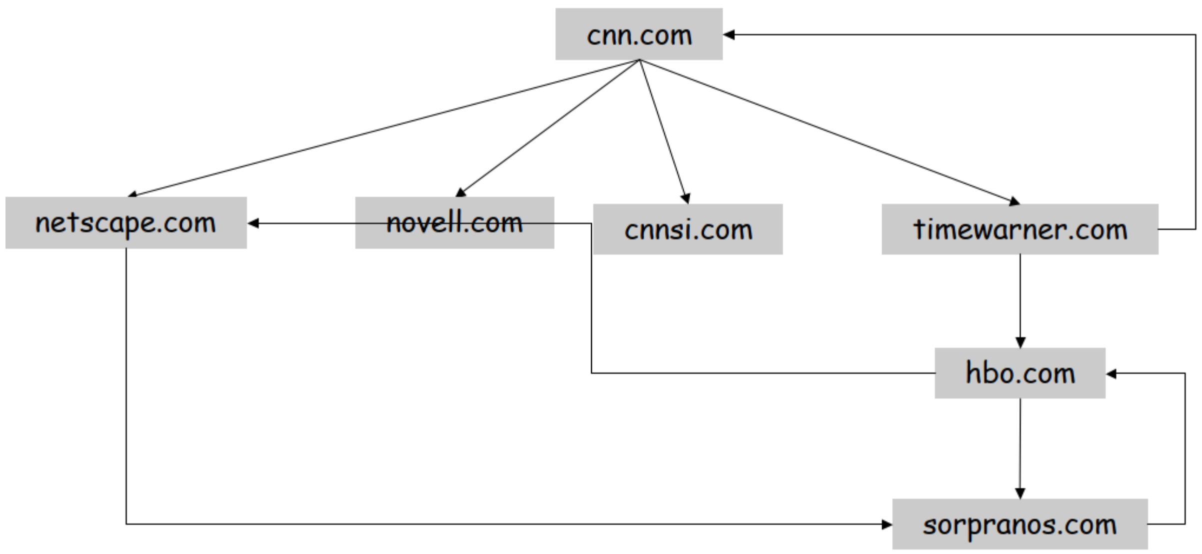
3.1 Basic Definitions and Applications

Graph	Nodes	Edges
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

3.1 Basic Definitions and Applications

Application 1: Web graph

- Node: web page
- Edge: hyperlink from one page to another.



3.1 Basic Definitions and Applications

Application 2: Social network graph

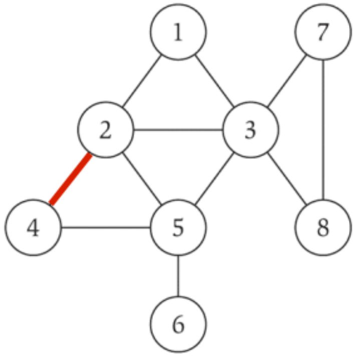
- Node: users
- Edge: relationship between two users

3.1 Basic Definitions and Applications

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.

- Two representations of each edge.
- Space proportional to n^2.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

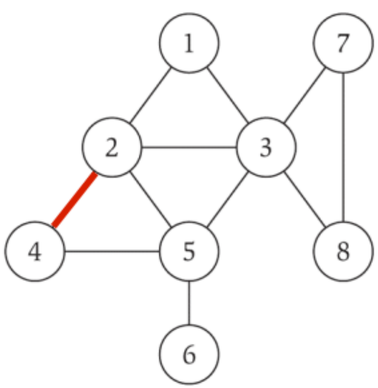
3.1 Basic Definitions and Applications

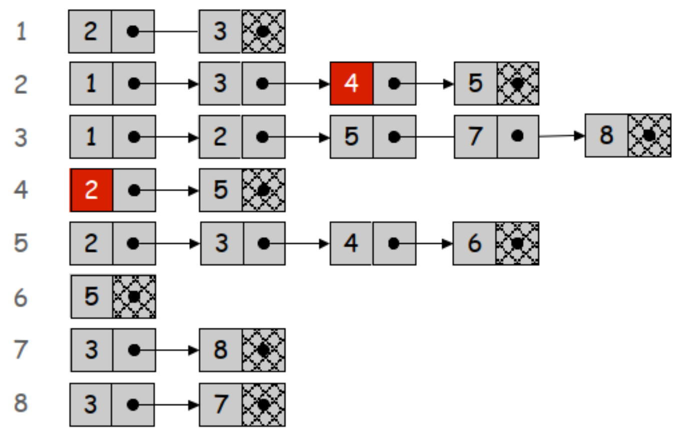
Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if (u, v) is an edge takes $O(\deg(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.

degree=# neighbors of u



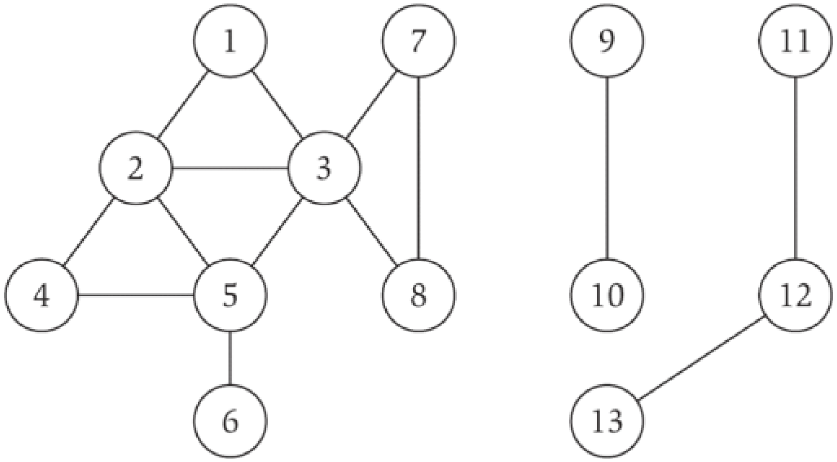


3.1 Basic Definitions and Applications

Def(3-1). A path in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \cdots, v_{k-1}, v_k$ with the property that each consecutive pair (v_i, v_{i+1}) is joined by an edge in E.

Def(3-2). A path is simple if all nodes are distinct.

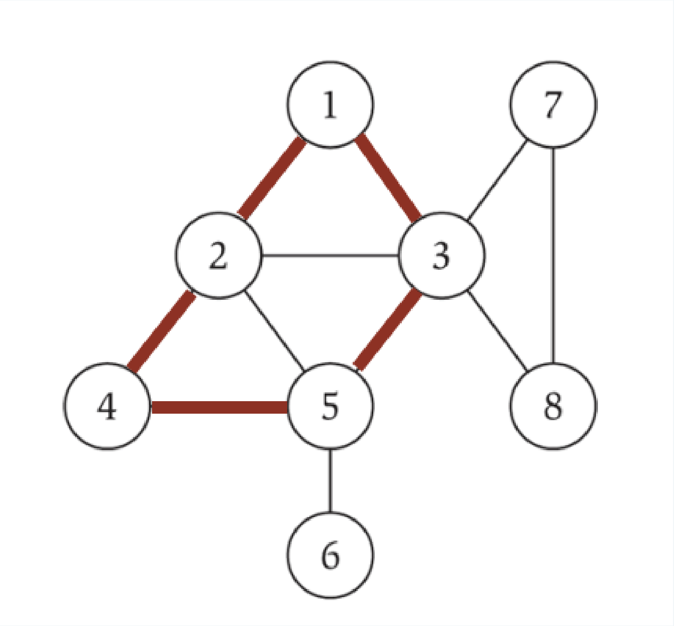
Def(3-3). An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.



3.1 Basic Definitions and Applications

Def(3-4). A cycle is a path $v_1, v_2, \cdots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct.

Def(3-5). A cycle is simple if all nodes are distinct (except for v_1 and v_k).



