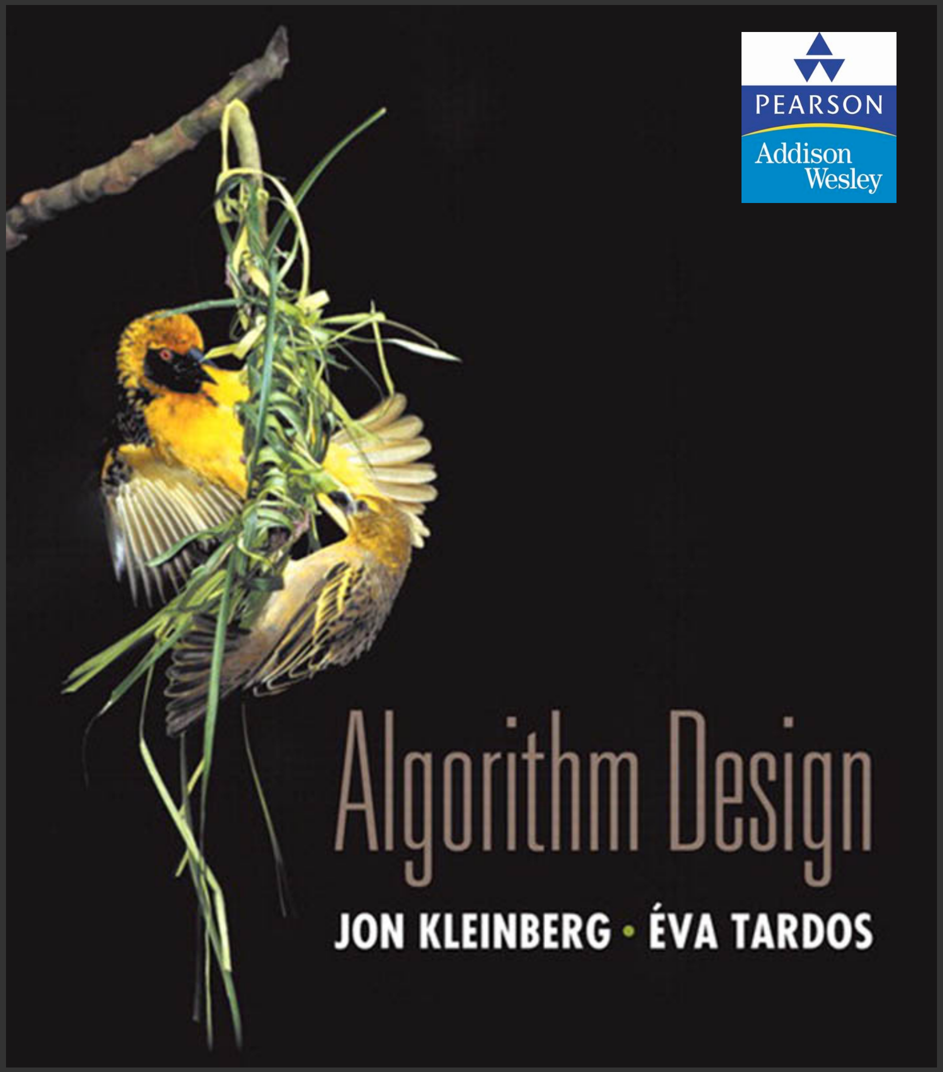
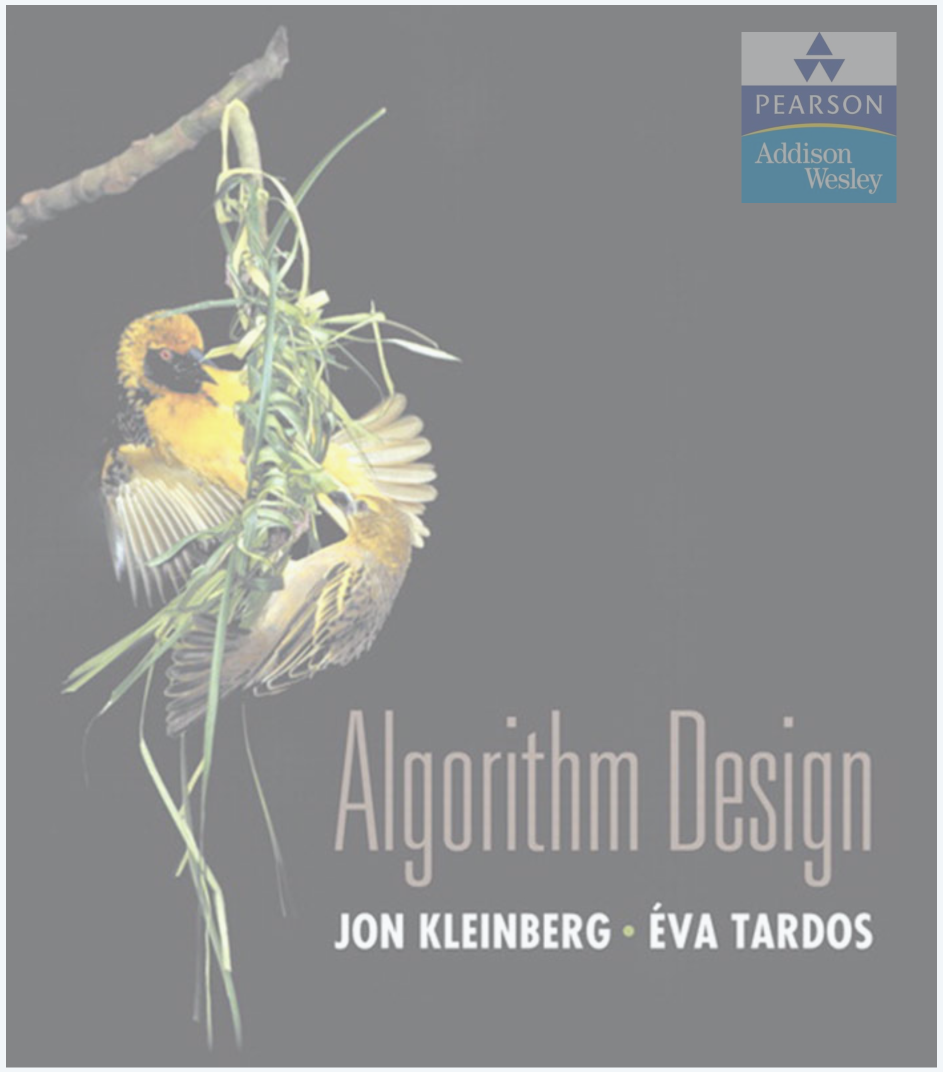


3. Graphs

- *basic definitions and applications*
- *graph connectivity and graph traversal*
- *testing bipartiteness*
- *connectivity in directed graphs*
- *DAGs and topological ordering*

Last updated on 2020/10/9 下午7:50



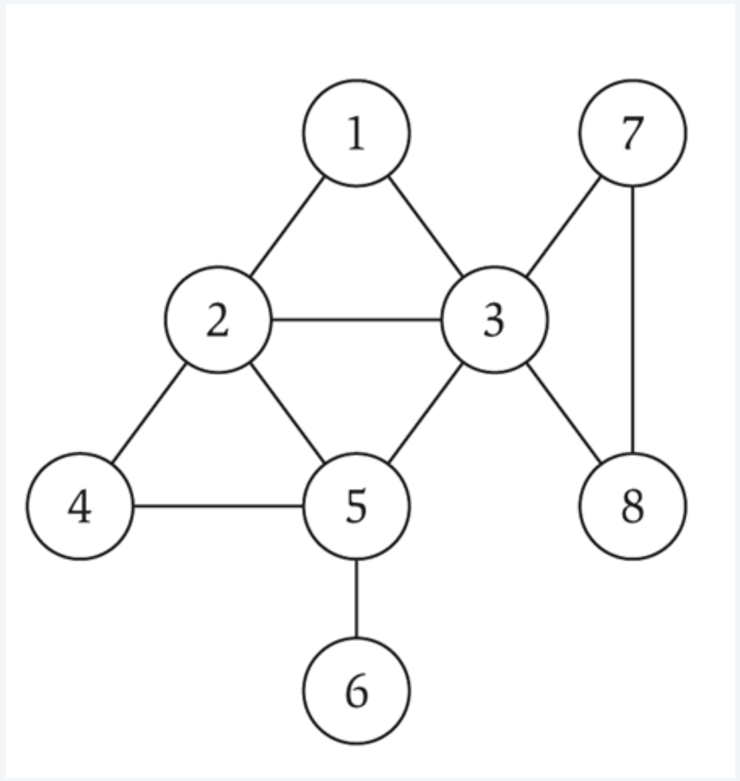
3. Graphs

- *basic definitions and applications*
- *graph connectivity and graph traversal*
- *testing bipartiteness*
- *connectivity in directed graphs*
- *DAGs and topological ordering*

Undirected graphs

Notation. $G = (V, E)$

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



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

$E = \{ 1–2, 1–3, 2–3, 2–4, 2–5, 3–5, 3–7, 3–8, 4–5, 5–6, 7–8 \}$

$m = 11, n = 8$

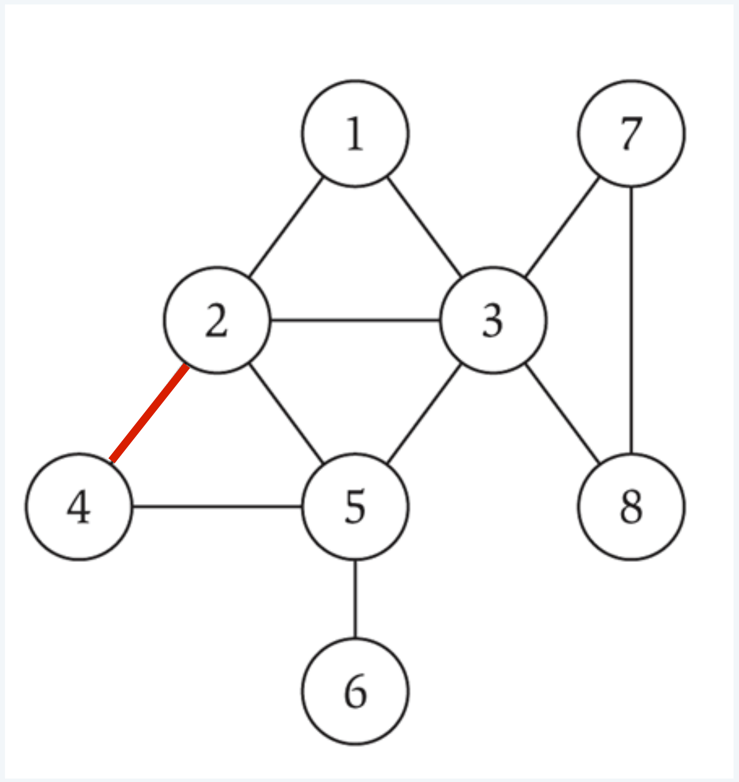
Some graph applications

graph	node	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

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.



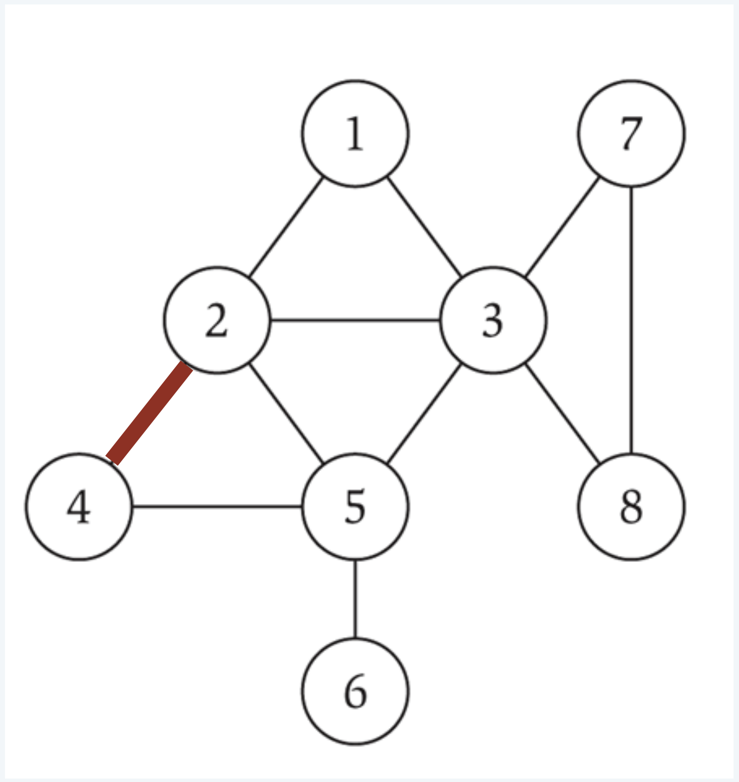
$$
\begin{array}{c|cccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
3 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
5 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
7 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

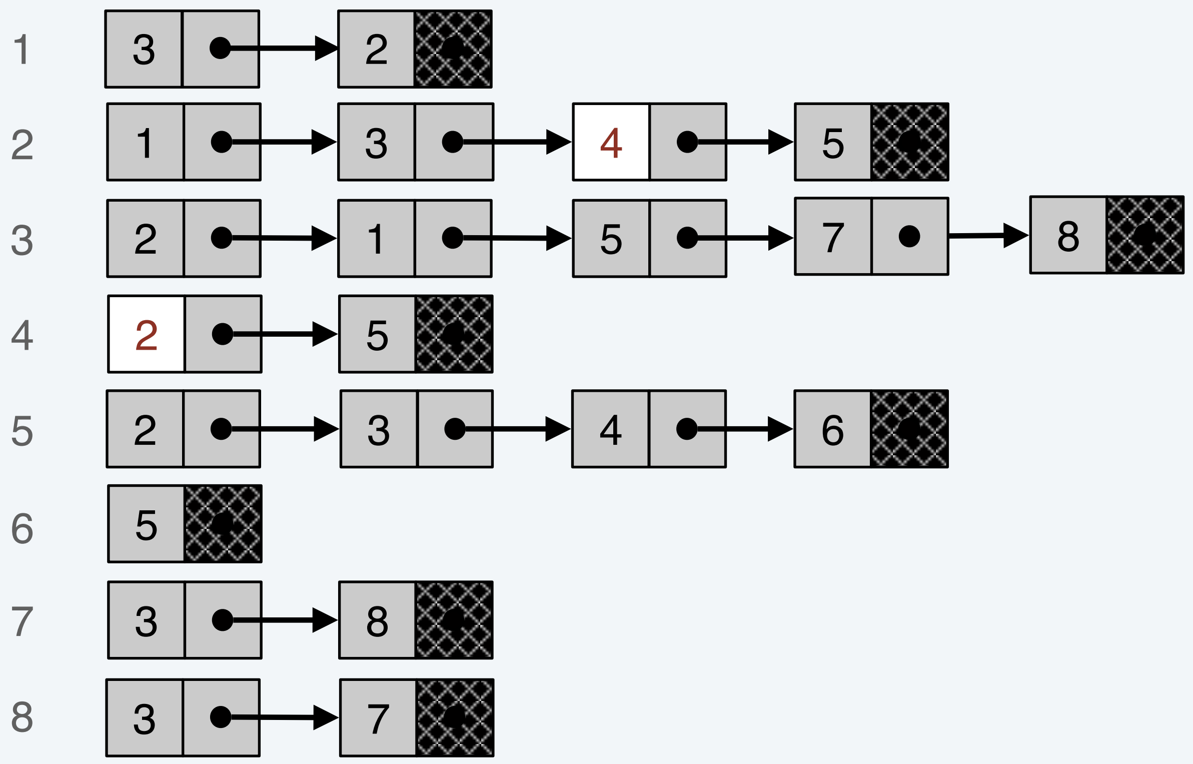
Graph representation: adjacency lists

Adjacency lists. Node-indexed array of lists.

