Algorithm II

3. Graphs

WU Xiaokun 吴晓堃

xkun.wu [at] gmail

Graphs in Discrete Math

Computer deals with discrete mathematics.

- core subject: combinatorial structures
- graphs: fundamental, expressive

Content

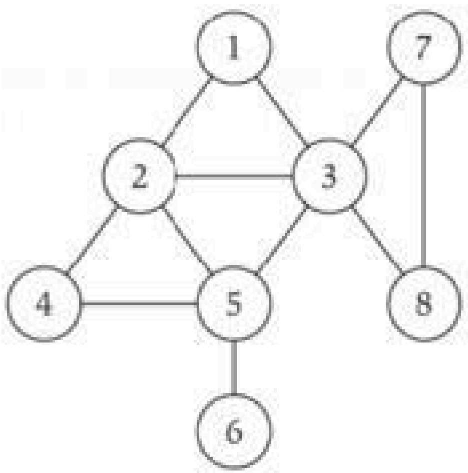
- Basic Definitions
- Graph Connectivity and Graph Traversal
- Testing Bipartiteness
- Connectivity in Directed Graphs
- DAGs and Topological Ordering

Basic Definitions

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: $n = |V|, m = |E|$.



$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$E =$
$\{1\text{–}2, 1\text{–}3, 2\text{–}3, 2\text{–}4, 2\text{–}5, 3\text{–}5, 3\text{–}7, 3\text{–}8, 4\text{–}5, 5\text{–}6,$
$m = 11, n = 8$

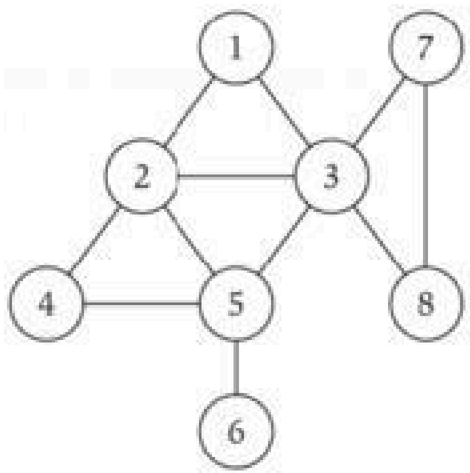
Examples of Graphs

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	network hub	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.

- Symmetry: two representations of each edge.
- Space proportional to n^2.
- Check if (u, v) is an edge: $\Theta(1)$ time.
- Identify all edges: $\Theta(n^2)$ time.

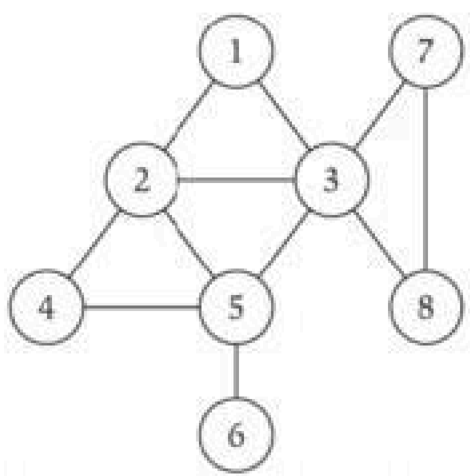


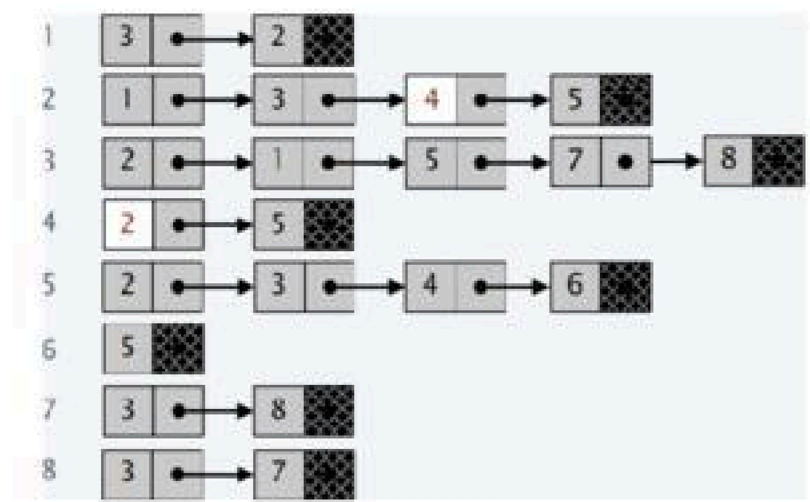
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph representation: adjacency lists

Adjacency lists. Node-indexed array of lists.

- Symmetry: two representations of each edge.
- Space is $\Theta(m + n)$.
- Check if (u, v) is an edge: $O(degree(u))$ time.
- Identify all edges: $\Theta(m + n)$ time.