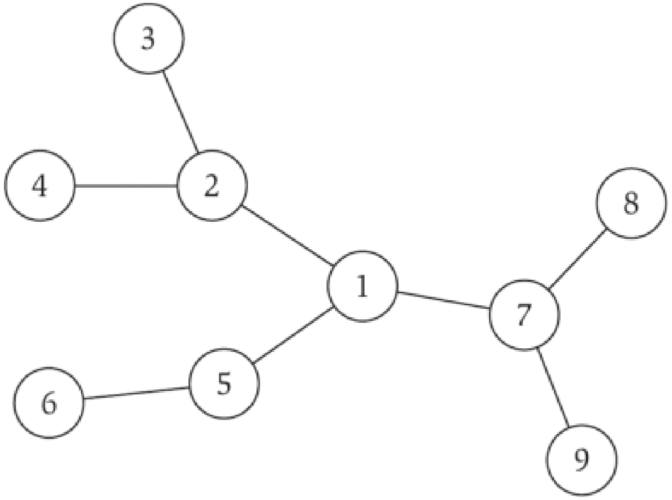
cycle C = 1-2-4-5-3-1

3.1 Basic Definitions and Applications

Def(3-6). An undirected graph is a tree if it is connected and does not contain a cycle.

Theorem(3-1). Let G be an undirected graph on n nodes. Any two of the following statements imply the third.

- G is *connected*.
- G does not contain a *cycle*.
- G has $n - 1$ edges.

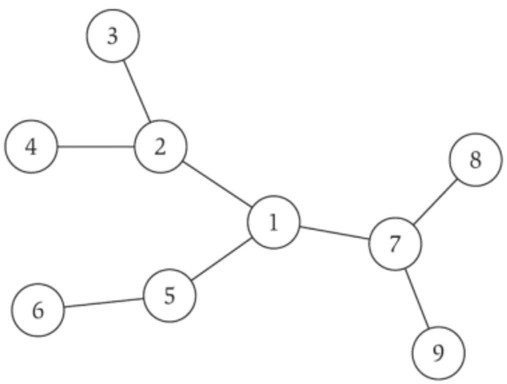


3.1 Basic Definitions and Applications

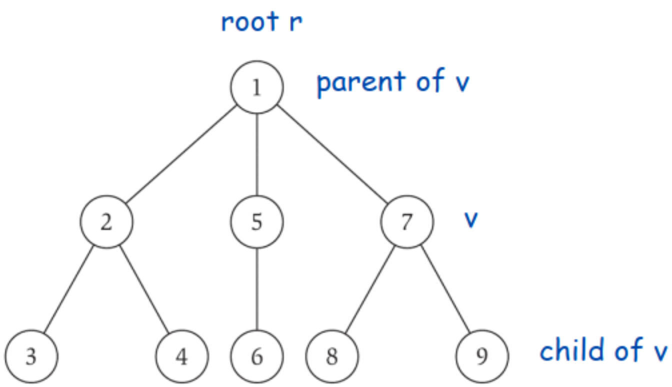
Rooted Trees

Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.

Importance. Models hierarchical structure.



a tree

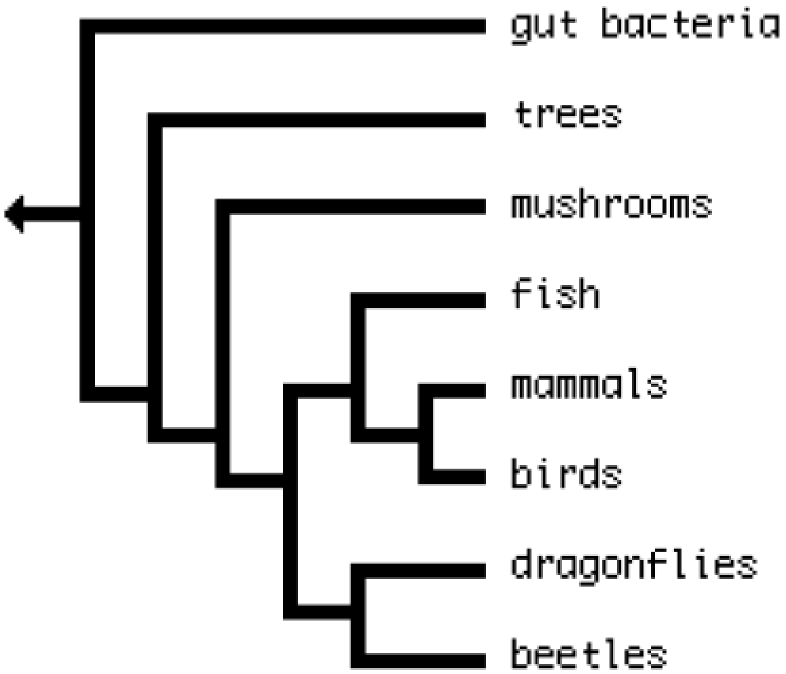


the same tree, rooted at 1

3.1 Basic Definitions and Applications

Phylogeny trees.

- Describe evolutionary history of species.



Chap03-Graphs Outline

3.1 Basic Definitions and Applications

3.2 Graph Traversal and Graph Connectivity

3.3 Testing Bipartiteness

3.4 Connectivity in Directed Graphs

3.5 DAGs and Topological Ordering

3.2 Graph Traversal and Graph Connectivity

Connectivity

- s-t connectivity problem. Given two node s and t, is there a path between s and t?
- s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Applications

- Kevin Bacon number
- Fewest hops in a communication network

3.2 Graph Traversal and Graph Connectivity

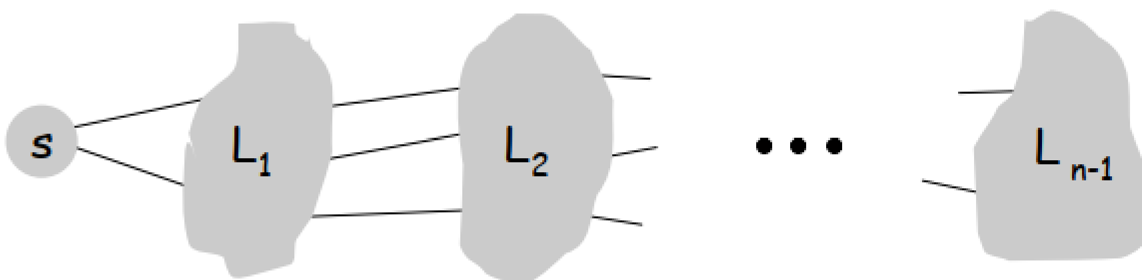
Breadth-First Search(BFS)

- BFS intuition.
  - Explore outward from s in all possible directions, adding nodes one "layer" at a time.
- BFS algorithm.
  - $L_0 = \{s\}$
  - L_1 = all neighbors of L_0.
  - L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
  - L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

3.2 Graph Traversal and Graph Connectivity

Input: s,G=(V,E)
Output: L, BFS tree T

```
Initialize visited and visited[s]=true
for v in V-s:
    visited[v] = false
End for
Initialize L and add s into L[0]
i = 0
Initialize T
while  L[i] is not empty:
    initialize L[i]
    for u in L[i]
        select (u,v) ∈ E
        if visited(v) == false:
            visited(v) = true
            add (u,v) into T
            add v into L[i+1]
        end if
    end for
    i = i + 1
end while
```