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

degree = number of neighbors of u





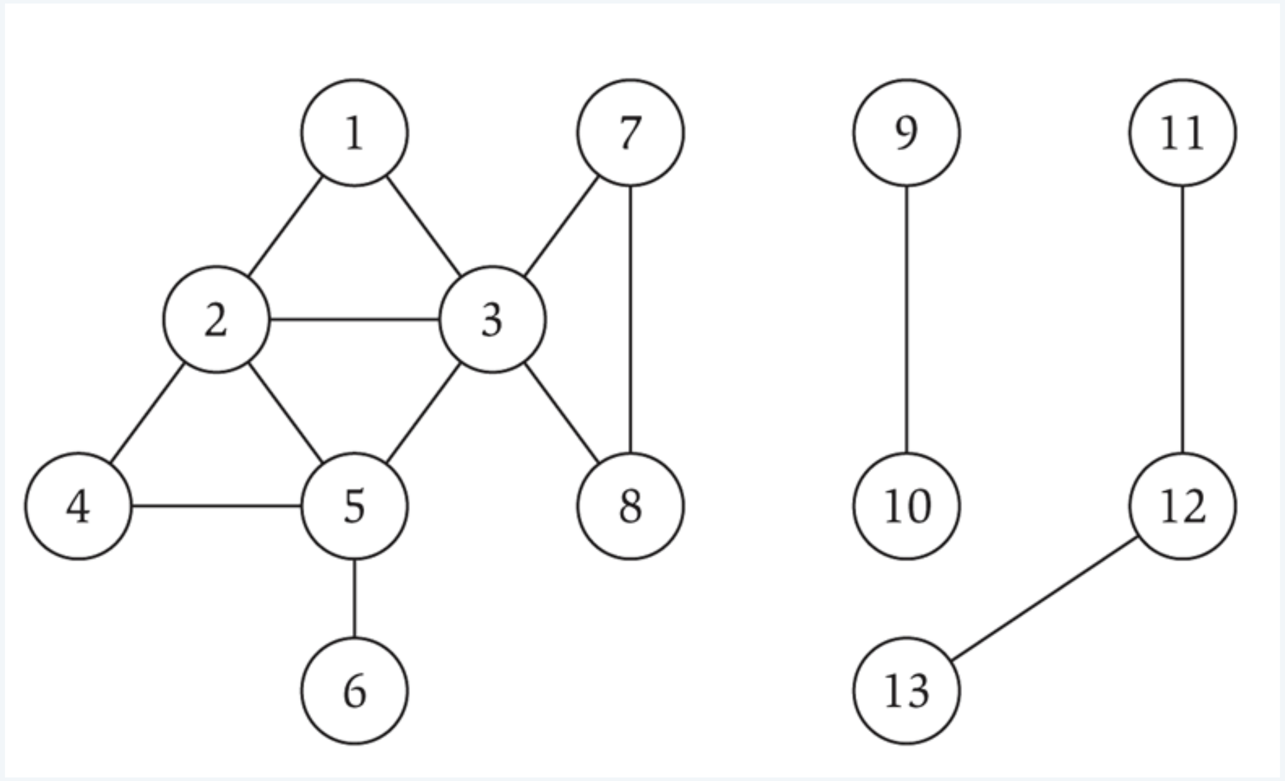
1	3	2			
2	1	3	4	5	
3	2	1	5	7	8
4	2	5			
5	2	3	4	6	
6	5				
7	3	8			
8	3	7			

Paths and connectivity

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

Def. A path is simple if all nodes are distinct.

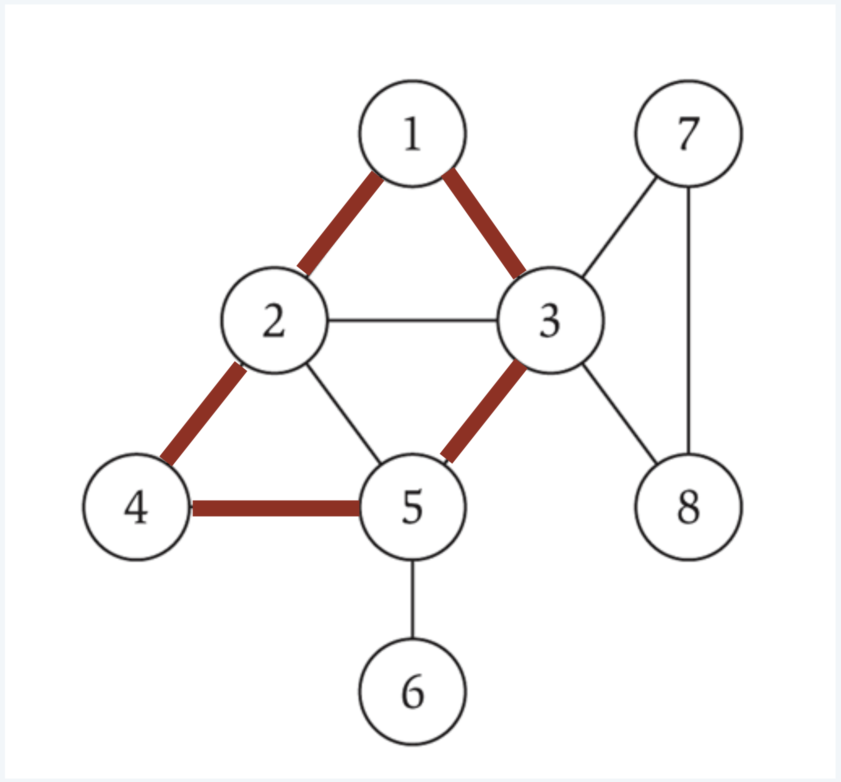
Def. An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.



Cycles

Def. A cycle is a path $v_1, v_2, \ldots, v_k$ in which $v_1 = v_k$ and $k \geq 2$.

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



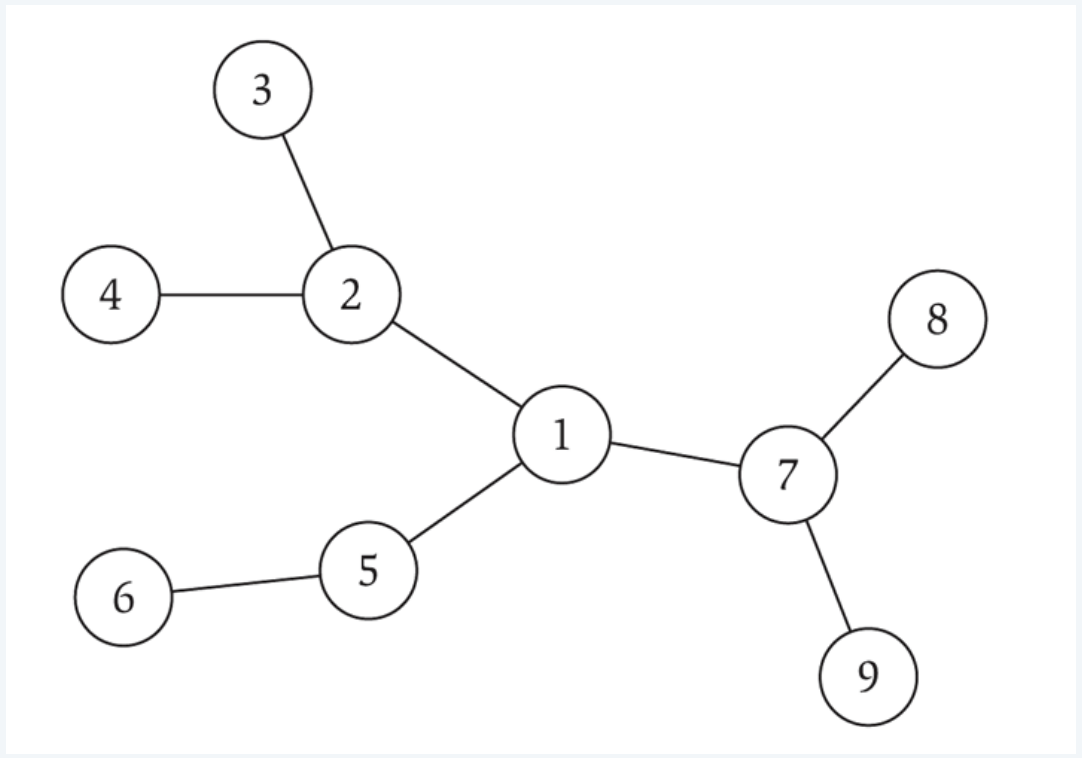
cycle C = 1-2-4-5-3-1

Trees

Def. An undirected graph is a tree if it is connected and does not contain a cycle.

Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third:

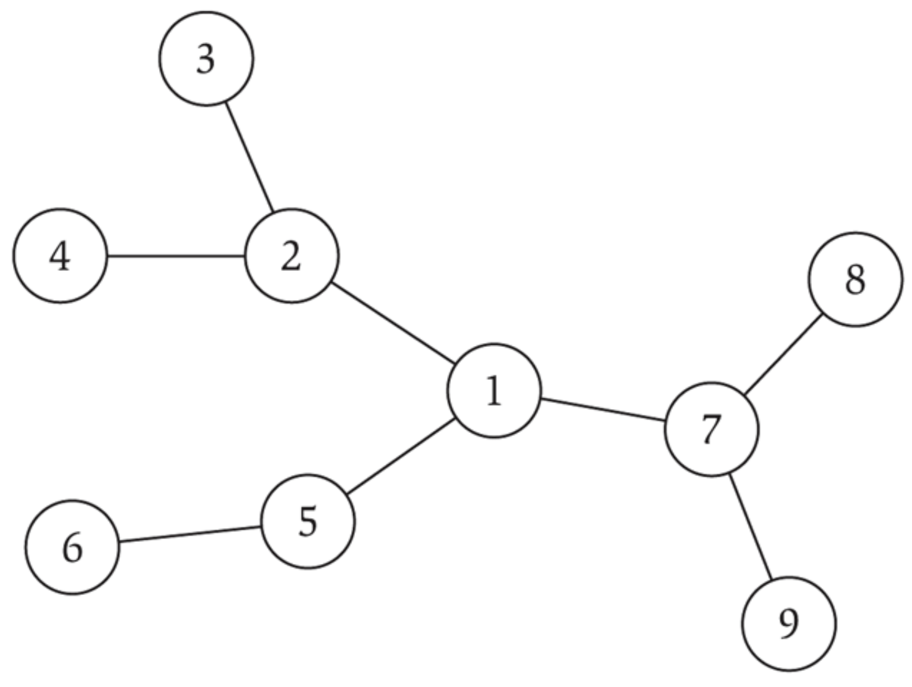
- G is connected.
- G does not contain a cycle.
- G has $n - 1$ edges.