Graph representation: space requirement

Degree n_v of a node v: the number of *incident* edges it has.

Sum of the degrees. $\sum_{v \in V} n_v = 2m$.
Pf. Each edge $e = (v, w)$ contributes exactly twice to this sum.

Graph representation: space requirement

Degree n_v of a node v: the number of *incident* edges it has.

Sum of the degrees. $\sum_{v \in V} n_v = 2m$.
Pf. Each edge $e = (v, w)$ contributes exactly twice to this sum.

Theorem. Adjacency matrix representation of a graph requires $O(n^2)$ space;
Adjacency list representation requires only $O(m + n)$ space.

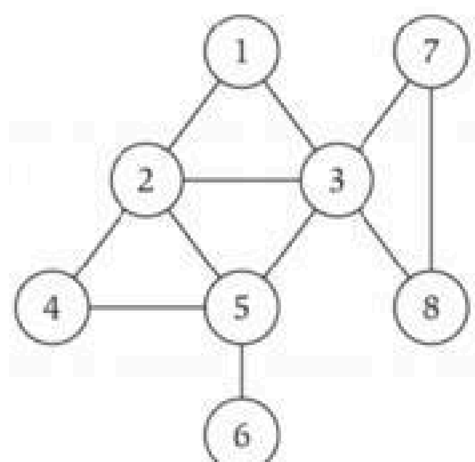
- since $m \leq n^2$, the bound $O(m + n)$ is never worse than $O(n^2)$
  - adjacency list is a natural representation for exploring graphs.

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.

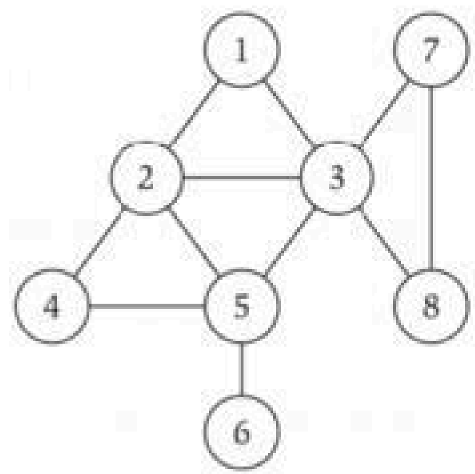
Def. An undirected graph is **connected** if for every pair of nodes u and v, there is a path between them.



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).

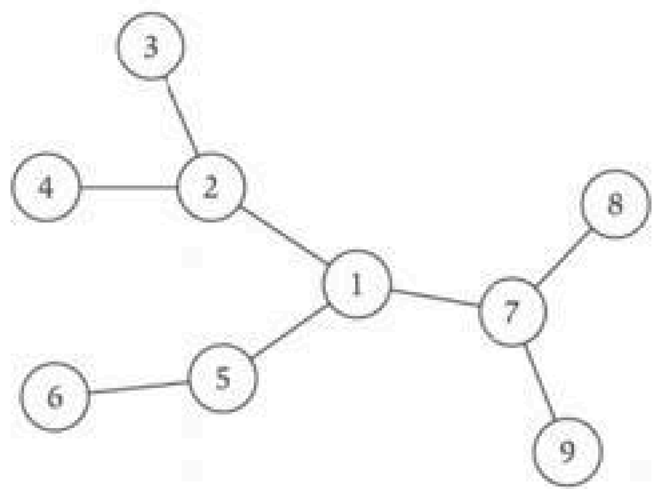


Trees

Def. An undirected graph is a **tree** if it is connected and contains no cycle.

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

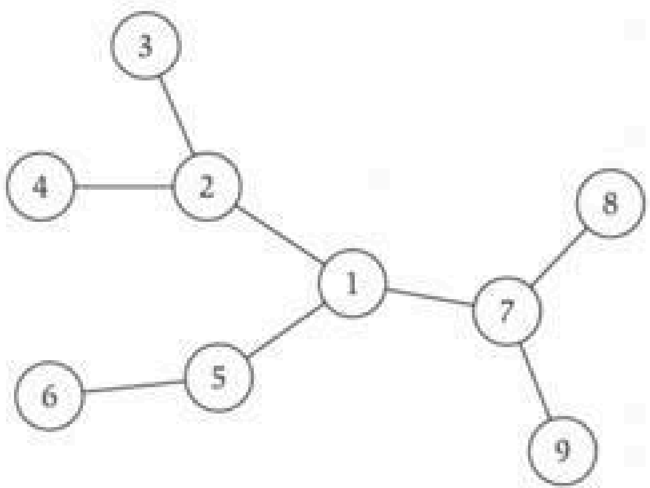
- G is connected.
- G does not contain a cycle.
- G has $n-1$ edges.

