Lecture 3: Graphs and Graph Search Methods

subject	TDT4121
topics	BFS DFS Directed Acyclic Graphs Directed Graphs Graph Implementation Graphs Testing Bipartiteness Topological Ordering Trees
created	@September 12, 2022 10:30 AM

Chapter 3 - Graphs

> One of the most fundamental and expressive data structures in discrete mathematics

Goals 🏁

The student should be able to:

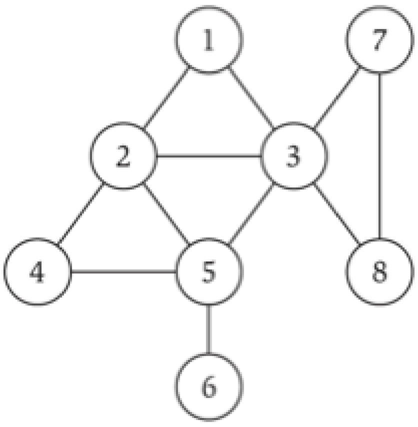
- Understand the basic definition of a *graph* ✅
    - The difference between a graph and a *directed graph* ✅
- Know of examples of applications of graphs ✅
- Understand what a *tree* is and their properties ✅
- Understand *BFS* - Breadth-First Search ✅
- Understand *DFS* - Depth-First Search ✅
- Understand how graphs are implemented
- Understand *Directed Acyclic Graphs* and *Topological Ordering* ✅

Basic Definitions and Applications

Undirected Graphs

Undirected graph. G = (V,E)

- V = nodes
- E = edges between pairs of nodes
- captures pairwise relationship between objects
- graph sice parameters: $n = |V|, m = |E|$



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

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

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

$n = 8$

$m = 11$

An *undirected* graph is basically the same as a *directed* graph with **bidirectional** connections (= two connections in opposite directions) between the connected nodes.

So you don't really have to do anything to make it work for an undirected graph. You only need to know all of the nodes that can be reached from every given node through e.g. an **adjacency list**.

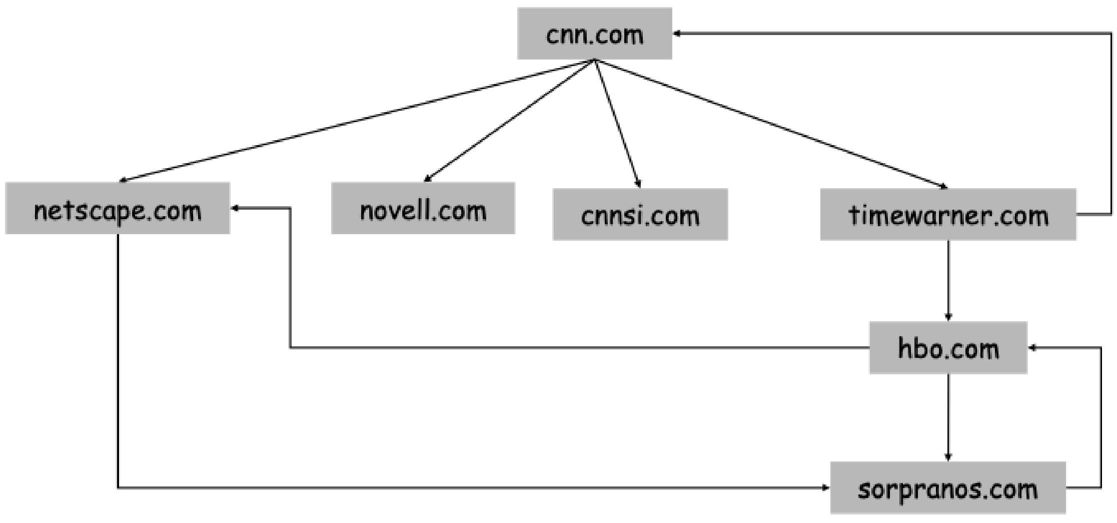
Some Graph Applications

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

World Wide Web

Web graph.