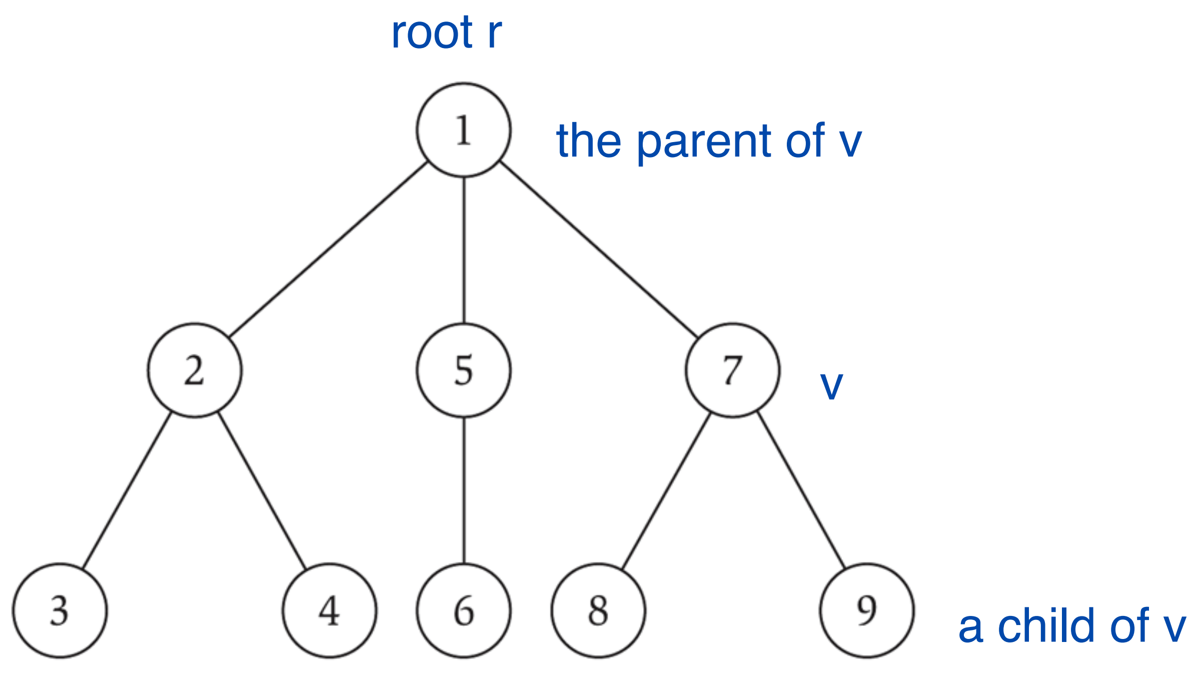
Rooted trees

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

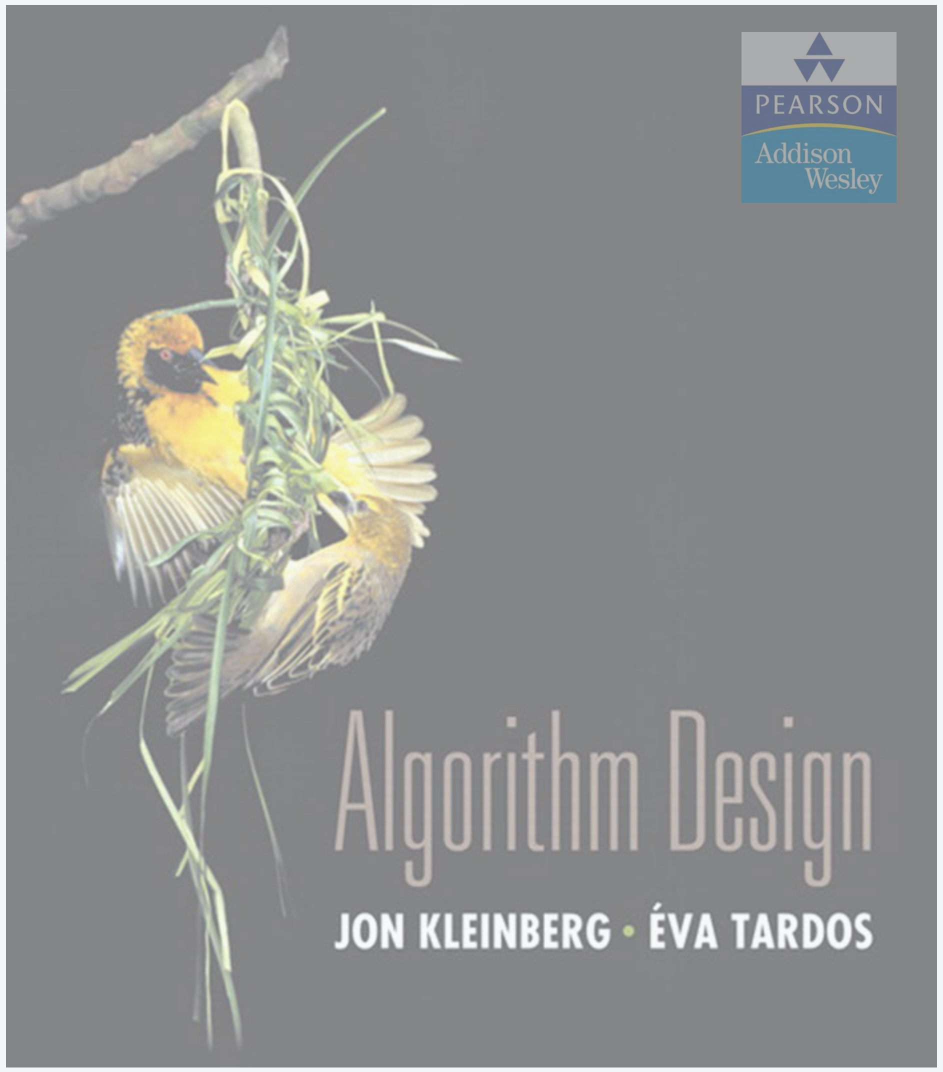
Importance. Models hierarchical structure.



a tree



the same tree, rooted at 1



3. Graphs

- *basic definitions and applications*
- *graph connectivity and graph traversal*
- *testing bipartiteness*
- *connectivity in directed graphs*
- *DAGs and topological ordering*

Connectivity

s-t connectivity problem. Given two nodes s and t, is there a path between s and t ?

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

Applications.

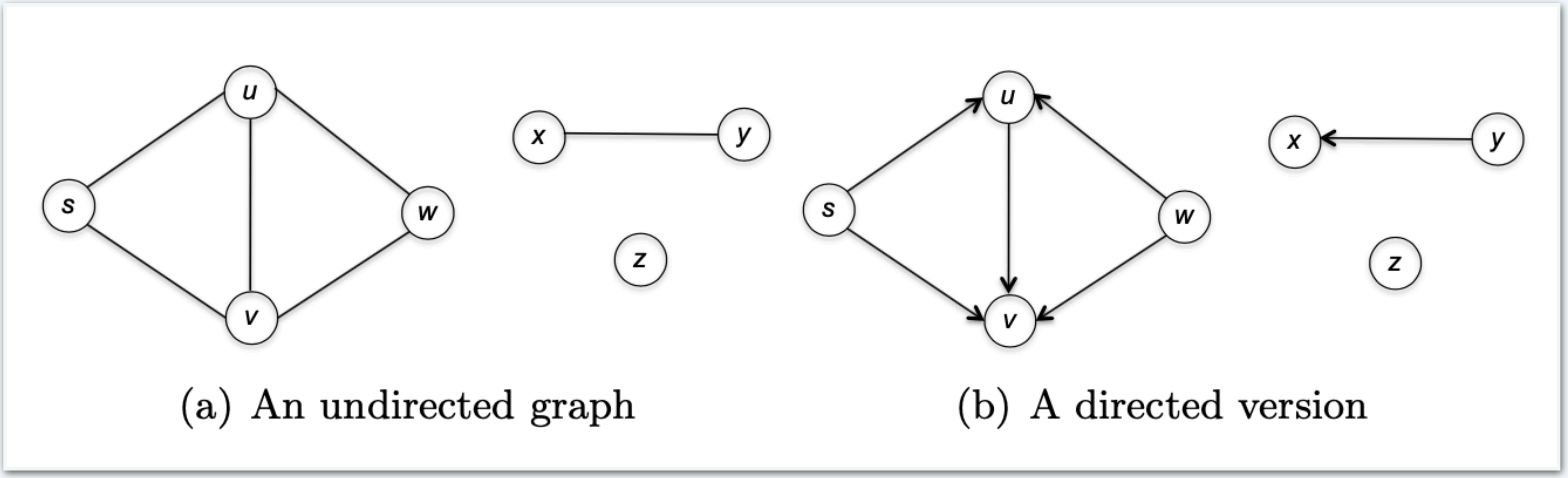
- Friendster.
- Maze traversal.
- Kevin Bacon number.
- Fewest hops in a communication network.

Graph Search Problem

Input. A graph $G = (V, E)$ and a starting vertex $s \in V$

Goal. Identify the vertices of V reachable from s in G

Example.



(a) An undirected graph (b) A directed version

In (a), $\{s, u, v, w\}$ are reachable from s

In (b), $\{s, u, v\}$ are reachable from s

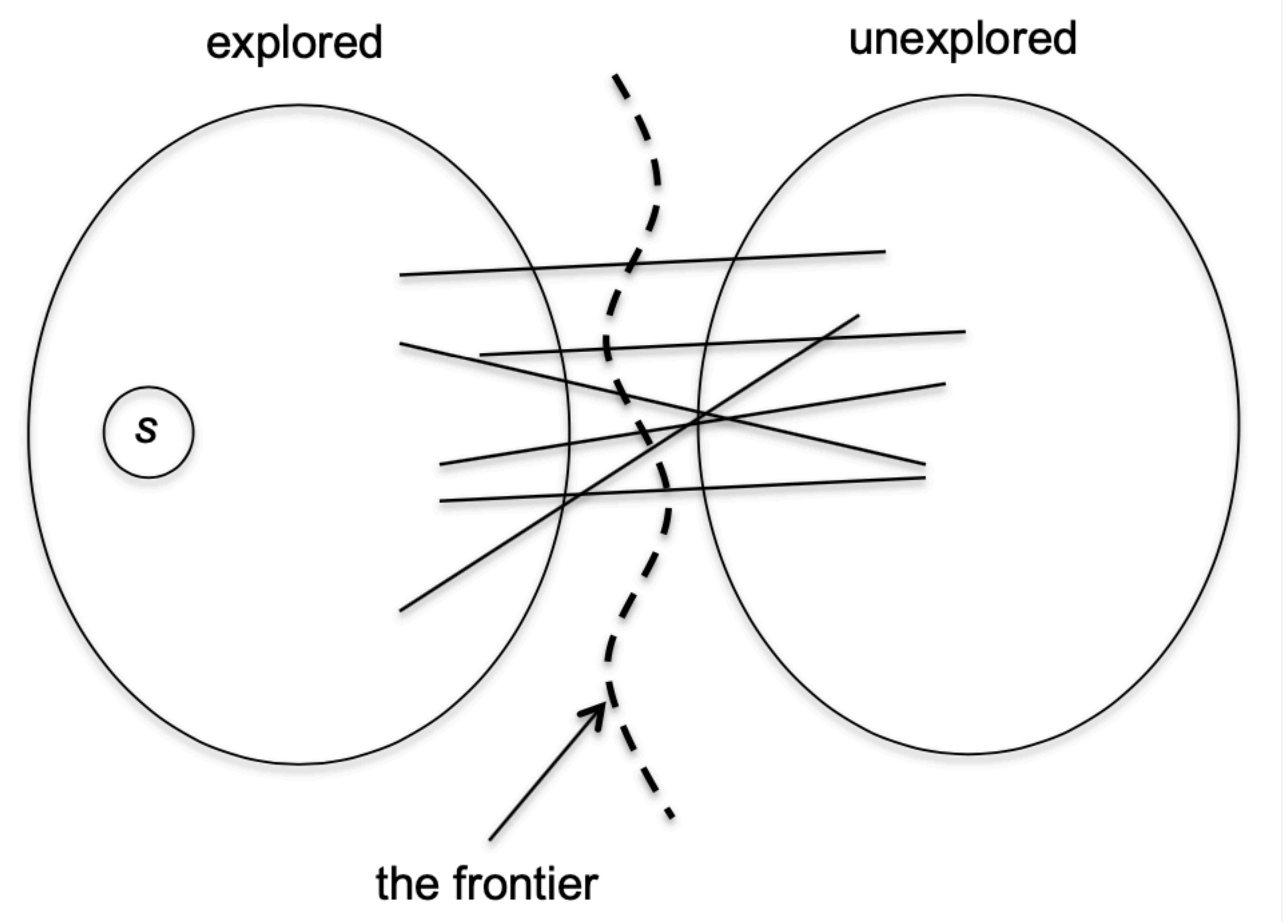
Generic Graph Search Strategy

Input. A graph $G = (V, E)$ and a starting vertex $s \in V$

Postcondition. a vertex is reachable from s if and only if it is marked as explored

GENERICSEARCH (G, s)

```
mark $s$ as explored, all other vertices as unexplored
WHILE there is an edge $(v, w) \in E$ with $v$ explored and $w$ unexplored
    choose some such edge $(v, w)$
    mark $w$ as explored
```



Generic Graph Search Strategy

Input. A graph $G = (V, E)$ and a starting vertex $s \in V$

Postcondition. a vertex is reachable from s if and only if it is marked as explored

```
GENERICSEARCH $(G, s)$

mark $s$ as explored, all other vertices as unexplored
WHILE there is an edge $(v, w) \in E$ with $v$ explored and $w$ unexplored
    choose some such edge $(v, w)$
    mark $w$ as explored
```

Complexity. Likely to be linear time as long as an eligible edge can be quickly identified in each iteration of the while loop

Correctness. At the conclusion the GenericSearch algorithm, a vertex $v \in V$ is marked as explored if and only if there is a path from s to v in G

Correctness of GenericSearch

Proposition. At the conclusion the GenericSearch algorithm, a vertex $v \in V$ is marked as explored if and only if there is a path from s to v in G

Pf. [only if direction by induction on l, the number of loop iterations]

- P(l): for every v marked as explored in lth iteration there is a path from s to v
- base case: P(l) clearly holds when $l = 1$ (the path is s, v)
- induction step: show P($k + 1$) holds when P(l) holds for $l \leq k$
  - v is marked as explored due to an edge (v', v)
  - $s \rightsquigarrow v', v$ is a path from s to v since v' was marked as explored earlier