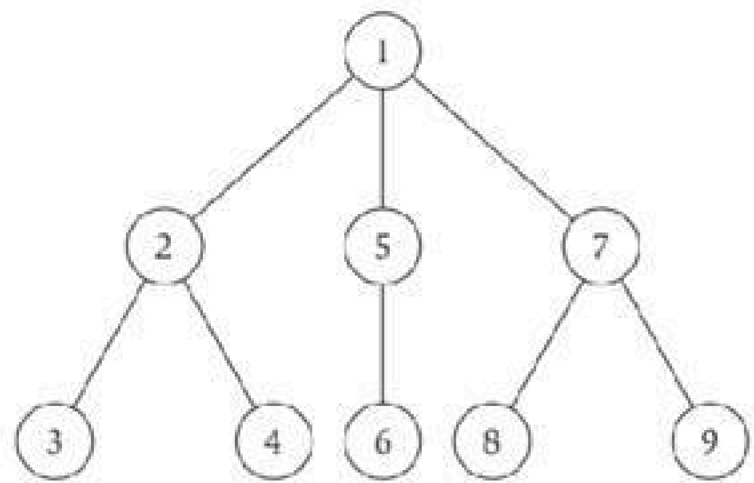


Rooted trees

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

Importance. Models hierarchical structure.



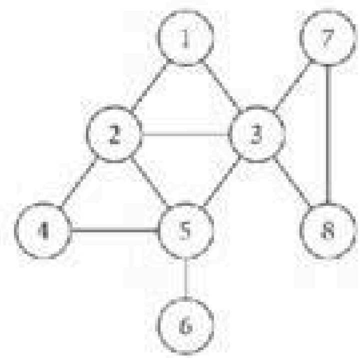


Graph Connectivity and Graph Traversal

Connectivity

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

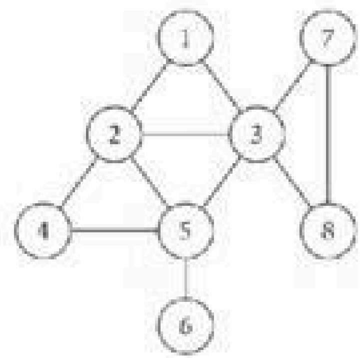
s-t shortest path problem. Given two nodes s and t, what is the length of a shortest path between them?



Connectivity

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

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



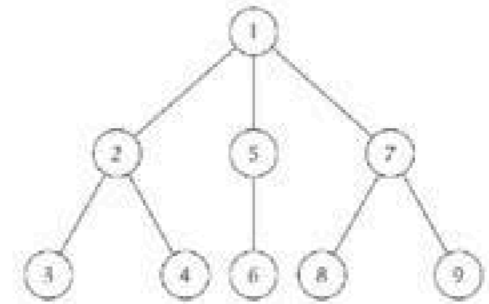
Applications.

- Maze traversal, map navigation, etc.

Breadth-first search (BFS)

BFS intuition. Start at root s and "flood" the graph.

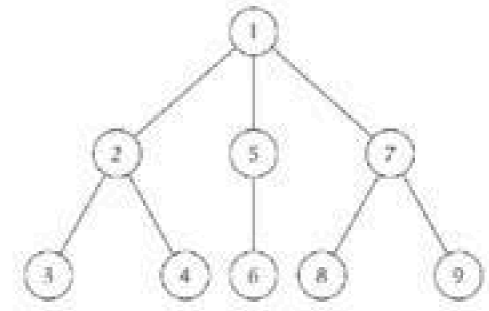
- Explore outward from s in all possible directions,
- Adding nodes one "layer" at a time.



Breadth-first search (BFS)

BFS intuition. Start at root s and “flood” the graph.

- Explore outward from s in all possible directions,
- Adding nodes one “layer” at a time.



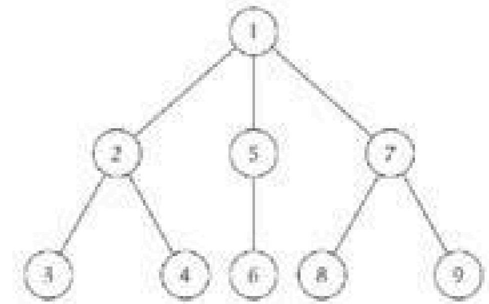
BFS algorithm.

- L_0: $\{s\}$.
- L_1: all neighbors of L_0.
- L_{i+1}: all nodes that do not belong to any earlier layer, and that have an edge to a node in L_i.

Breadth-first search (BFS)

BFS intuition. Start at root s and "flood" the graph.

- Explore outward from s in all possible directions,
- Adding nodes one "layer" at a time.