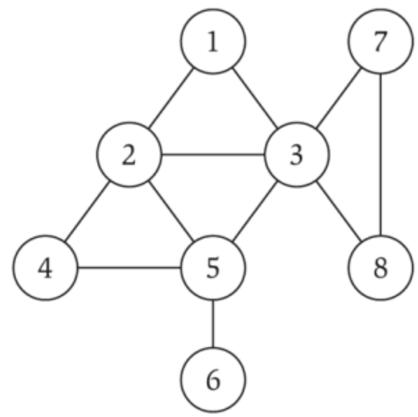


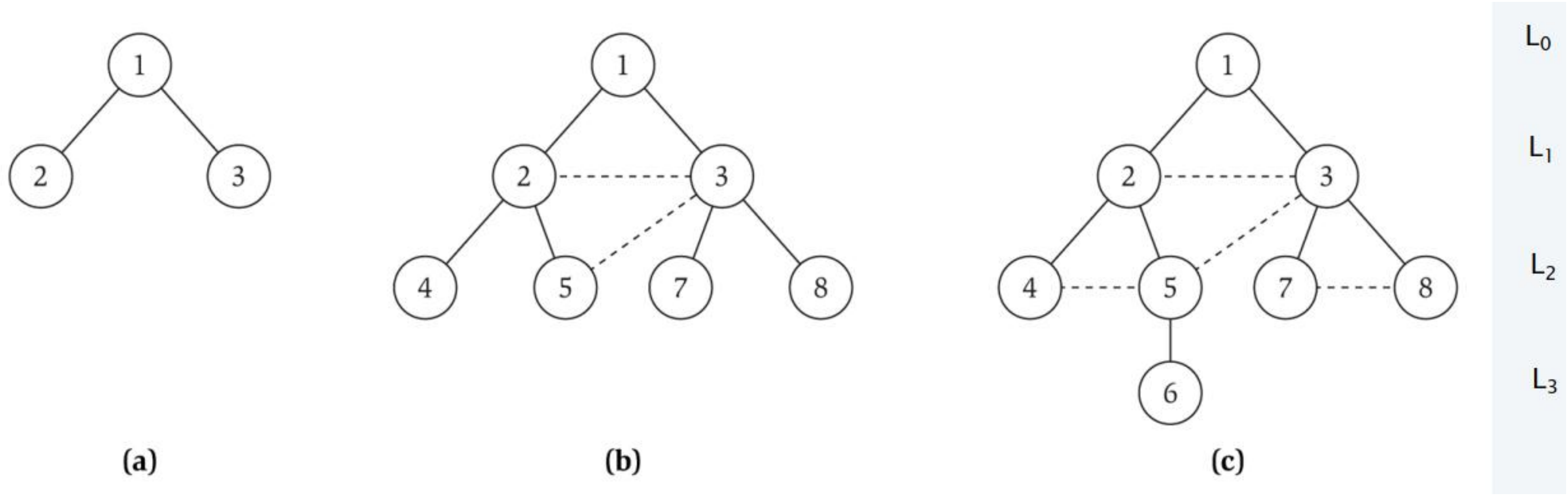
3.2 Graph Traversal and Graph Connectivity

Breadth-First Search

- Theorem(3-2). For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.
- Property. Let T be a BFS tree of $G = (V, E)$, and let (x, y) be an edge of G. Then, the levels of x and y differ by at most 1.

3.2 Graph Traversal and Graph Connectivity





(a) (b) (c)

3.2 Graph Traversal and Graph Connectivity

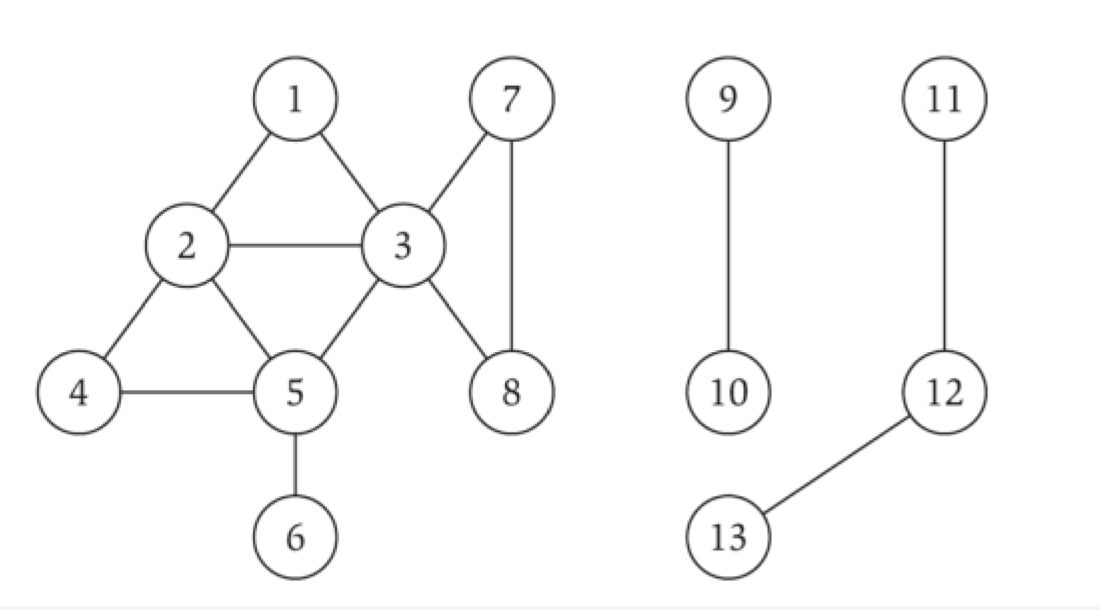
Theorem(3-3). The above BFS algorithm runs in $O(m+n)$ time if the graph is given by its adjacency representation.

Pf.

- Easy to prove $O(n^2)$ running time:
  - at most n lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node u, there are $\leq n$ incident edges (u, v), and we spend $O(1)$ processing each edge
- Actually runs in $O(m+n)$ time:
  - when we consider node u, there are $degree(u)$ incident edges (u, v)
  - total time processing edges is $\sum_{u \in V} degree(u) = 2m$

3.2 Graph Traversal and Graph Connectivity

Connected component. Find all nodes reachable from s.



Connected component containing node $1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$.

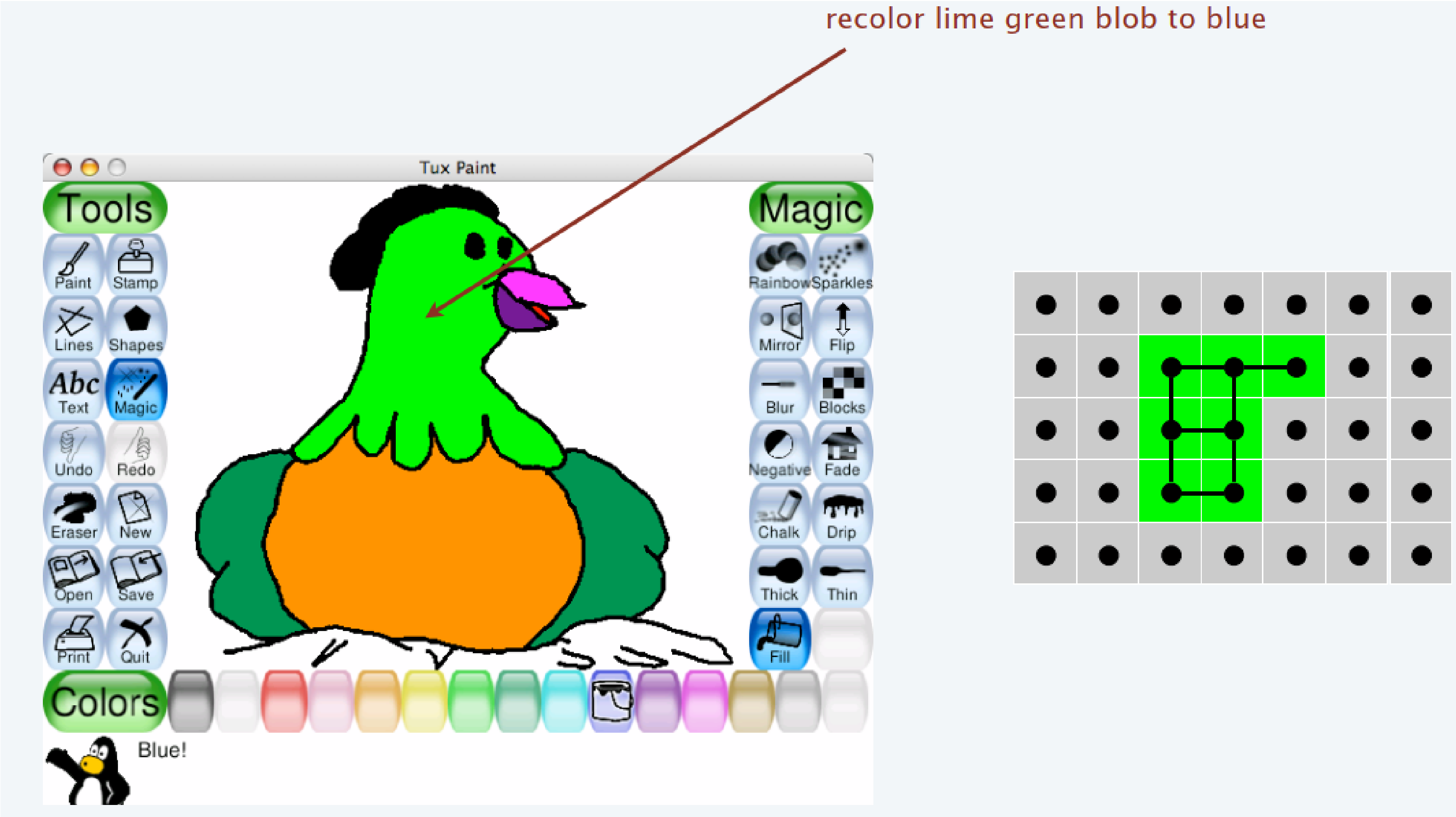
3.2 Graph Traversal and Graph Connectivity