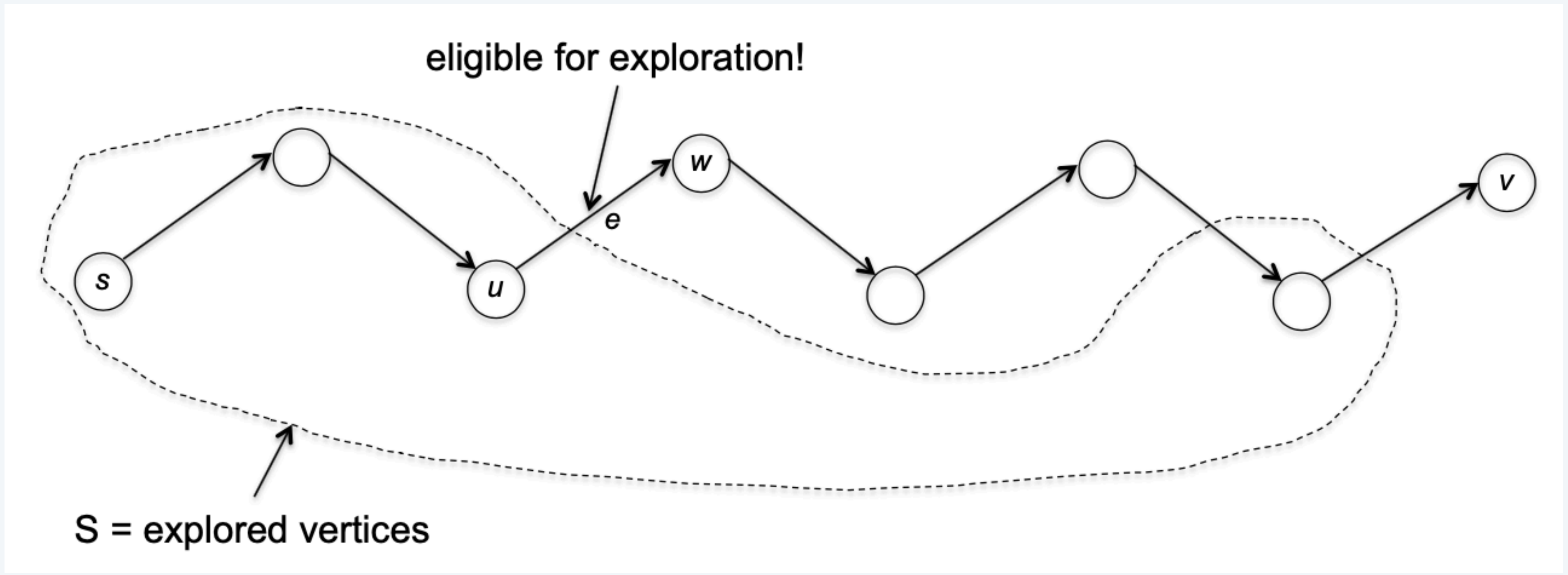
```
GENERICSEARCH $(G, s)$

mark $s$ as explored, all other vertices as unexplored
WHILE there is an edge $(v, w) \in E$ with $v$ explored and $w$ unexplored
    choose some such edge $(v, w)$
    mark $w$ as explored
```

Correctness of GenericSearch

Proposition. At the conclusion the GenericSearch algorithm, a vertex $v \in V$ is marked as explored if and only if there is a path from s to v in G

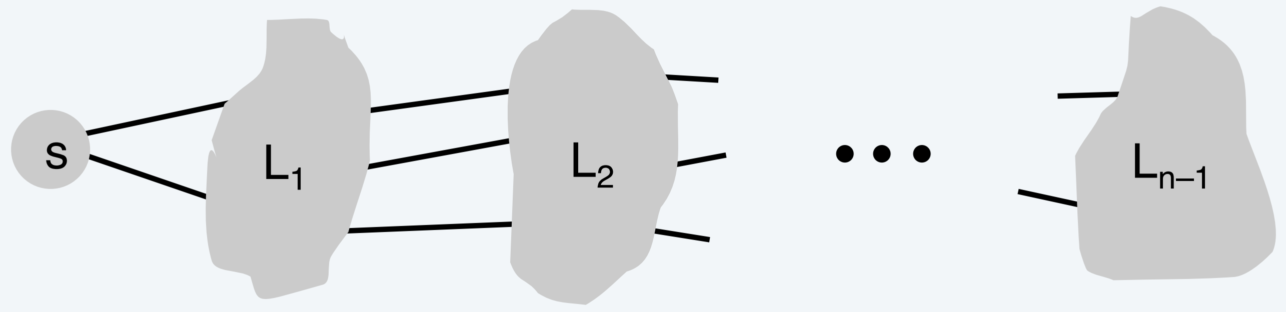
Pf. [if direction by contradiction]

- We claim that there is a path from s to v in G, but GenericSearch concludes with v marked as unexplored
- Let $S \subseteq V$ denote the vertices marked as explored
- Vertex s belongs to S but v does not
- At least one edge e of path $s \rightsquigarrow v$ has one endpoint u in S and the other w outside S, which is impossible!
  - GenericSearch would have explored at least one more vertex, rather than giving up



Breadth-first search

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.

BFS demo.

Breadth-first search

Input. A graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$

Postcondition. a vertex is reachable from s if and only if it is marked as explored

```
BFS $(G, s)$

mark $s$ as explored, all other vertices as unexplored
$Q \leftarrow$ a queue data structure, initialized with $s$
WHILE $Q$ is not empty
    remove the vertex from the front of $Q$, call it $v$
    FOR each edge $(v, w)$ in $v$'s adjacency-list
        IF $w$ is unexplored
            mark $w$ as explored // visiting $w$ now!
            add $w$ to the end of $Q$
```

Correctness of BFS

```
GENERICSEARCH $(G, s)$

mark $s$ as explored, all other vertices as unexplored
WHILE there is an edge $(v, w) \in E$ with $v$ explored and $w$ unexplored
    choose some such edge $(v, w)$
    mark $w$ as explored
```

BFS is a special case of GenericSearch: it chooses (v, w) for which v was discovered **the earliest**, breaking ties among v's eligible edges according to **their order** in v's adjacency-list

```
BFS $(G, s)$

mark $s$ as explored, all other vertices as unexplored
$Q \leftarrow$ a queue data structure, initialized with $s$
WHILE $Q$ is not empty
    remove the vertex from the front of $Q$, call it $v$
    FOR each edge $(v, w)$ in $v$'s adjacency-list
        IF $w$ is unexplored
            mark $w$ as explored
            add $w$ to the end of $Q$
```

Running time of BFS

BFS (G, s)

```
1 mark $s$ as explored, all other vertices as unexplored
2 $Q \leftarrow$ a queue data structure, initialized with $s$
3 WHILE $Q$ is not empty
4     remove the vertex from the front of $Q$, call it $v$
5     FOR each edge $(v, w)$ in $v$'s adjacency-list
6         IF $w$ is unexplored
7             mark $w$ as explored
8             add $w$ to the end of $Q$
```

Line 1: $O(n)$

Line 2: $O(1)$

Lines 3-4: $O(n)$

- no vertex is explored twice
- each dequeue operation takes $O(1)$ time

Lines 5-6: $O(m)$

- each edge (v, w) is processed at most twice: exploring v and exploring w

Lines 7-8: $O(n)$ (same reason as Lines 3-4)

Running time: $O(m + n)$

Depth-first search

DFS intuition. More aggressive than BFS, always exploring from the most recently discovered vertex and backtracking only when necessary (like exploring a maze).

DFS demo.

Depth-first search

Input. A graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$

Postcondition. a vertex is reachable from s if and only if it is marked as explored

```
DFS ($G$, $s$) (Iterative Version)

mark all vertices as unexplored
$S \leftarrow$ a stack data structure, initialized with $s$
WHILE $S$ is not empty
    remove (“pop”) the vertex $v$ from the front of $S$
    IF $v$ is unexplored
        mark $v$ as explored // visiting $v$ now
        FOR each edge $(v, w)$ in $v$’s adjacency-list
            add (“push”) $w$ to the front of $S$
```

Correctness of DFS

```
GENERICSEARCH $(G, s)$

mark $s$ as explored, all other vertices as unexplored
WHILE there is an edge $(v, w) \in E$ with $v$ explored and $w$ unexplored
    choose some such edge $(v, w)$
    mark $w$ as explored
```

DFS is a special case of GenericSearch:
it chooses (v, w) for which v was discovered **most recently**,
breaking ties among v's eligible edges according to
their reverse order in v's adjacency-list

```
DFS $(G, s)$ (Iterative Version)

mark all vertices as unexplored
$S \leftarrow$ a stack data structure, initialized with $s$
WHILE $S$ is not empty
    remove ("pop") the vertex $v$ from the front of $S$
    IF $v$ is unexplored
        mark $v$ as explored
        FOR each edge $(v, w)$ in $v$'s adjacency-list
            add ("push") $w$ to the front of $S$
```

Running time of DFS

```
DFS $(G, s)$ (Iterative Version)
1 mark all vertices as unexplored
2 $S \leftarrow$ a stack data structure, initialized with $s$
3 WHILE $S$ is not empty
4     remove ("pop") the vertex $v$ from the front of $S$
5     IF $v$ is unexplored
6         mark $v$ as explored
7         FOR each edge $(v, w)$ in $v$'s adjacency-list
8             add ("push") $w$ to the front of $S$
```

Line 1: $O(n)$

Line 2: $O(1)$

~~Lines 3-4: $O(n)$~~

- ~~no vertex is explored twice~~
- ~~each pop operation takes $O(1)$ time~~

Lines 6: $O(n)$

Lines 7-8: $O(m)$

- each edge (v, w) is processed at most twice: exploring v and exploring w

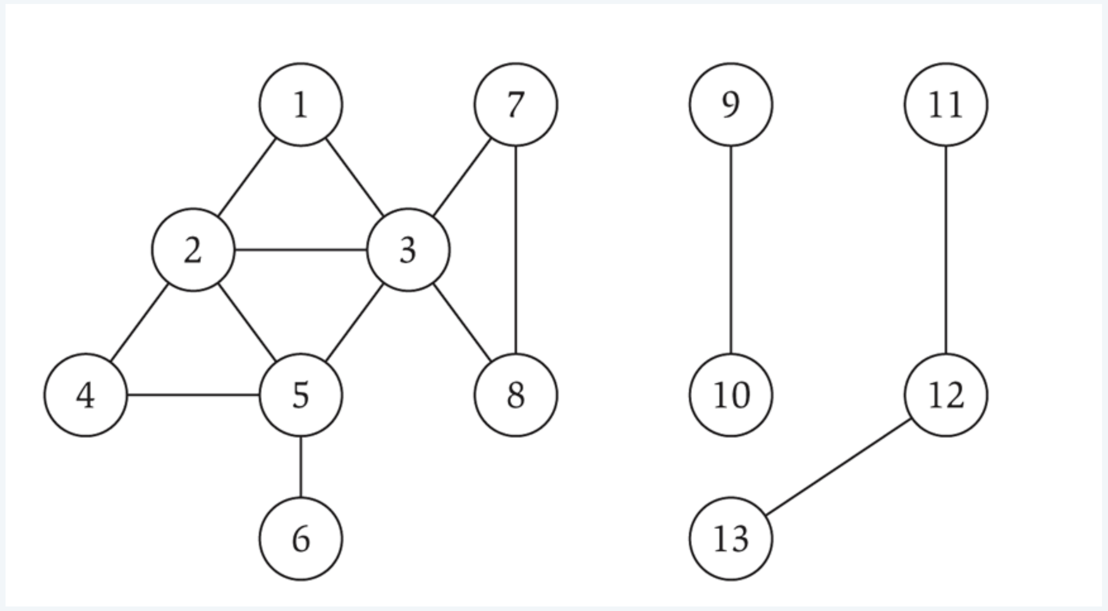
Lines 3-5: $O(m)$

- push $O(m)$ vertices to S, and each pop/push operation takes $O(1)$ time

Running time: $O(m + n)$

Connected component

Connected component. Find all nodes reachable from s.



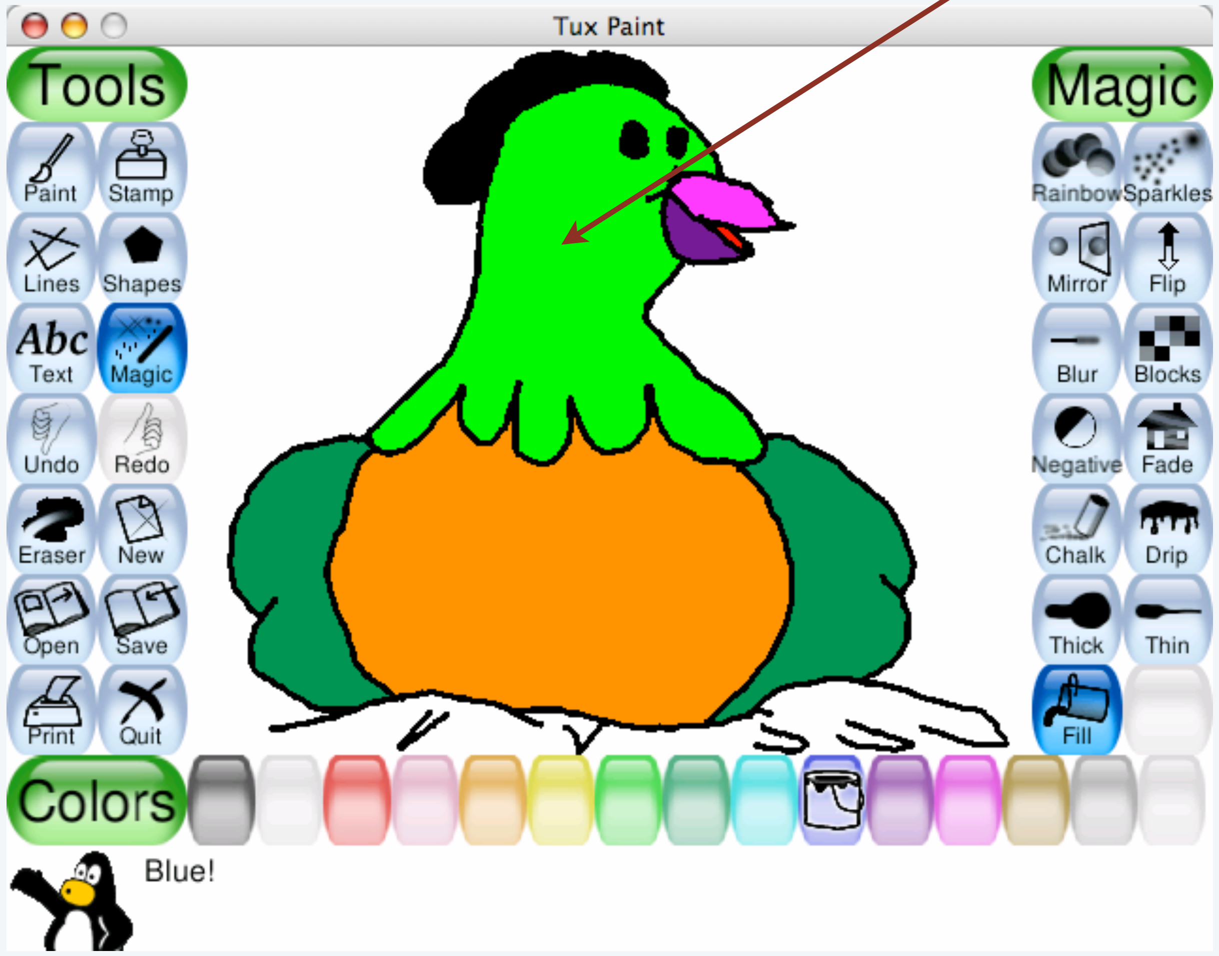
Connected component containing node $1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$.