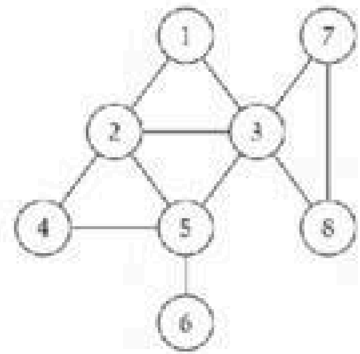
BFS algorithm.

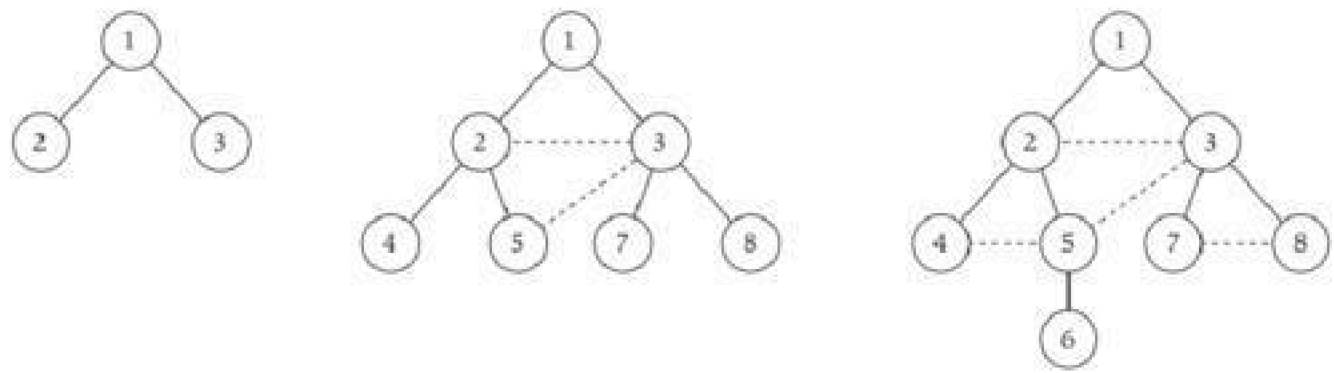
- L_0: $\{s\}$.
- L_1: all neighbors of L_0.
- L_{i+1}: all nodes that do not belong to any earlier layer, and that have an edge to a node in L_i.

Theorem. For each $i \geq 1$, 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.