Application

- Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

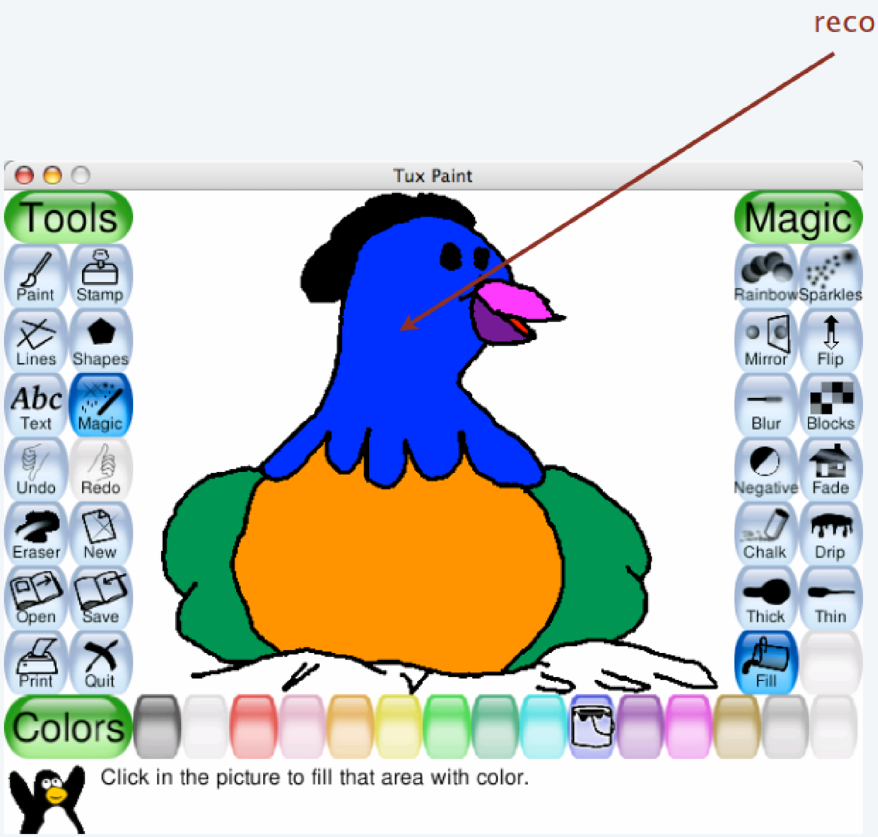
3.2 Graph Traversal and Graph Connectivity

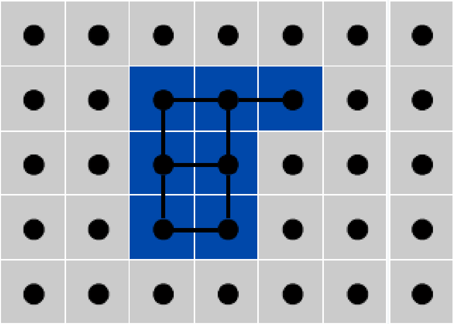
Flood fill



3.2 Graph Traversal and Graph Connectivity

Flood fill

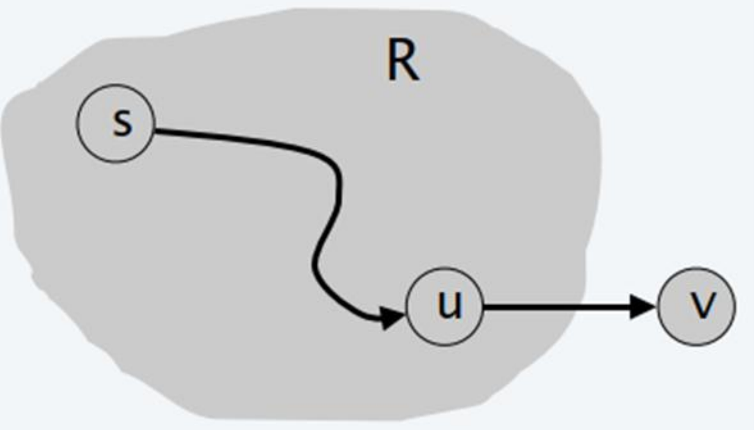




3.2 Graph Traversal and Graph Connectivity

Connected component. Find all nodes reachable from s.

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile



it's safe to add v

3.2 Graph Traversal and Graph Connectivity

Theorem(3-4). Upon termination, R is the connected component containing s.

- BFS = explore in order of distance from s.
- DFS = explore in a different way.

Chap03-Graphs Outline

3.1 Basic Definitions and Applications

3.2 Graph Traversal and Graph Connectivity

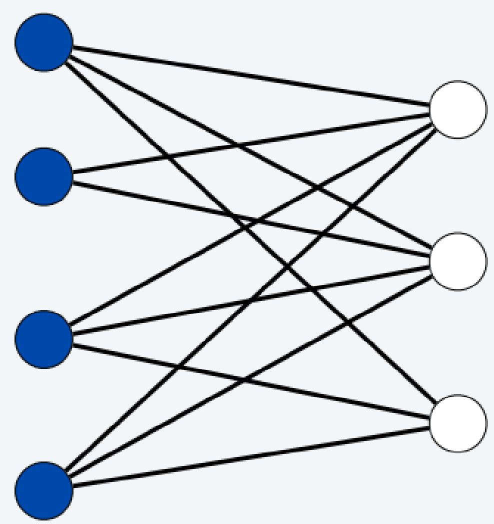
3.3 Testing Bipartiteness

3.4 Connectivity in Directed Graphs

3.5 DAGs and Topological Ordering

3.3 Testing Bipartiteness

Def(3-7). An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored blue or white such that every edge has one white and one blue end.



a bipartite graph

3.3 Testing Bipartiteness

Applications.

- Stable matching: med-school students= blue, hospitals = white.
- Scheduling: machines = blue, jobs = white.

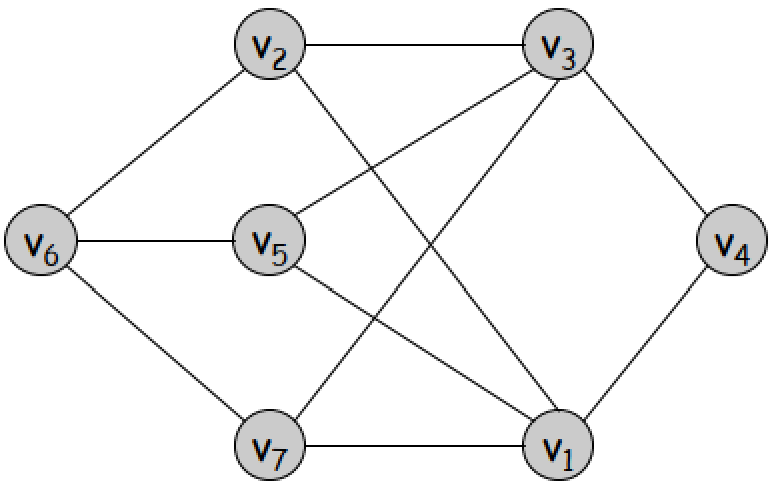
3.3 Testing Bipartiteness

Testing Bipartiteness

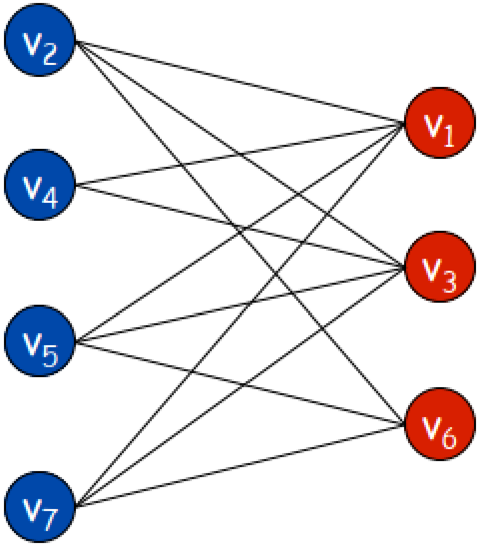
- Given a graph G, is it bipartite?
- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