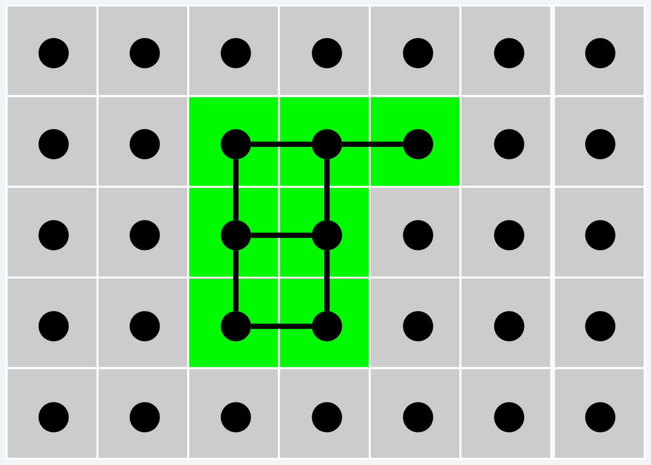
Flood fill (填色游戏)

Flood fill. 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.

recolor lime green blob to blue



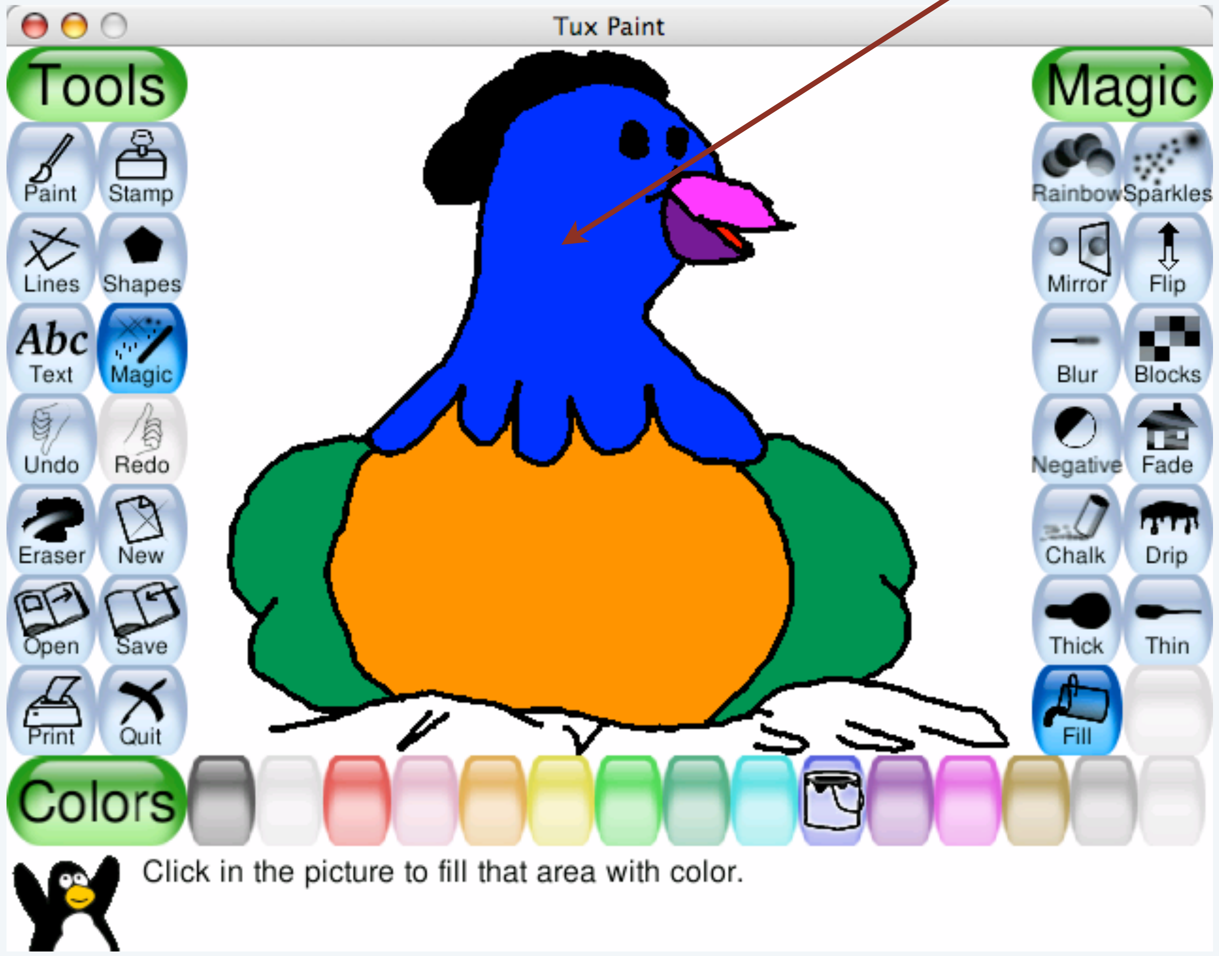


Flood fill (填色游戏)

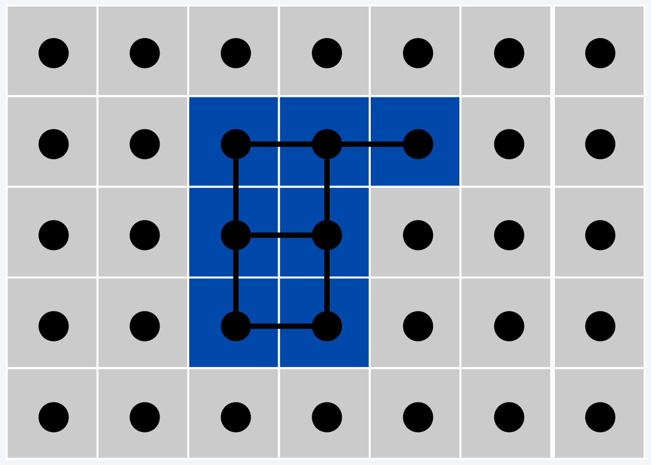
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue



Click in the picture to fill that area with color.



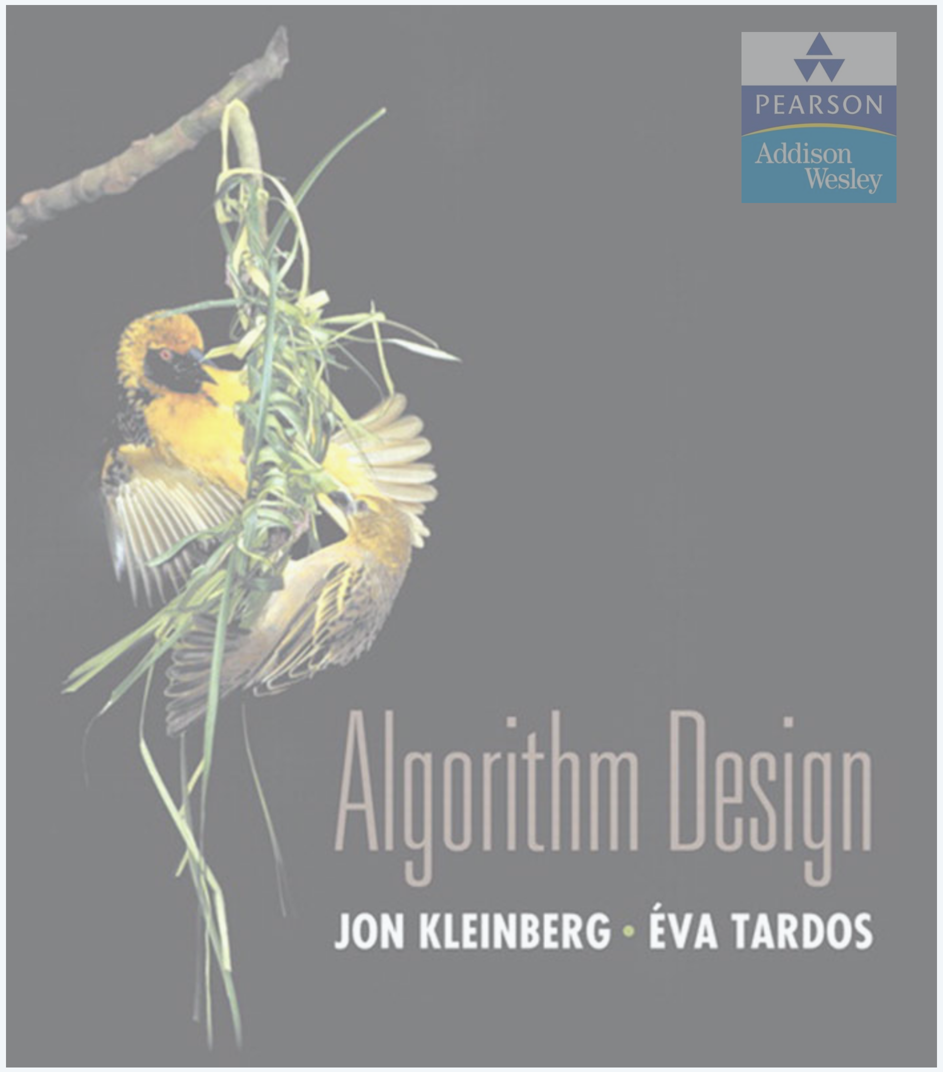
Undirected connected components

UCC (G)

Input. A undirected graph $G = (V, E)$ in adjacency-list representation, with $V = \{1, 2, \cdots, n\}$

Postcondition. For every $u, v \in V$, $cc(u) = cc(v)$ if and only if u, v are in the same connected component

```
mark all vertices as unexplored
$numCC \leftarrow 0$
FOR $i \leftarrow 1$ to $n$ // try all vertices
    If $i$ is unexplored // avoid redundancy
    $numCC \leftarrow numCC + 1$ // new component
    // call BFS starting at $i$
    $Q \leftarrow$ a queue data structure, initialized with $i$
    WHILE $Q$ is not empty
        remove the vertex from the front of $Q$, call it $v$
        $cc(v) \leftarrow numCC$
        FOR each edge $(v, w)$ in $v$'s adjacency-list
            IF $w$ is unexplored
                mark $w$ as explored
                add $w$ to the end of $Q$
```



3. Graphs

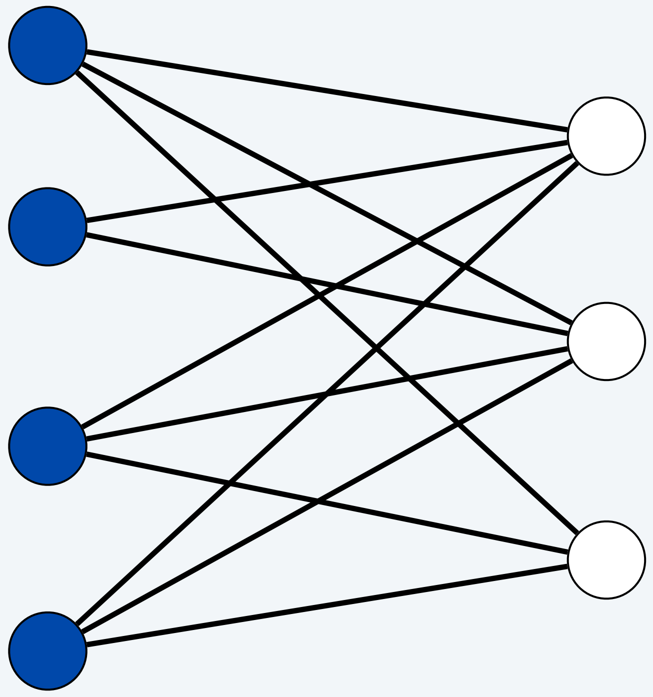
- *basic definitions and applications*
- *graph connectivity and graph traversal*
- *testing bipartiteness*
- *connectivity in directed graphs*
- *DAGs and topological ordering*

Bipartite graphs

Def. 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.

Applications.