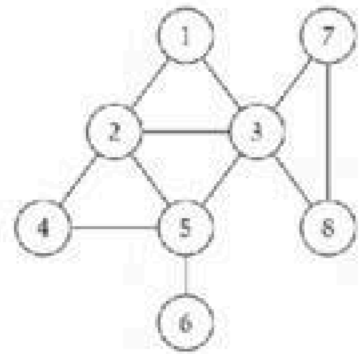
- produce a tree with root s

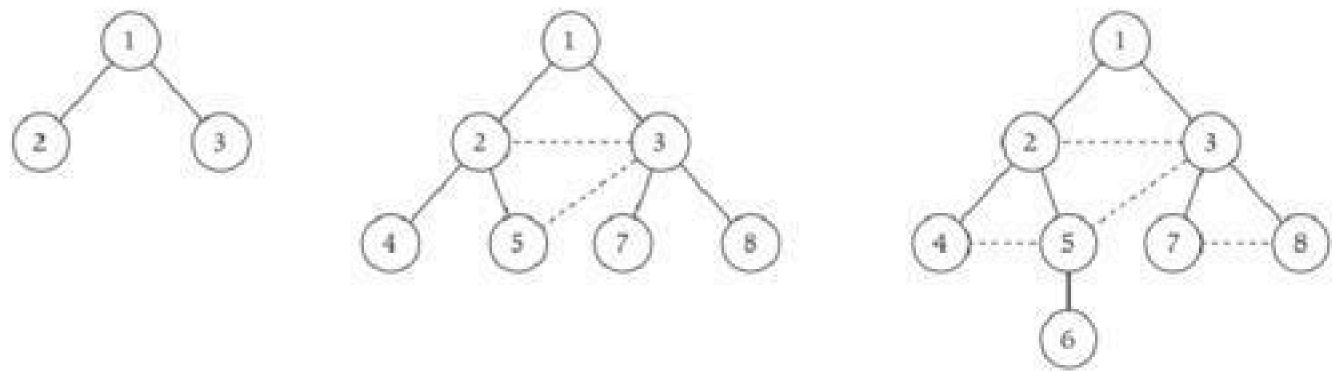
BFS tree





BFS tree





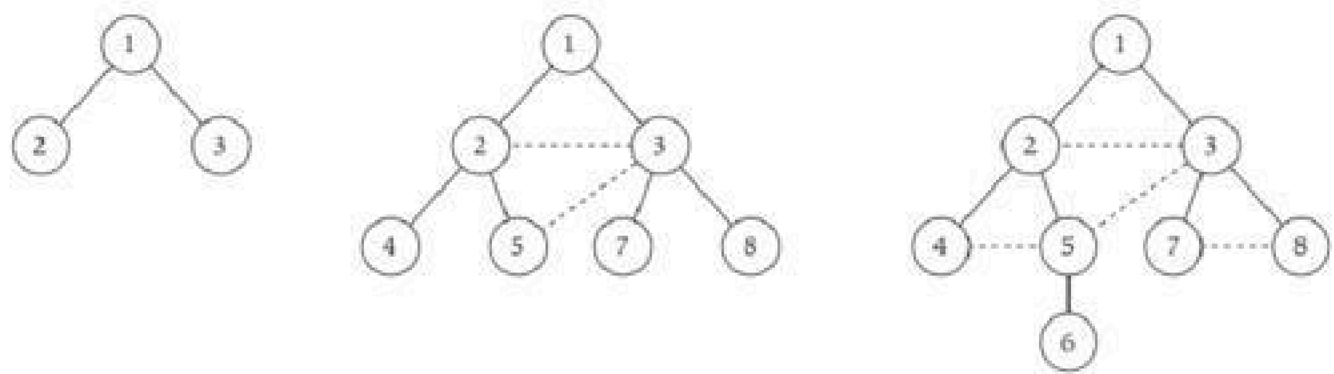
Note: non-tree edges all either connected nodes in the same layer, or connected nodes in adjacent layers.

BFS tree property

Property. Let T be a breadth-first search tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G. Then i and j differ by at most 1.

Pf.

- consider the moment BFS just examined x
  - nodes discovered from x belong to layers L_{i+1} or earlier



BFS: representation

BFS corresponds exactly to *queue* structure.

- extract elements in *first-in, first-out (FIFO)* order
- can be implemented via *doubly linked list*

BFS: representation

BFS corresponds exactly to *queue* structure.

- extract elements in *first-in, first-out (FIFO)* order
- can be implemented via *doubly linked list*

Cycle? Array `Discovered` of length n

- set `Discovered[v]= true` as soon as our search *first* sees v.

BFS: implementation

`Discovered`[s] = `true`; `Discovered`[v] = `false` for all other v;
$L[0] = \{s\}$; layer counter $i = 0$; current tree $T = \{\}$;
While $L[i]$ is not empty:

1. Initialize an empty list $L[i+1]$;
2. For each node $u \in L[i]$:
   1. Consider each edge (u, v) incident to u:
   2. If `Discovered`[v] = `false`:
      1. Set `Discovered`[v] = `true`;
      2. Add edge (u, v) to the tree T;
      3. Add v to the list $L[i+1]$;
   3. ++i;

BFS: analysis

Theorem. The above implementation of the BFS algorithm runs in time $O(m+n)$ (i.e., linear in the input size), if the graph is given by the adjacency list representation.
Pf.

- [worst case] easy to prove $O(n^2)$ time
  - at most n lists $L[i]$
    - while loop runs at most n times
  - at most n neighbors for each node
    - each spend $O(1)$ time

BFS: analysis

Theorem. The above implementation of the BFS algorithm runs in time $O(m+n)$ (i.e., linear in the input size), if the graph is given by the adjacency list representation.
Pf.

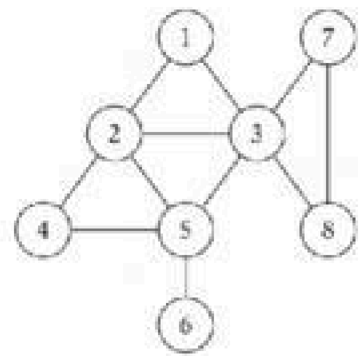
- [worst case] easy to prove $O(n^2)$ time
  - at most n lists $L[i]$
    - while loop runs at most n times
  - at most n neighbors for each node
    - each spend $O(1)$ time

- Actually runs in $O(m+n)$ time:
  - each node u has $degree(u)$ neighbors
    - total time processing edges: $O(\sum_{v \in V} n_v = 2m) = O(m)$
  - $O(n)$ additional time: set up lists, manage `Discovered`.

Depth-First Search (DFS)

DFS intuition: explore a maze.