3.3 Testing Bipartiteness

For example



a bipartite graph G



another drawing of G

3.3 Testing Bipartiteness

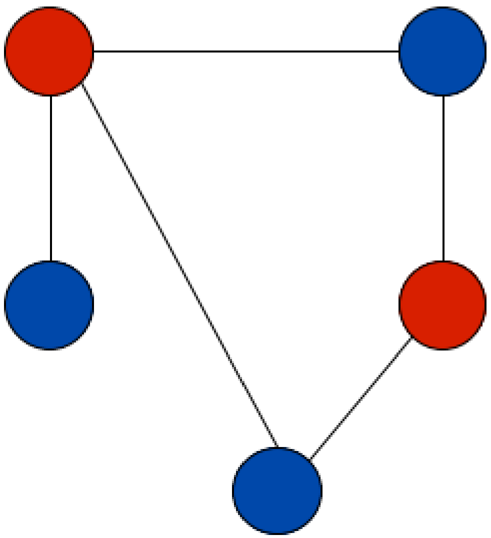
An Obstruction to Bipartiteness

Lemma(3-1). If a graph G is bipartite, it cannot contain an odd length cycle.

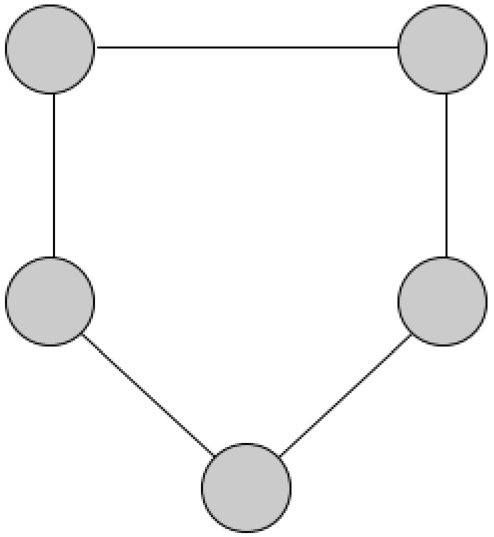
Pf. Not possible to 2-color the odd cycle, let alone G.

3.3 Testing Bipartiteness

For example



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

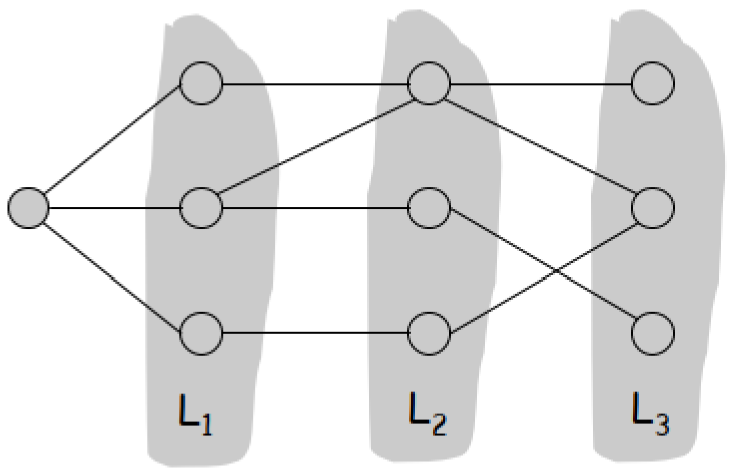
3.3 Testing Bipartiteness

Lemma(3-2). Let G be a connected graph, and let $L_0, \cdots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

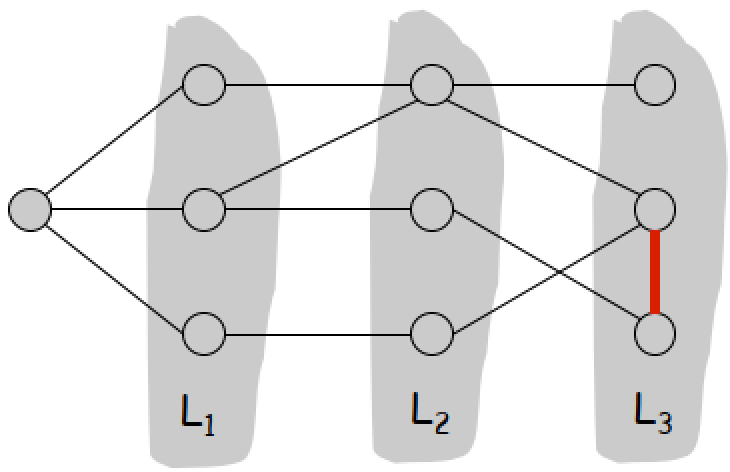
- (i) No edge of G joins two nodes of the same layer, and G is bipartite.
- (ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

3.3 Testing Bipartiteness

For example

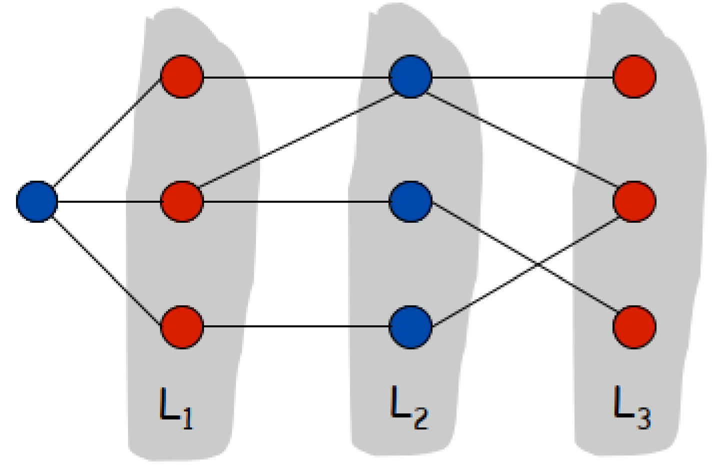


Case (i)



Case (ii)

3.3 Testing Bipartiteness



Case (i)

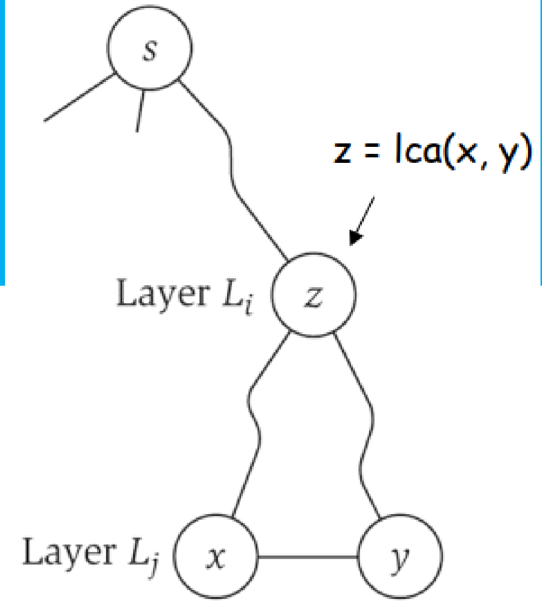
Lemma(3-2).

Pf.

(i)

- Suppose no edge joins two nodes in same layers.
- By previous lemma, this implies all edges join nodes on same level.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.

3.3 Testing Bipartiteness



Lemma(3-2).