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



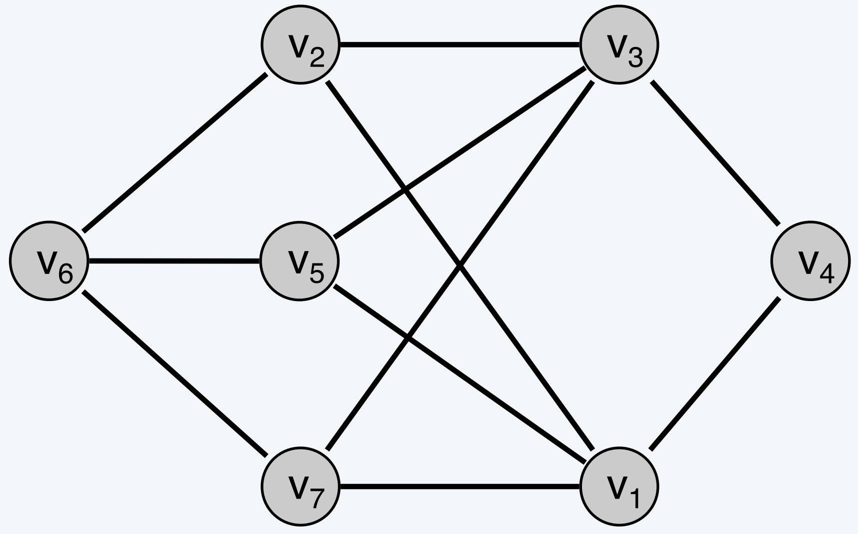
a bipartite graph

Testing bipartiteness

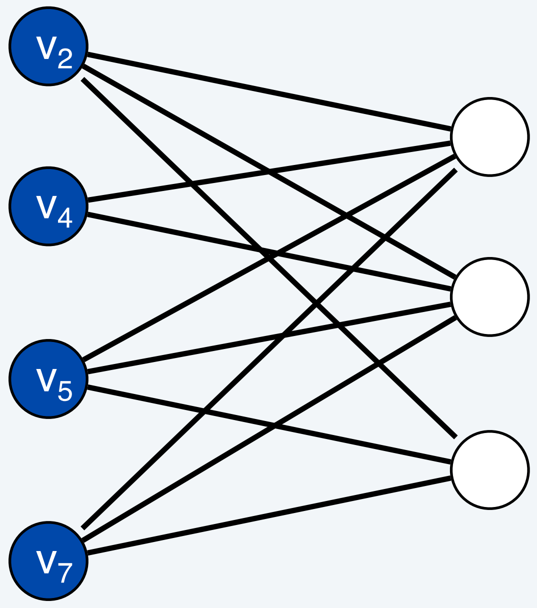
Many graph problems become:

- 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.



a bipartite graph G

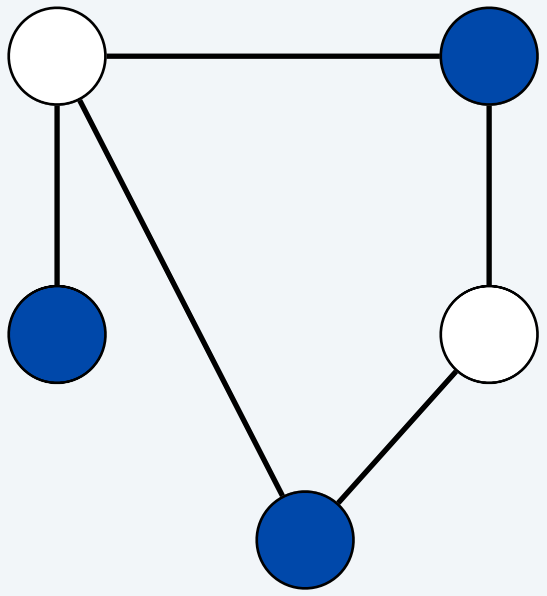


another drawing of G

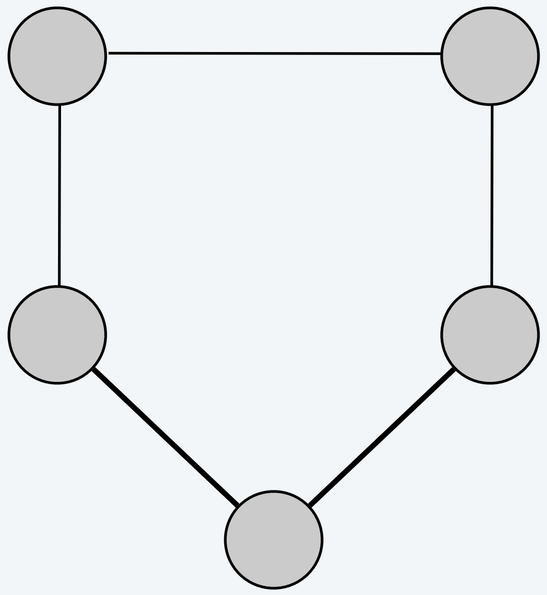
An obstruction to bipartiteness

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

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



bipartite
(2-colorable)



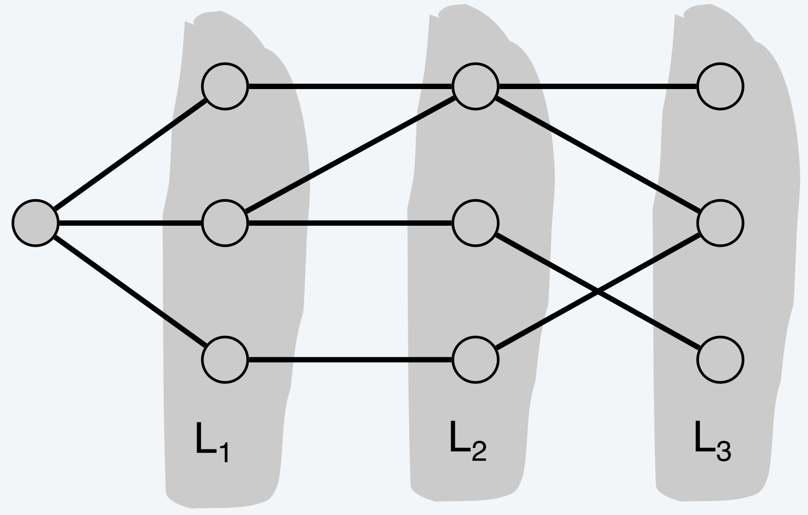
not bipartite
(not 2-colorable)

Bipartite graphs

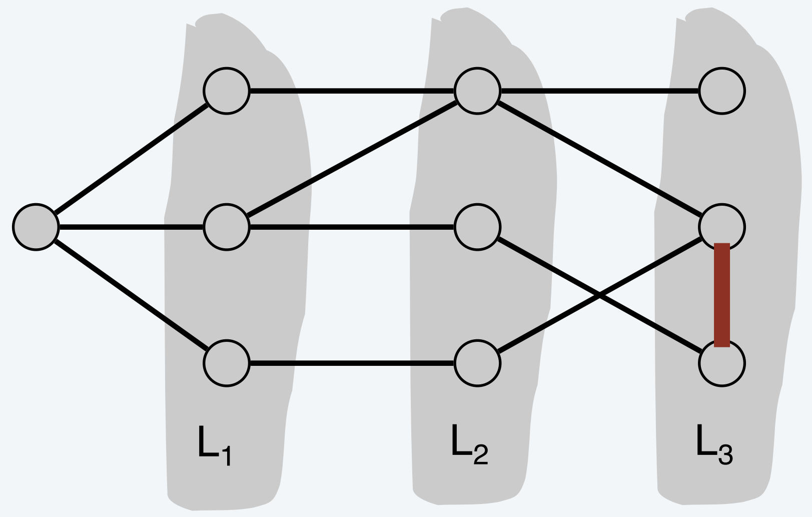
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(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).



Case (i)



Case (ii)

Bipartite graphs

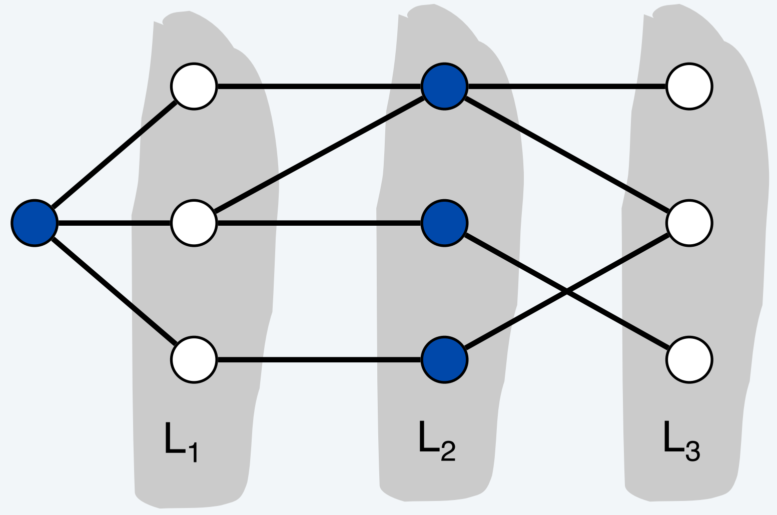
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(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).

Pf. (i)

- Suppose no edge joins two nodes in same layer.
- By BFS property, each edge joins two nodes in adjacent levels.
- Bipartition: white = nodes on odd levels, blue = nodes on even levels.



Case (i)

Bipartite graphs

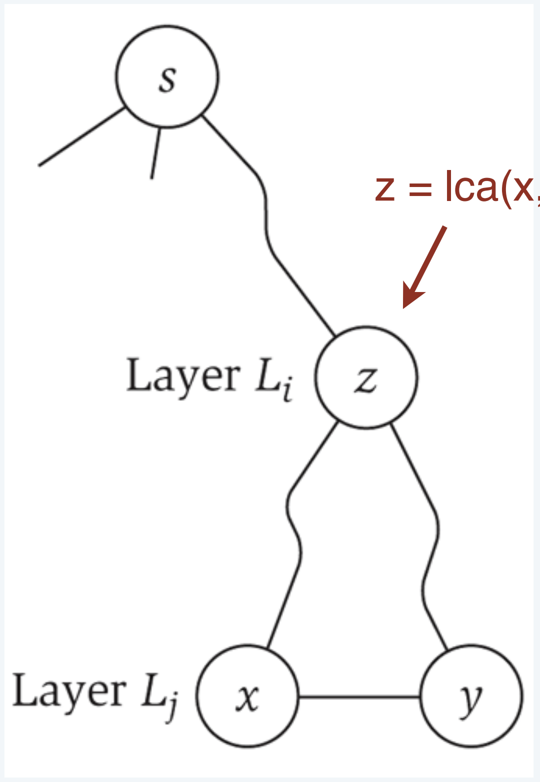
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(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).

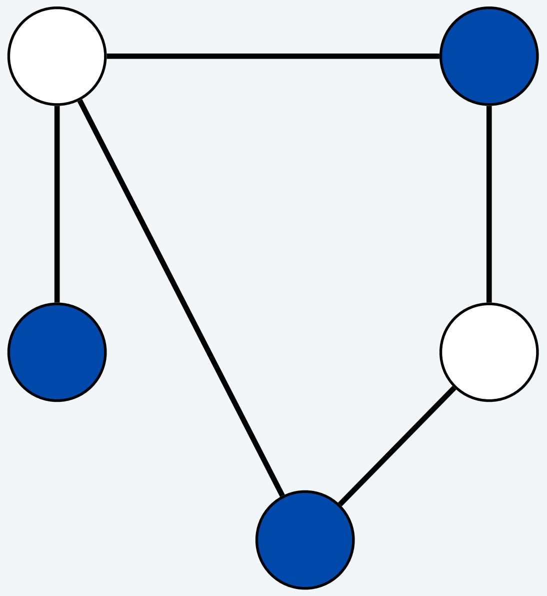
Pf. (ii)

- 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 $\underbrace{1}_{\text{(x, y)}} + \underbrace{(j-i)}_{\text{path from y to z}} + \underbrace{(j-i)}_{\text{path from z to x}}$, which is odd. ▪