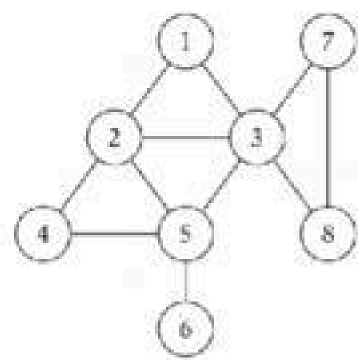
- keep going until reached a “dead end”
  - backtrack until an unexplored branch



Depth-First Search (DFS)

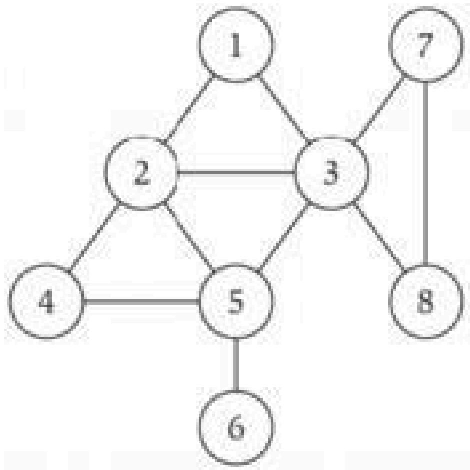
DFS intuition: explore a maze.

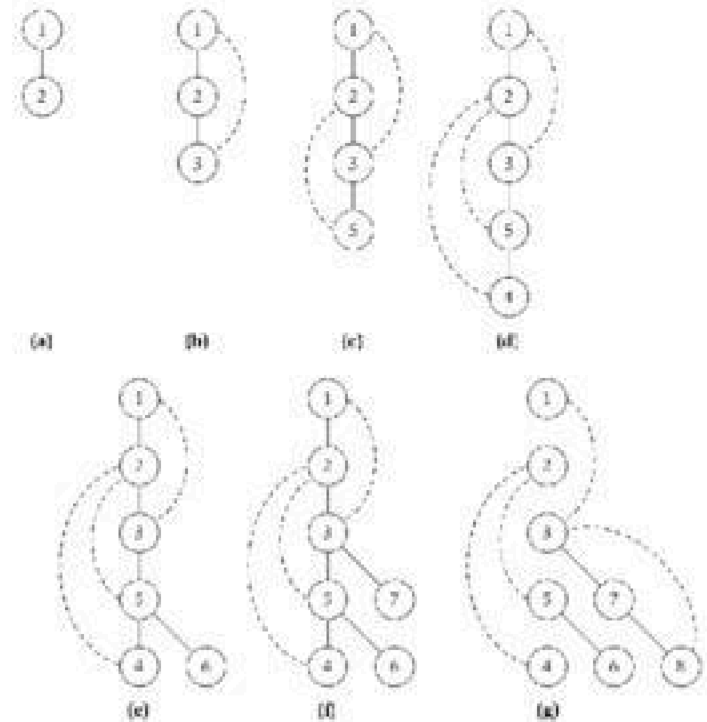
- keep going until reached a "dead end"
  - backtrack until an unexplored branch



Depth-first search tree: non-tree edges can only connect ancestors to descendants.

Depth-first search tree





DFS: representation

DFS corresponds exactly to *stack* structure.

- extract elements in *last-in, first-out (LIFO)* order
- can be implemented via *doubly linked list*

DFS: representation

DFS corresponds exactly to *stack* structure.

- extract elements in *last-in, first-out (LIFO)* order
- can be implemented via *doubly linked list*

Cycle? Array `Discovered` of length n

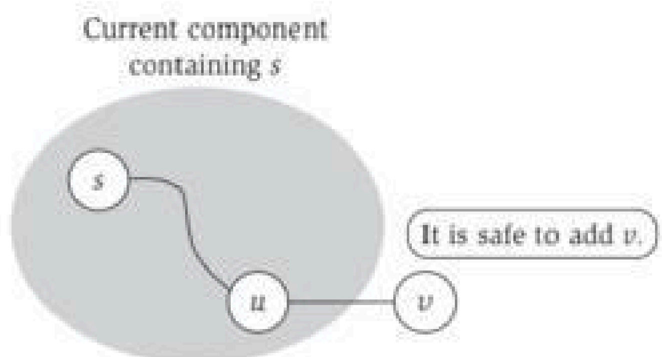
- set `Discovered[v]= true` as soon as our search *first* sees v.

Application: connected component

Connected component. Find all nodes reachable from s.

- Initially $R = \{s\}$
- While there is an edge (u, v) where $u \in R$ and $v \notin R$
  - Add v to R

Theorem. Upon termination, R is the connected component containing s.