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

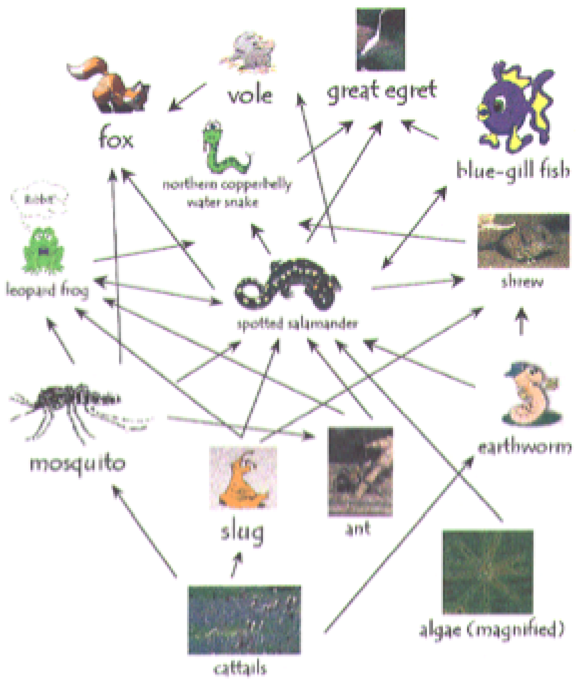


Ecological Food Web

Food web graph.

- Node = species

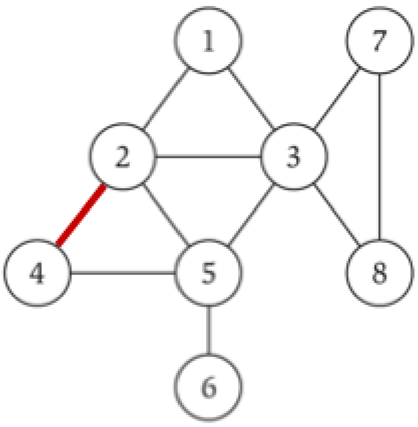
- Edge = from prey to predator

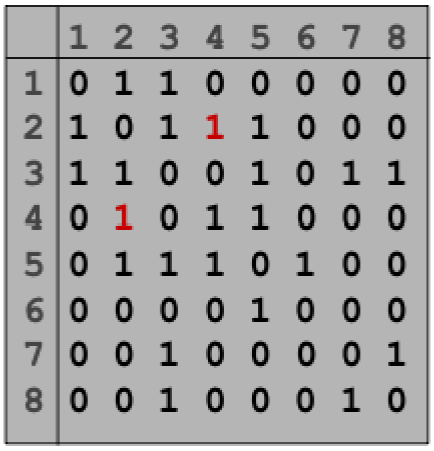


Graph Representation: Adjacency Matrix

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



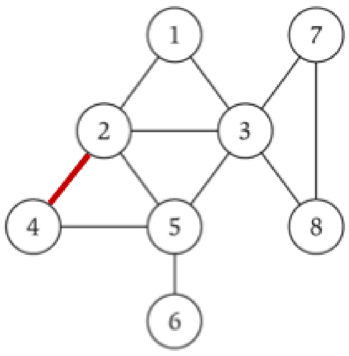


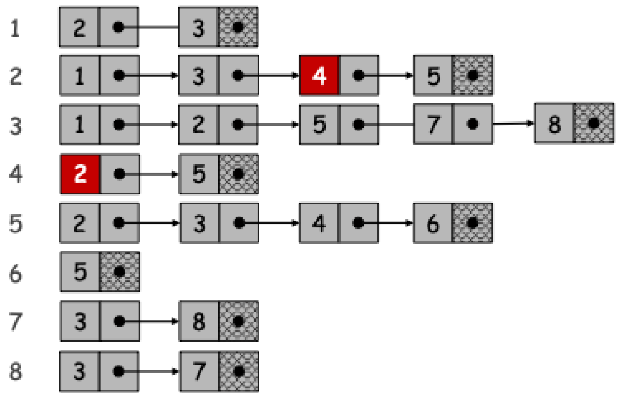
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

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
    - degree = number of neighbours of u
- Identifying all edges takes $\Theta(m+n)$ time