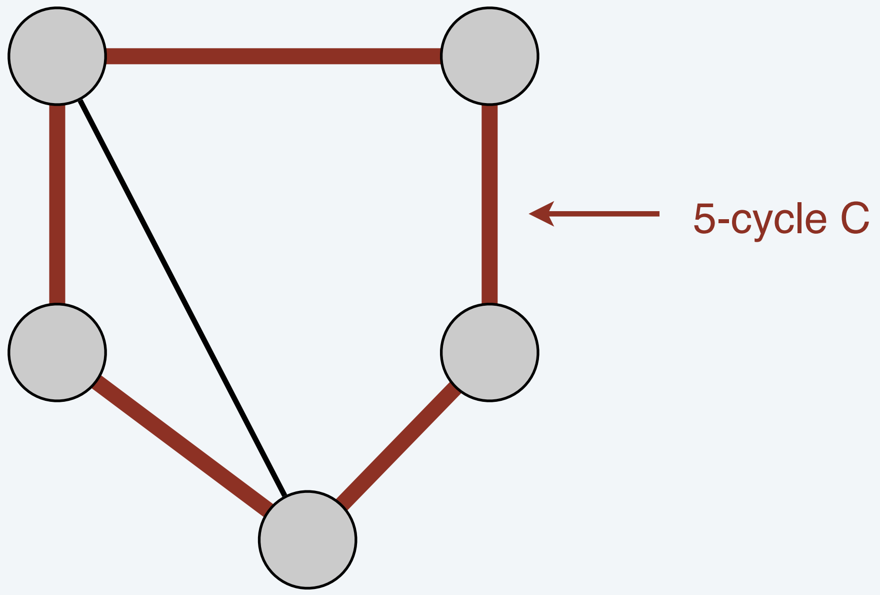


The only obstruction to bipartiteness

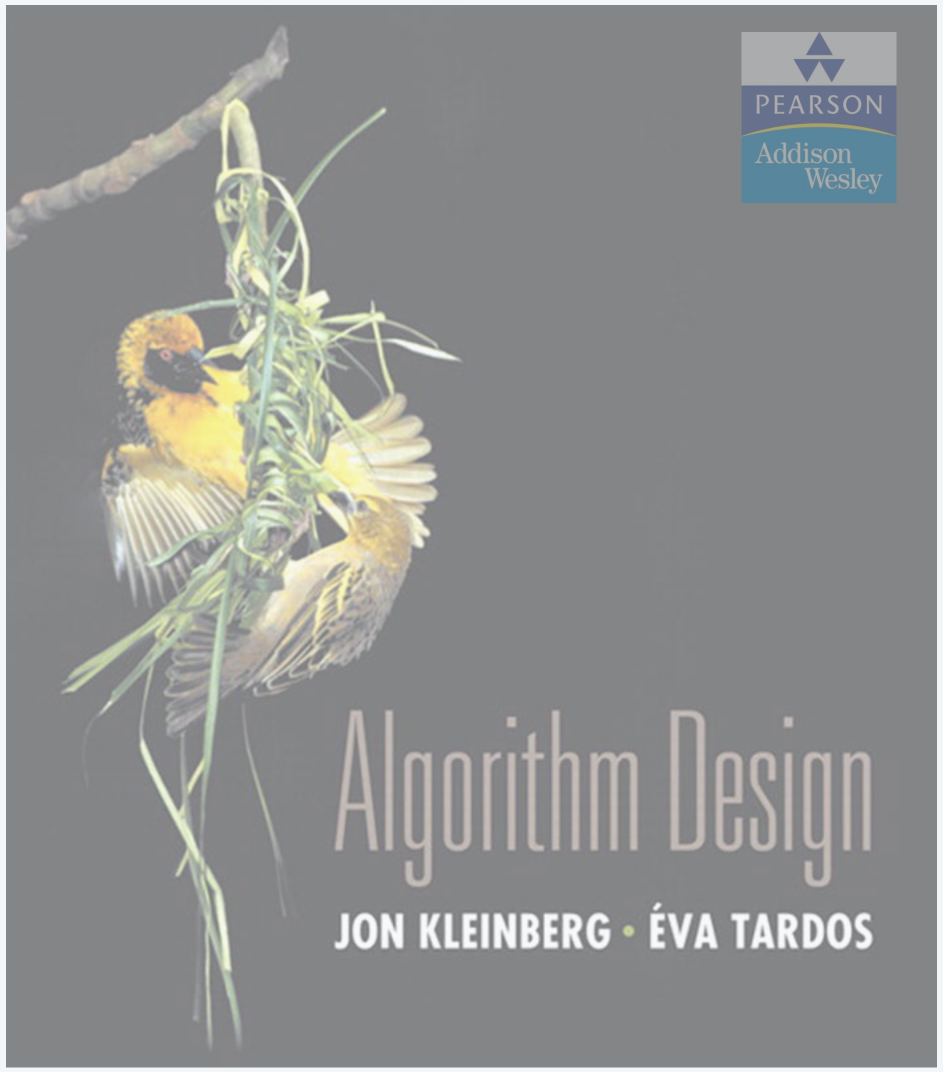
Corollary. A graph G is bipartite iff it contains no odd-length cycle.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)



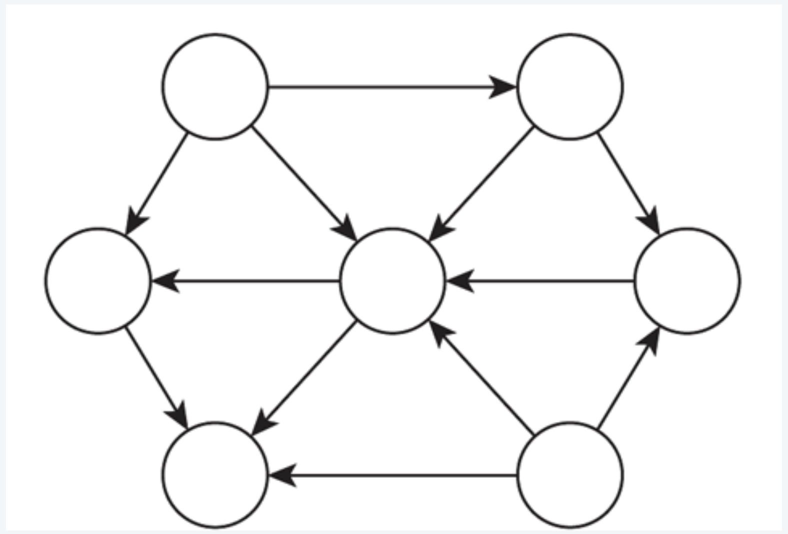
3. Graphs

- *basic definitions and applications*
- *graph connectivity and graph traversal*
- *testing bipartiteness*
- *connectivity in directed graphs*
- *DAGs and topological ordering*

Directed graphs

Notation. $G = (V, E)$.

- Edge (u, v) leaves node u and enters node v.



Ex. 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.

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

Graph search

Directed reachability. Given a node s, find all nodes reachable from s.

Directed s↝t shortest path problem. Given two nodes s and t, what is the length of a shortest path from s to 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.

Strong connectivity

Def. 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. A graph is strongly connected if every pair of nodes is mutually reachable.

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

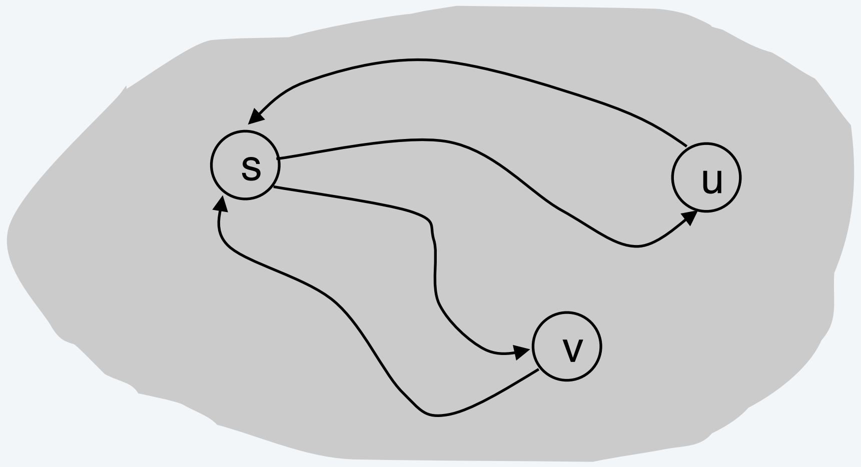
Pf. $\Rightarrow$ Follows from definition.

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

Path from v to u: concatenate $v \rightsquigarrow s$ path with $s \rightsquigarrow u$ path. ▪

$\nwarrow$

ok if paths overlap

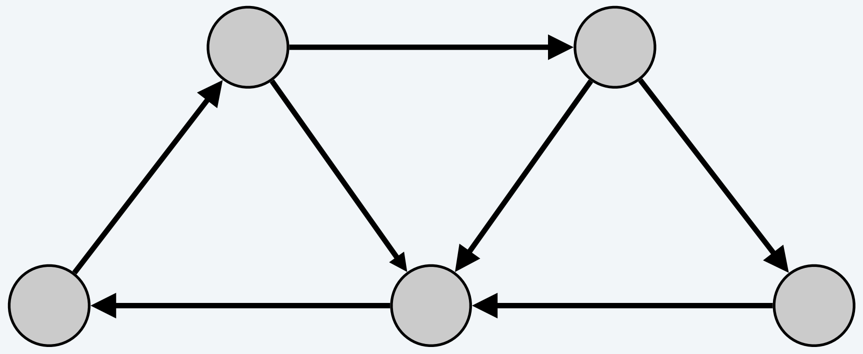


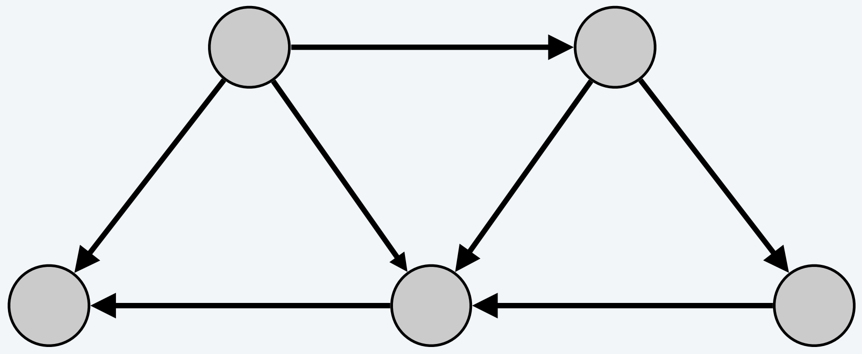
Strong connectivity: algorithm

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

Pf.

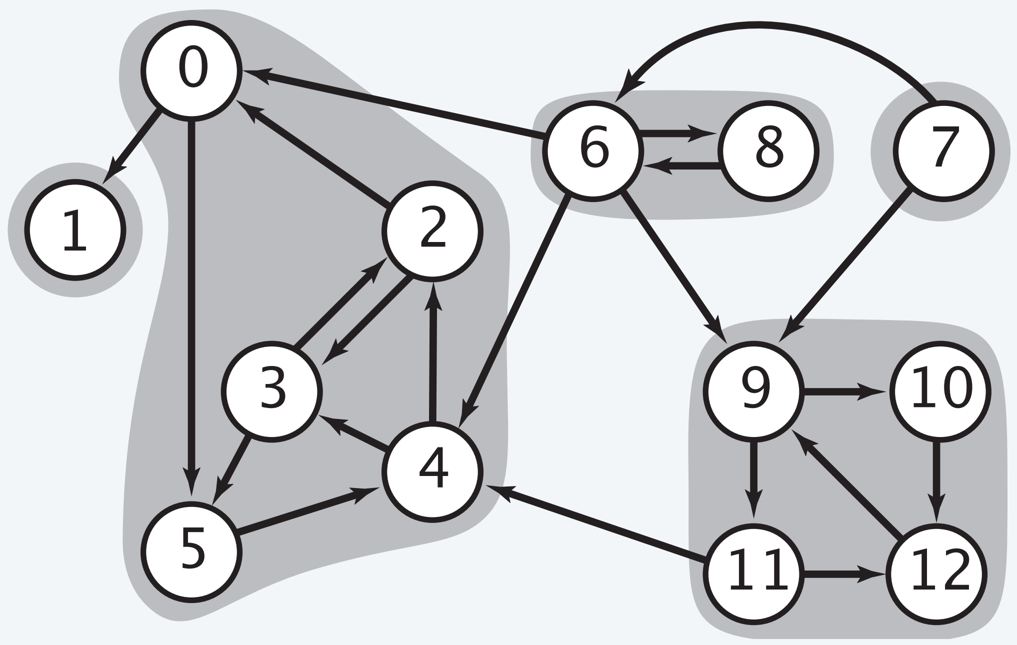
- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in $G^{reverse}$. reverse orientation of every edge in G
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. ▪


strongly connected

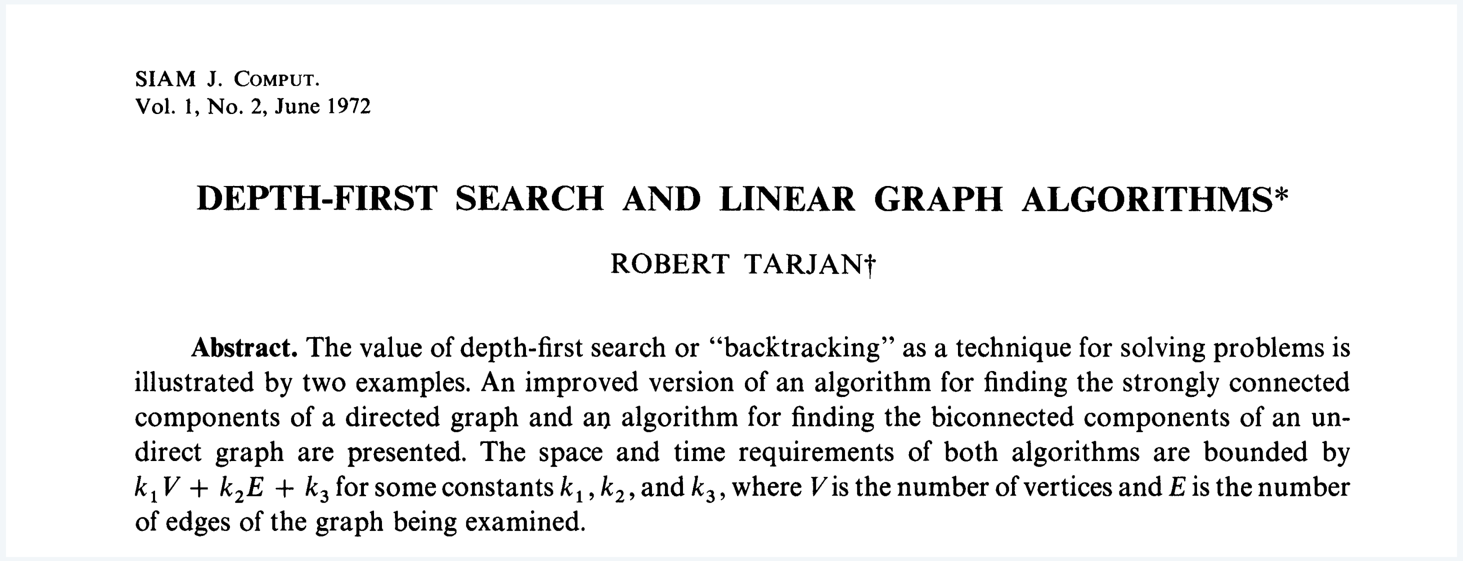

not strongly connected

Strong components

Def. A strong component is a maximal subset of mutually reachable nodes.



Theorem. [Tarjan 1972] Can find all strong components in $O(m + n)$ time.