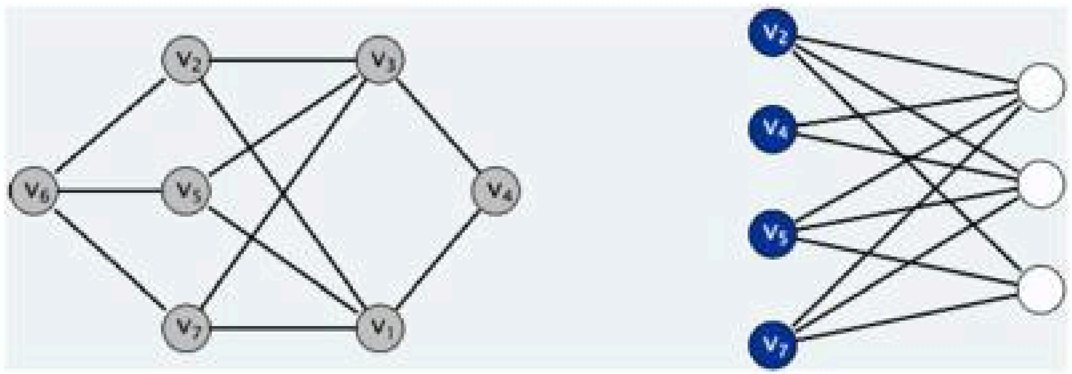


- **BFS**: explore in order of distance from s.
- **DFS**: explore in a recursive way.

Testing Bipartiteness

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.



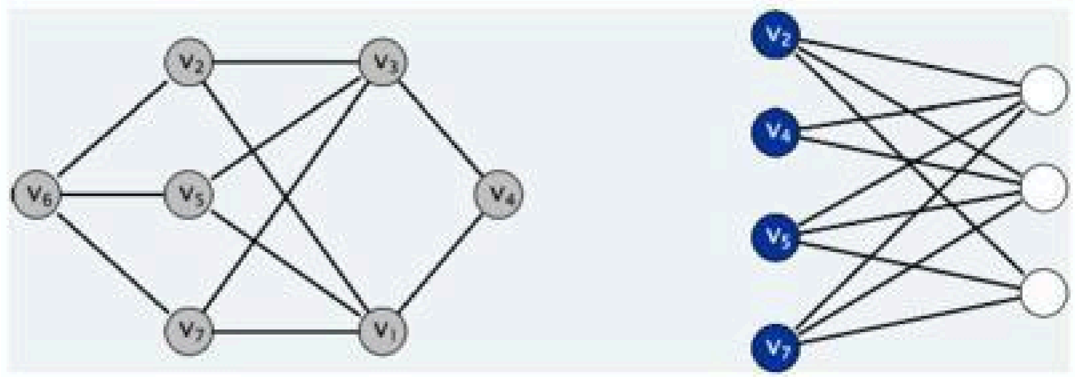
Applications.

- Stable matching: men = `blue`, women = `white`.
- Scheduling: machines = `blue`, jobs = `white`.

Testing bipartiteness

If the underlying graph is bipartite, many graph problems become:

- *Easier* (matching).
- *Tractable* (independent set).



Before attempting to design an algorithm, we need to *understand structure* of bipartite graphs.

An obstruction to bipartiteness

Clearly a triangle is not bipartite.

An obstruction to bipartiteness

Clearly a triangle is not bipartite.

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.

Bipartiteness: BFS algorithm

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:

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

(Proofs in the following slides.)

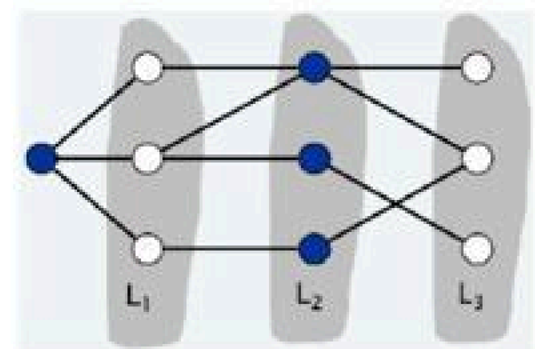
Bipartiteness: BFS algorithm, pf. I

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:

1. No edge of G joins two nodes of the same layer, and G is bipartite.
2. 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.



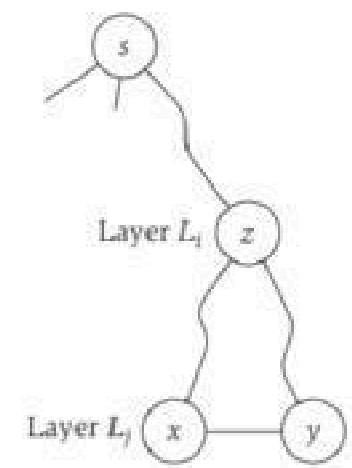
Bipartiteness: BFS algorithm, pf. II

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:

1. No edge of G joins two nodes of the same layer, and G is bipartite.
2. 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 $1 + (j–i) + (j–i)$, which is odd.



The only obstruction to bipartiteness

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

Connectivity in Directed Graphs

Directed graphs

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

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

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 webpages by importance.

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 webpages by importance.

Ex. Road network

- Node = crossroad;
- edge = one-way street.

Graph search

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

Directed $s \rightsquigarrow t$ shortest path problem. Given two nodes s and t, what is the length of a shortest path between them?