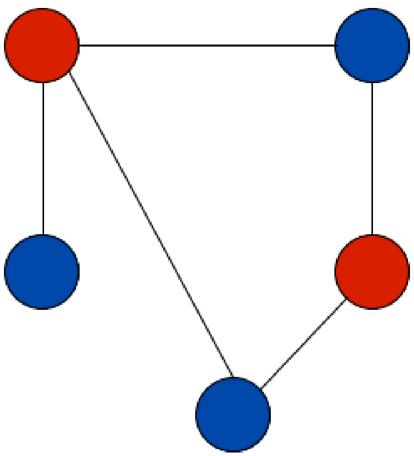
Pf.(ii)

- Suppose (x, y) is an edge with x, y in same level L_j.
- Let $z = lca(x, y)$ = lowest common ancestor.
- Let L_i be level containing z.
- Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
- Its length is $1 + (j - i) + (j - i)$, which is odd.

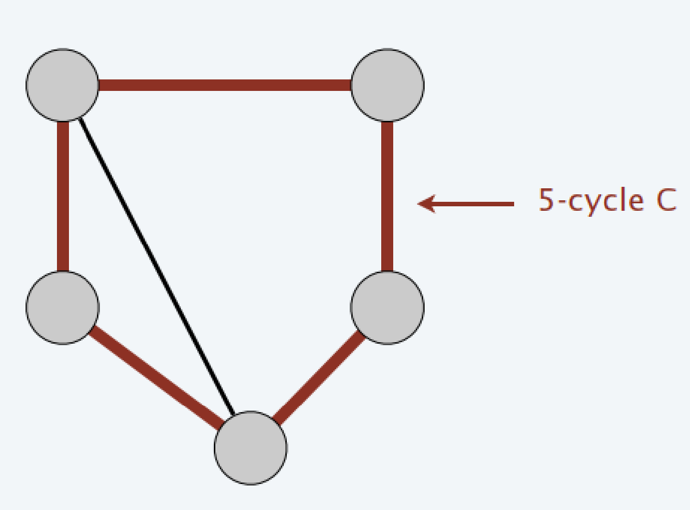
3.3 Testing Bipartiteness

The only obstruction to bipartiteness

Corollary(3-1). A graph G is bipartite iff it contains no odd-length cycle.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

Chap03-Graphs Outline

3.1 Basic Definitions and Applications

3.2 Graph Traversal and Graph Connectivity

3.3 Testing Bipartiteness

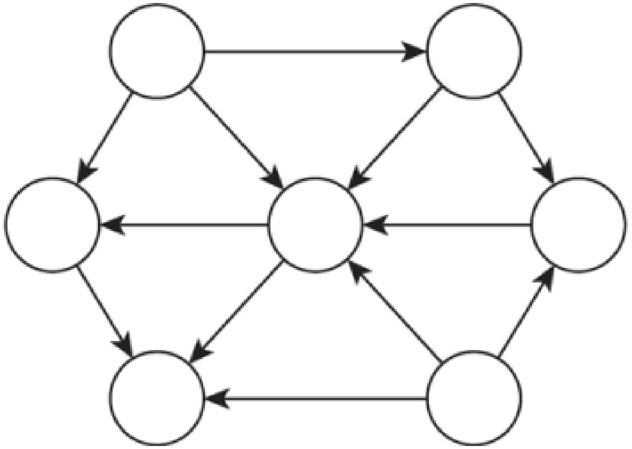
3.4 Connectivity in Directed Graphs

3.5 DAGs and Topological Ordering

3.4 Connectivity in Directed Graphs

Directed graphs. $G = (V, E)$

- Edge (u, v) goes from node u to node v.



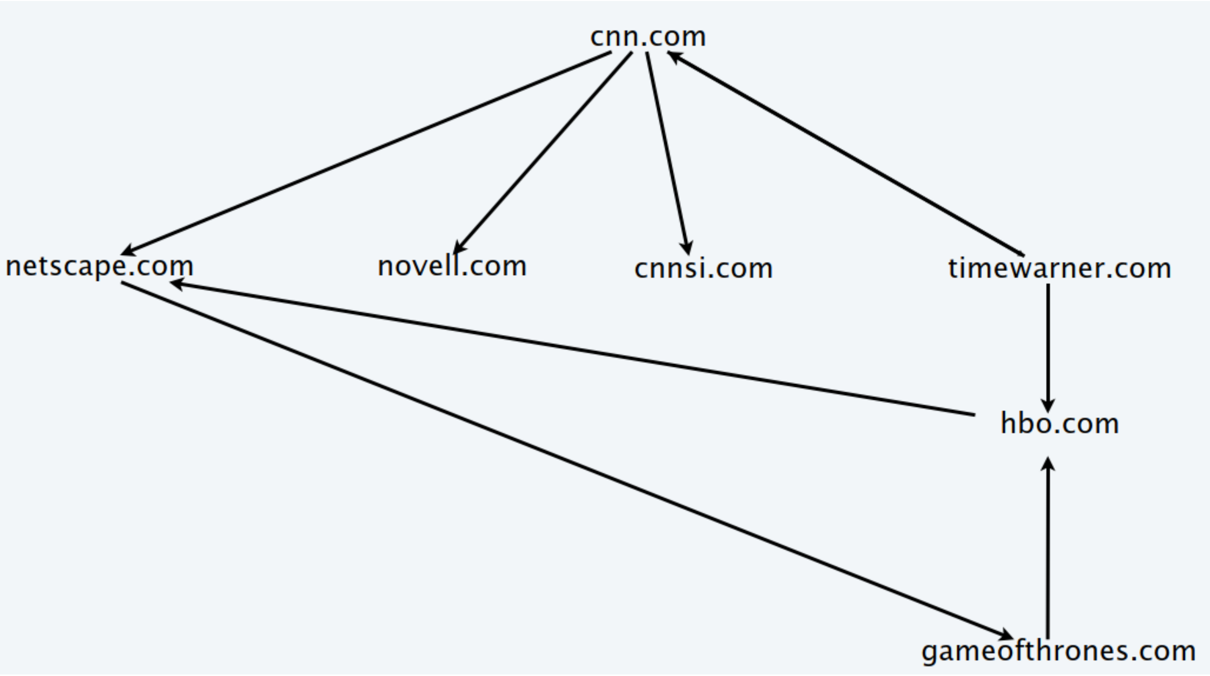
E.g. **Web graph**: hyperlink points from one web page to another.

- Orientation of edges is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.

3.4 Connectivity in Directed Graphs

Web graph.

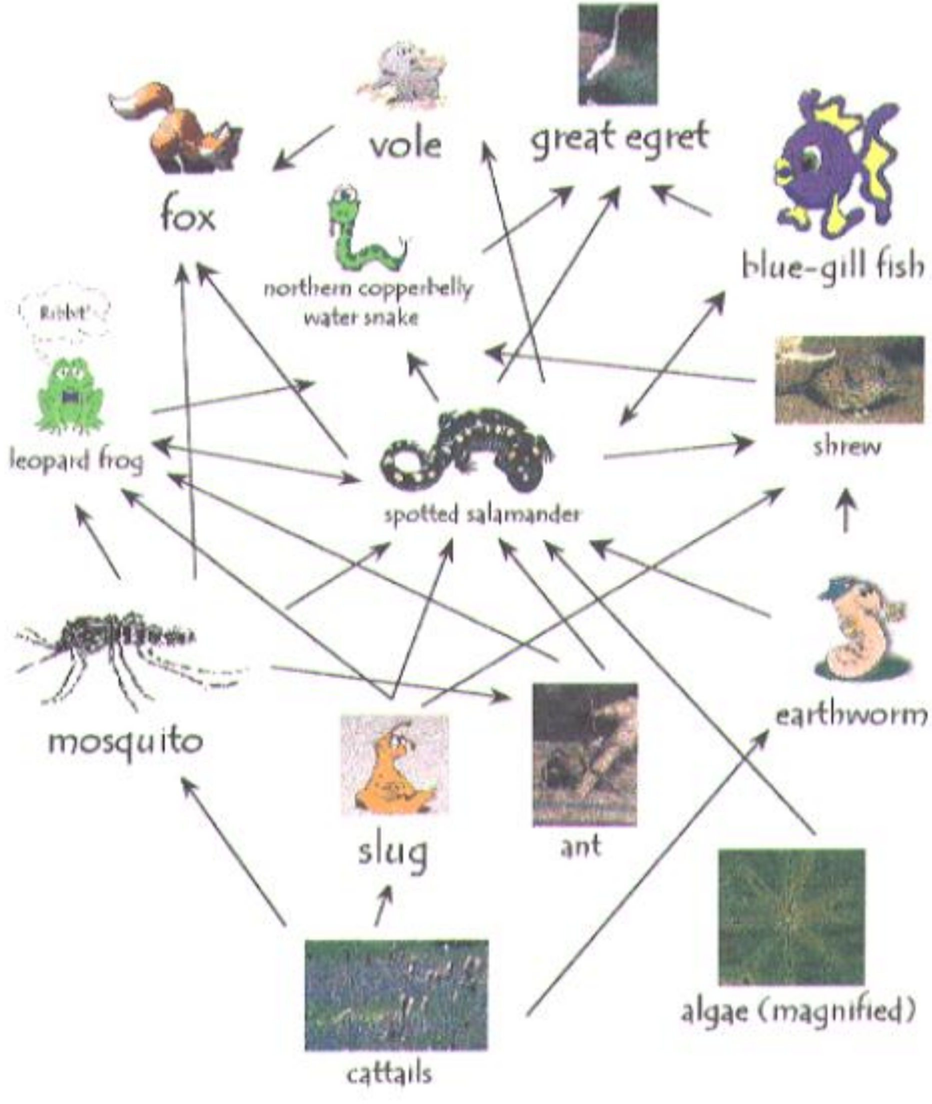
- Node: web page.
- Edge: hyperlink from one page to another (orientation is crucial).
- Modern search engines exploit hyperlink structure to rank web pages by importance.