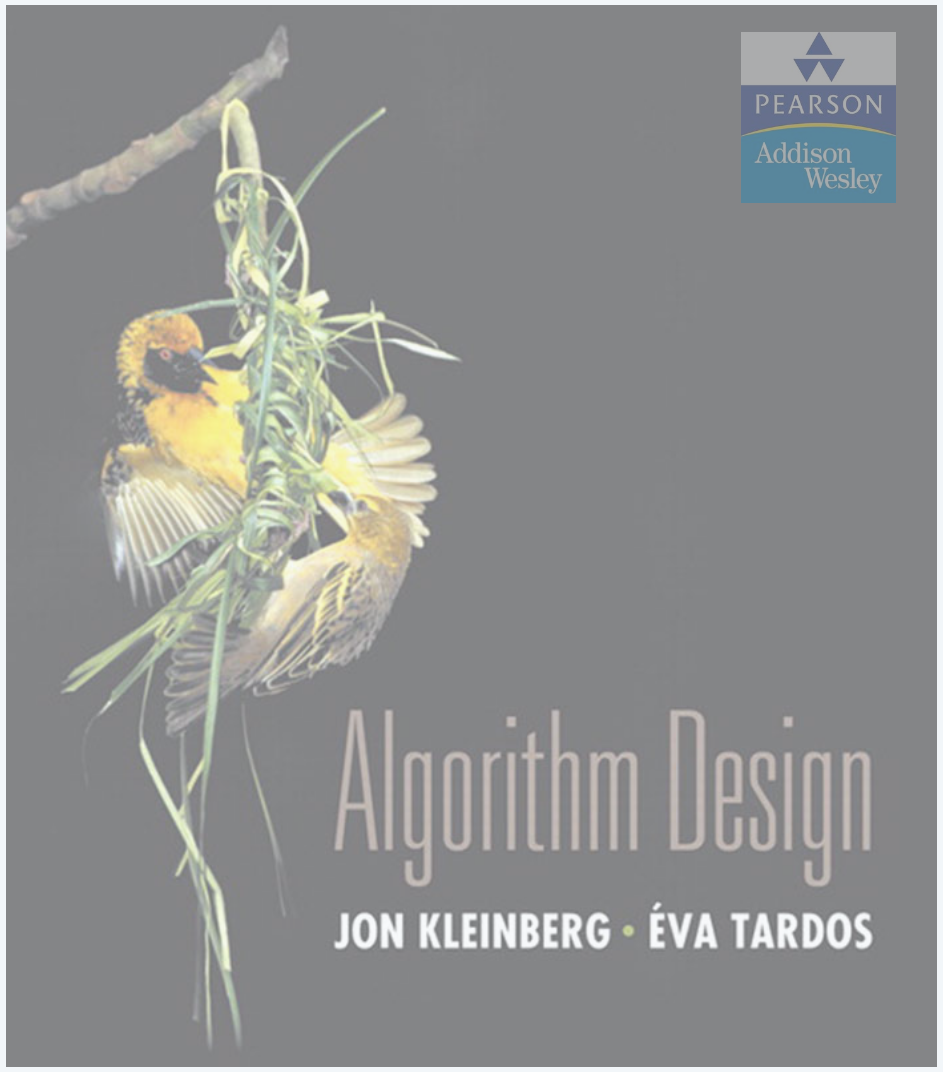


SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.



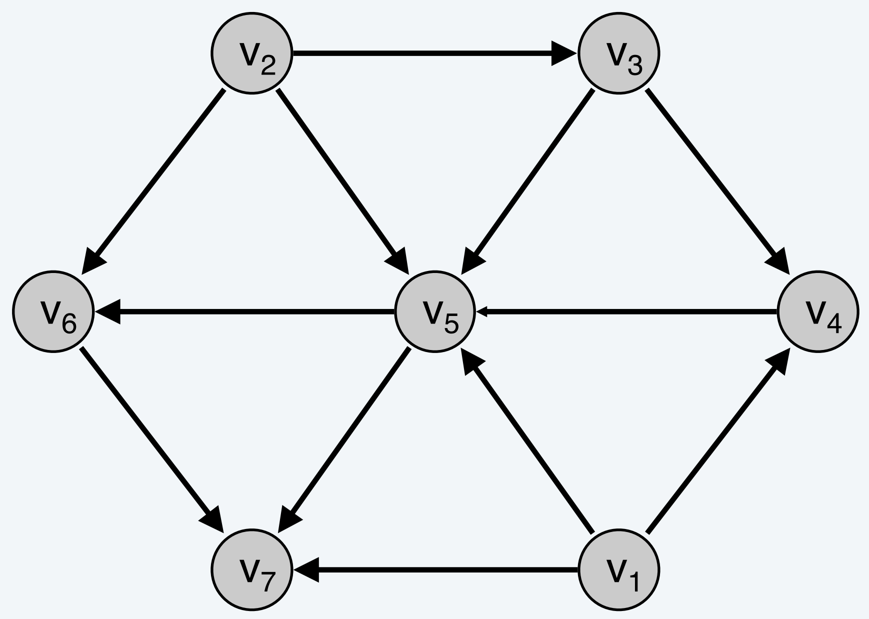
3. Graphs

- *basic definitions and applications*
- *graph connectivity and graph traversal*
- *testing bipartiteness*
- *connectivity in directed graphs*
- *DAGs and topological ordering*

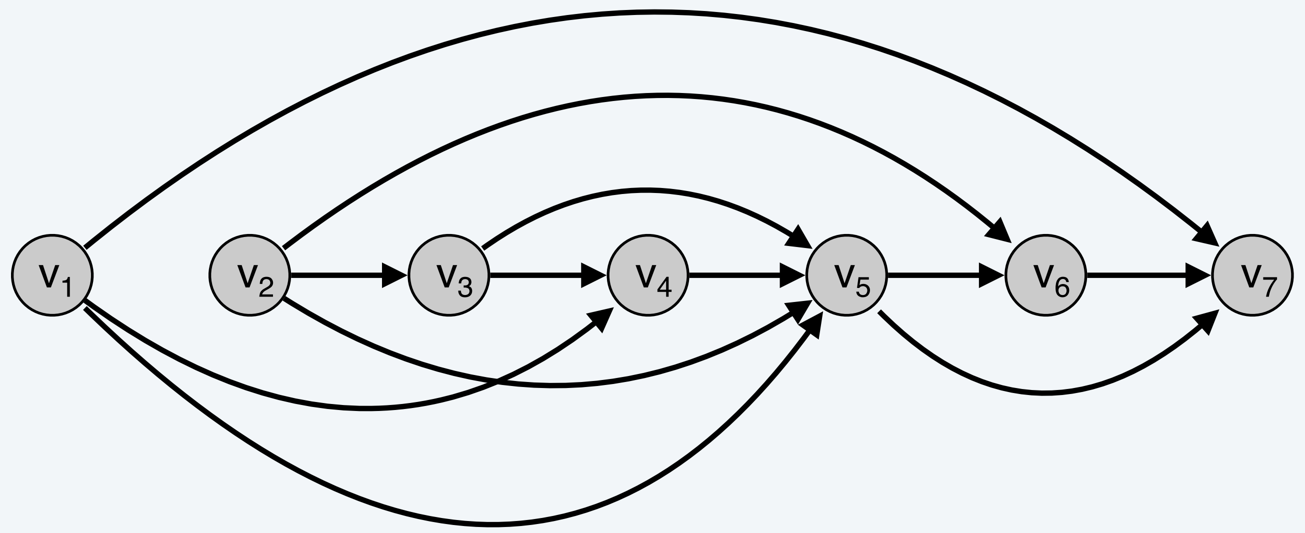
Directed acyclic graphs

Def. A DAG is a directed graph that contains no directed cycles.

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



a DAG



a topological ordering

Precedence constraints

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

Applications.

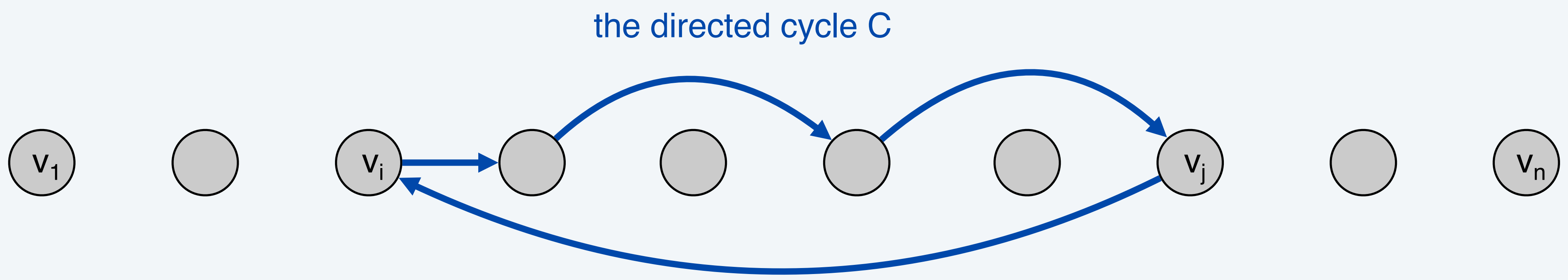
- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j.
- Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

Directed acyclic graphs

Lemma. If G has a topological order, then G is a DAG.

Pf. [by contradiction]

- Suppose that G has a topological order $v_1, v_2, \ldots, v_n$ 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 $v_1, v_2, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪



the supposed topological order: $v_1, \ldots, v_n$

Directed acyclic graphs

Lemma. If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

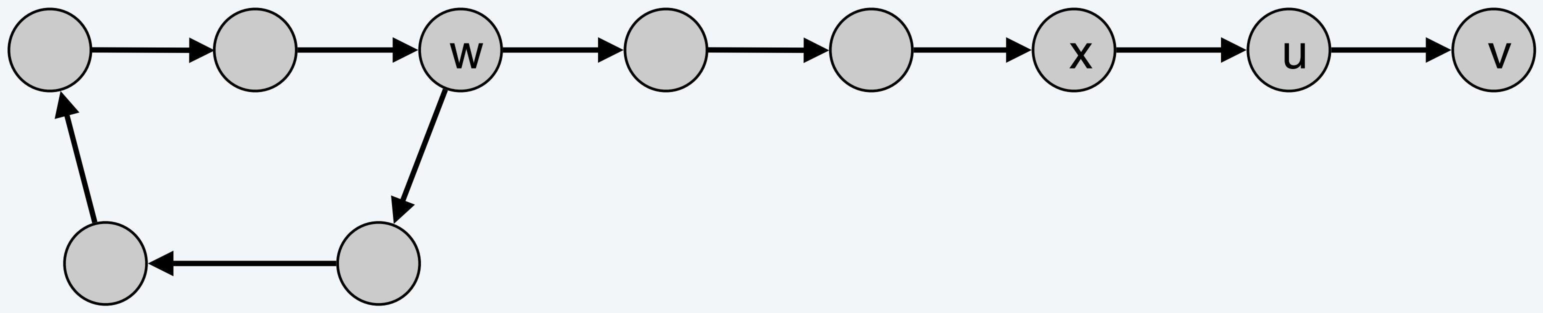
Q. If so, how do we compute one?

Directed acyclic graphs

Lemma. If G is a DAG, then G has a node with no entering 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. ▪

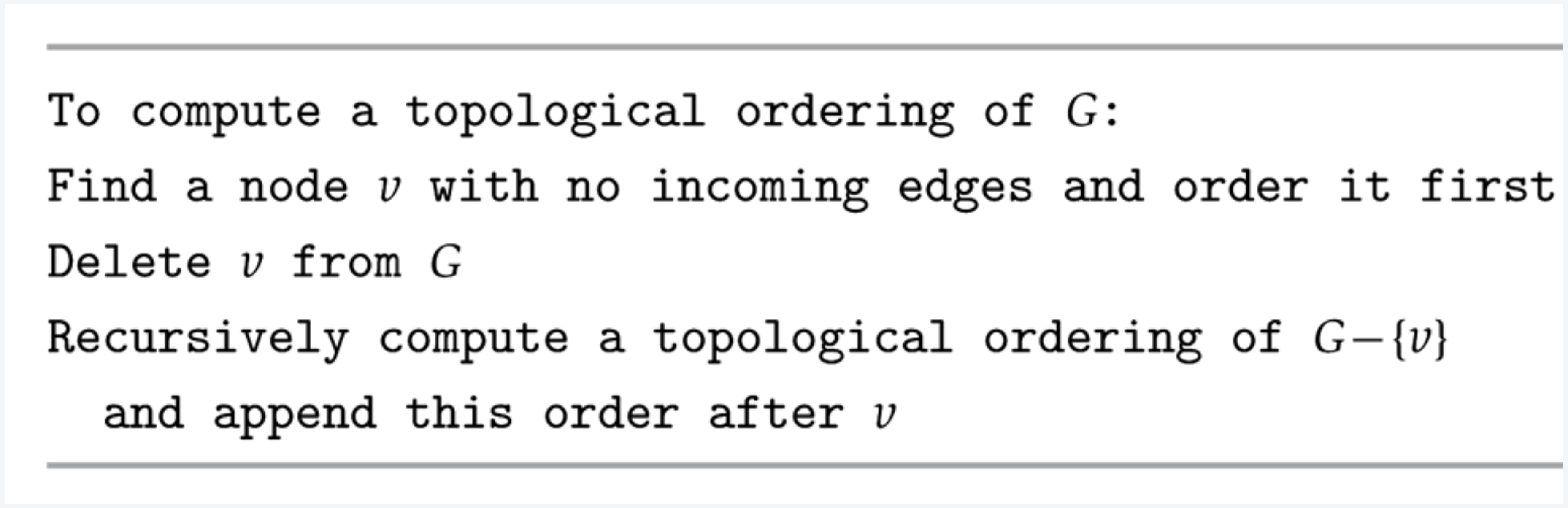


Directed acyclic graphs

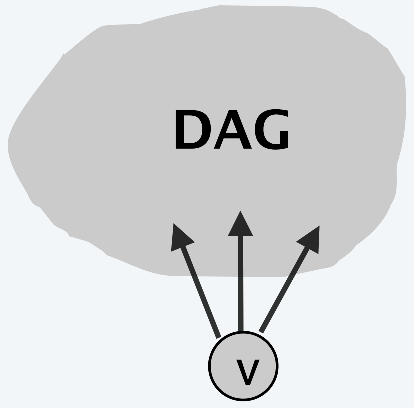
Lemma. If G is a DAG, then G has a topological ordering.

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. ▪



```
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 $v$
```



Topological sorting algorithm: running time

Theorem. 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 ▪