Graph search. BFS extends naturally to directed graphs.

Graph search

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

Directed $s \rightsquigarrow t$ shortest path problem. Given two nodes s and t, what is the length of a shortest path between them?

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.

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

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 $G^{reverse}$.
  - Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.

Strong components

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

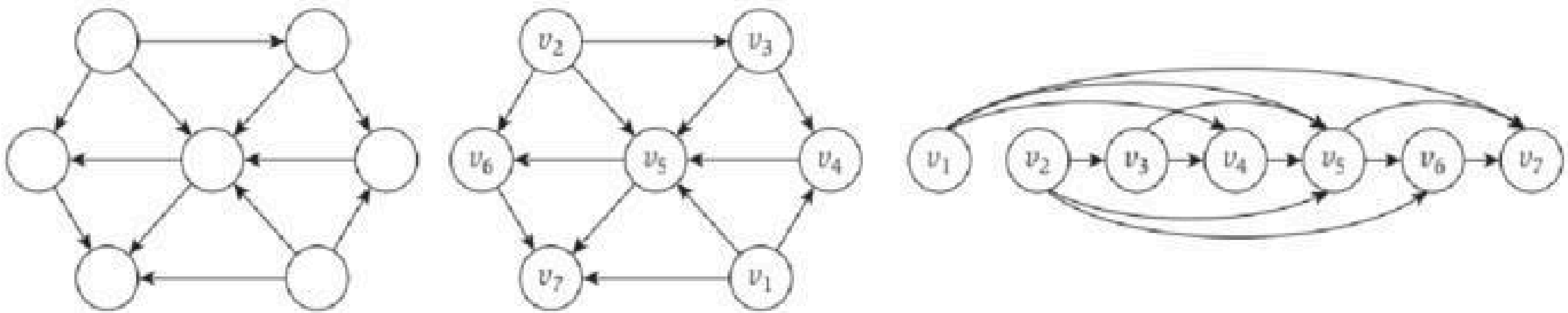
Theorem. [Tarjan 1972] Can find all strong components in $O(m + n)$ time.

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 $i < j$.



Precedence constraints

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

Applications.

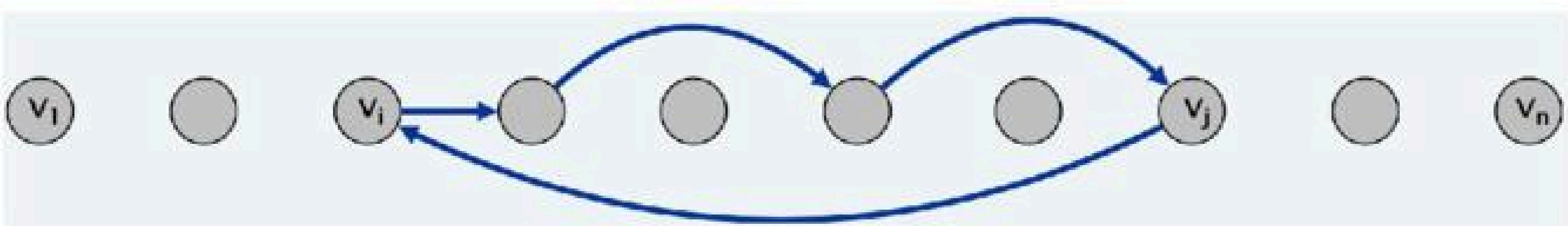
- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j.
- Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

DAG: determinant

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.

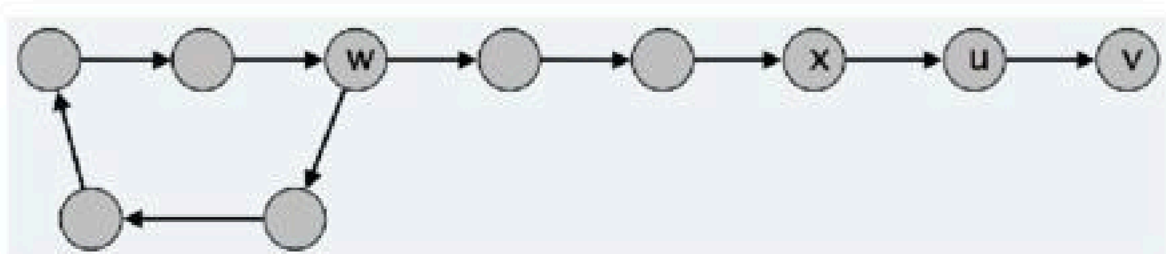


DAG: head

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.
  - 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.



DAG: property

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.

TS algorithm: analysis

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
    - add w to S if $count(w)$ hits 0
  - this is $O(1)$ per edge

TS algorithm: example