3.4 Connectivity in Directed Graphs

Food web graph.

- Node = species.
- Edge = from prey to predator.



3.4 Connectivity in Directed Graphs

Some directed graph applications

directed graph	node	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

3.4 Connectivity in Directed Graphs

Graph search

Directed reachability.

- Given a node s, find all nodes reachable from s.

Directed s-t shortest path problem.

- Given two node s and t, what is the length of the shortest path between s and t?

Graph search.

- BFS extends naturally to directed graphs.

Web crawler.

- Start from web page s. Find all web pages linked from s, either directly or indirectly.

3.4 Connectivity in Directed Graphs

Def(3-8). Nodes u and v are mutually reachable if there is both a path from u to v and also a path from v to u.

Def(3-9). A graph is strongly connected if every pair of nodes is mutually reachable.

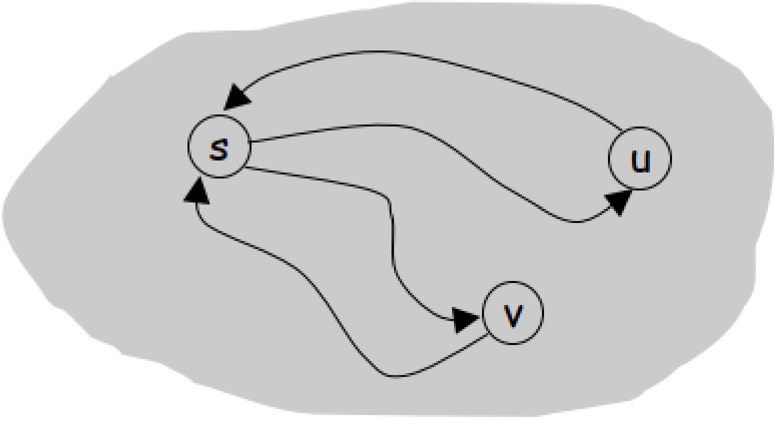
3.4 Connectivity in Directed Graphs

Lemma(3-3). Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf. $\Rightarrow$ Follows from definition.

Pf. $\Leftarrow$ Path from u to v: concatenate u-s path with s-v path.

Path from v to u: concatenate v-s path with s-u path.



3.4 Connectivity in Directed Graphs

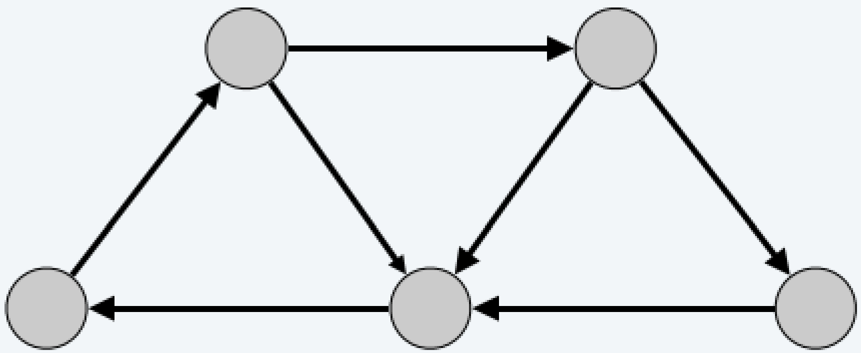
Theorem(3-5). Can determine if G is strongly connected in $O(m + n)$ time.

Pf.

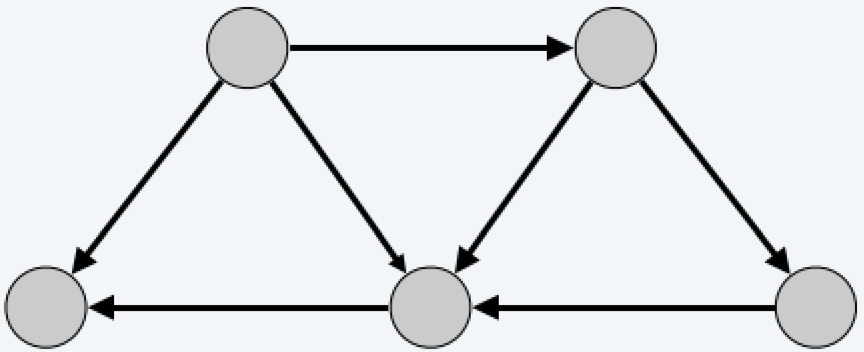
- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in G^{rev}.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.

3.4 Connectivity in Directed Graphs

E.g.



strongly connected



not strongly connected

Chap03-Graphs Outline

3.1 Basic Definitions and Applications

3.2 Graph Traversal and Graph Connectivity

3.3 Testing Bipartiteness

3.4 Connectivity in Directed Graphs

3.5 DAGs and Topological Ordering

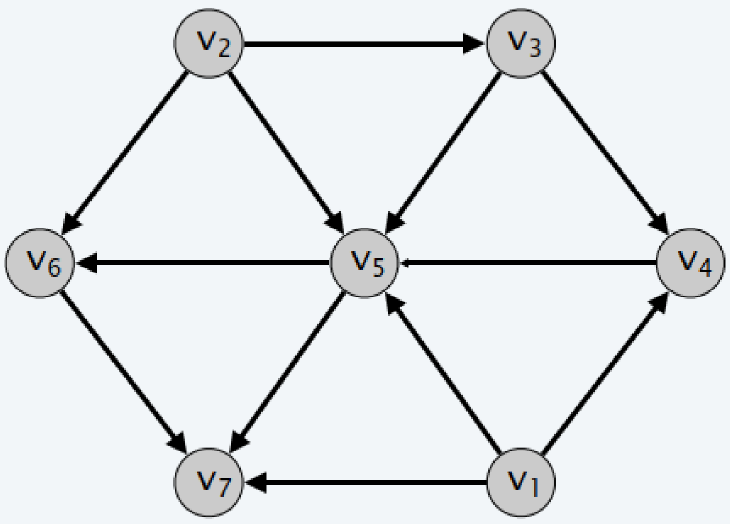
3.5 DAGs and Topological Ordering

Def(3-10). An Directed Acyclic Graphs (DAG) is a directed graph that contains no directed cycles.

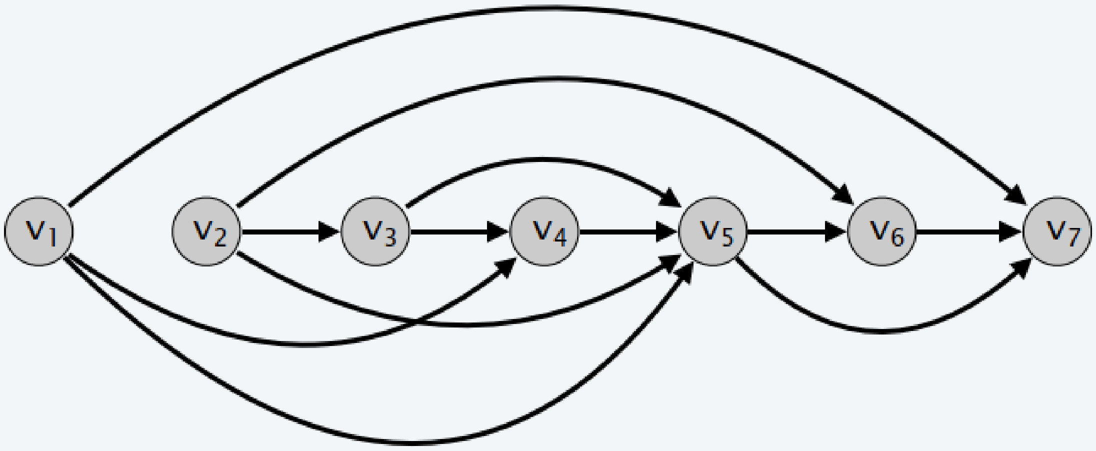
Def(3-11). A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \cdots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.

3.5 DAGs and Topological Ordering

E.g.



a DAG



a topological ordering

3.5 DAGs and Topological Ordering

Precedence constraints

- Edge (v_i, v_j) means task v_i must occur before v_j.

Applications

- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j.
- Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

3.5 DAGs and Topological Ordering

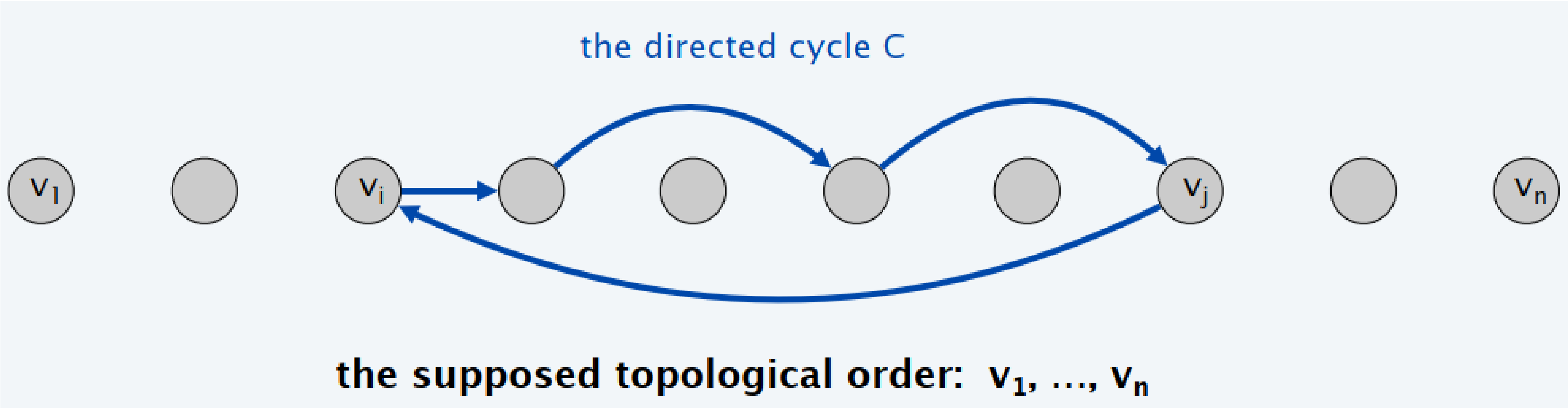
Lemma(3-4). If G has a topological order, then G is a DAG.

Pf. [by contradiction]

- Suppose that G has a topological order $v_1, v_2, \cdots, v_n$ and that G also has a directed cycle C. Let's see what happens.
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, v_2, \cdots, v_n$ is a topological order, we must have $j < i$, a contradiction.

3.5 DAGs and Topological Ordering

E.g.



3.5 DAGs and Topological Ordering

Lemma(3-5). If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?

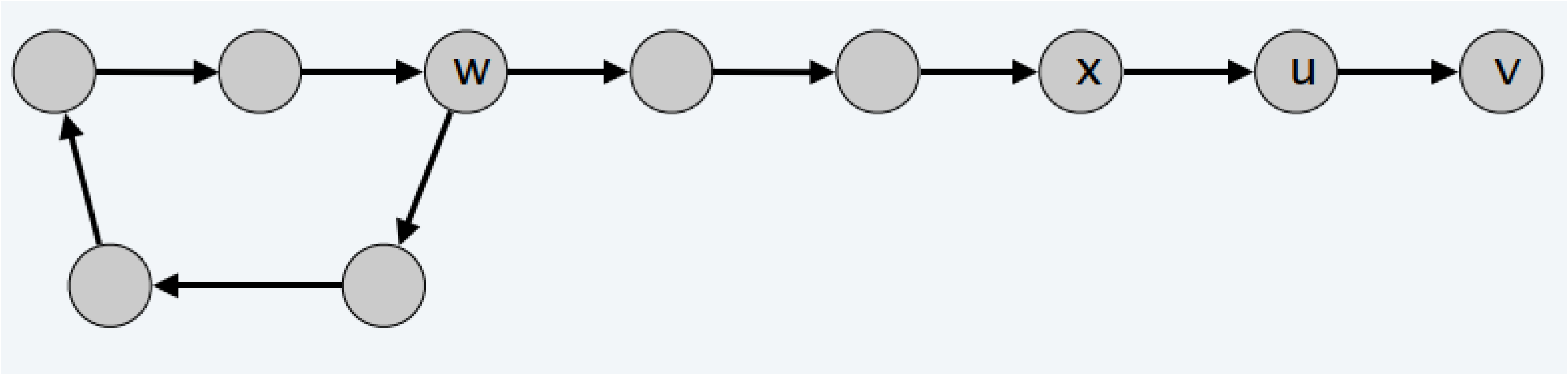
3.5 DAGs and Topological Ordering

Lemma(3-6). If G is a DAG, then G has a node with no incoming edges.

Pf. [by contradiction]

- Suppose that G is a DAG and every node has at least one entering edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one entering edge (u, v) we can walk backward to u.
- Then, since u has at least one entering edge (x, u), we can walk backward to x.
- Repeat until we visit a node, say w, twice.
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle.

3.5 DAGs and Topological Ordering



3.5 DAGs and Topological Ordering

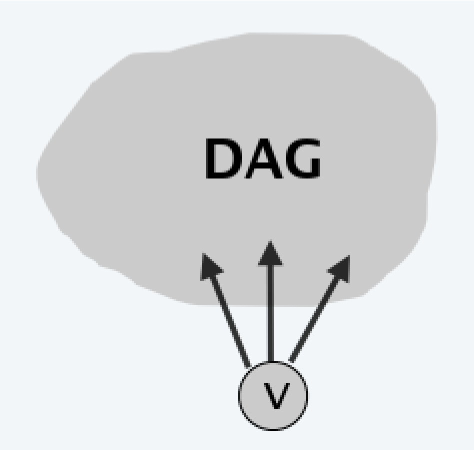
Lemma(3-7). If G is a DAG, then G has a topological ordering.

Pf. [by induction on n]

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node v with no entering edges.
- $G - \{v\}$ is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place v first in topological ordering; then append nodes of $G - \{v\}$
- in topological order. This is valid since v has no entering edges.

3.5 DAGs and Topological Ordering

The inductive proof contains the following algorithm to compute a topological ordering of G.




```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
   and append this order after v
```

3.5 DAGs and Topological Ordering

Theorem(3-6). Algorithm finds a topological order in $O(m+n)$ time.

Pf.

- Maintain the following information:
  - $count(w)$ = remaining number of incoming edges
  - S = set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete v
  - remove v from S
  - decrement $count(w)$ for all edges from v to w; and add w to S if $count(w)$ hits 0
  - this is $O(1)$ per edge

Thanks for Listening

College of Computer Science
Nankai University
Tianjin, P.R.China