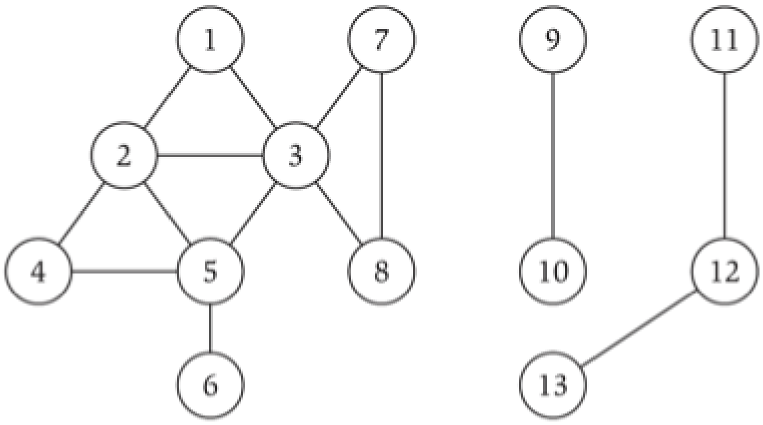


Paths and Connectivity

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

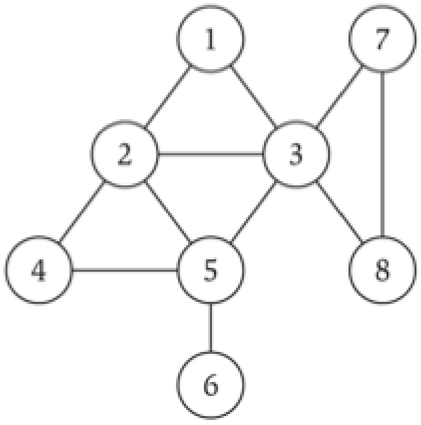
A path is **simple** if all nodes are distinct.

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



Cycles

A **cycle** is a path $v_1, v_2, ..., v_{k-1}, v_k$ in which $v_1 = v_k, k > 2$, and the first $k - 1$ nodes are all distinct.



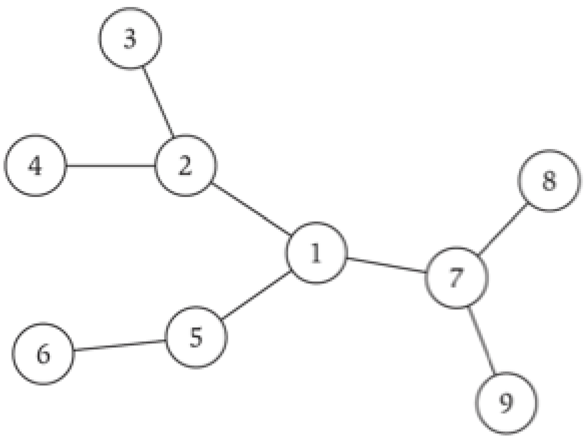
cycle C = 1-2-4-5-3-1

Trees

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.

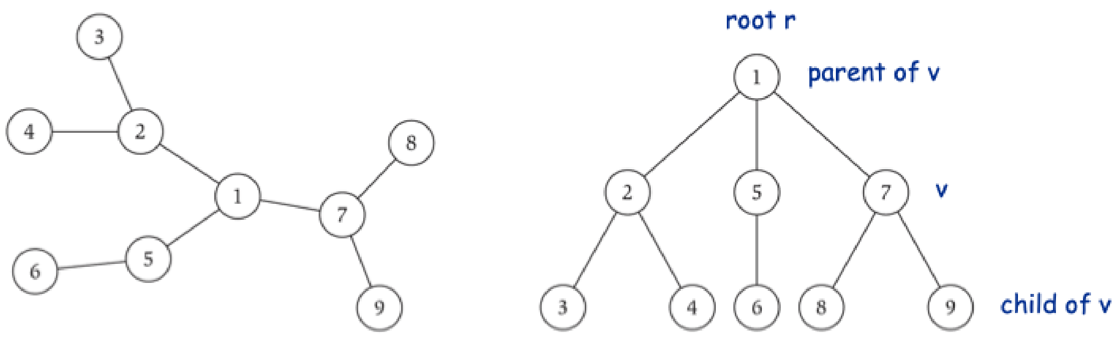
- G is connected
- G does not contain a cycle
- G has $n - 1$ edges



Rooted Trees

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

Importance. Models hierarchical structure.

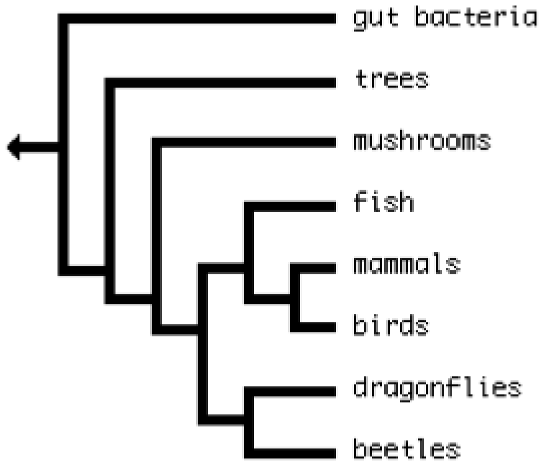


a tree

the same tree, rooted at 1

Phylogeny Trees

Phylogeny Trees . Describes evolutionary history of species.



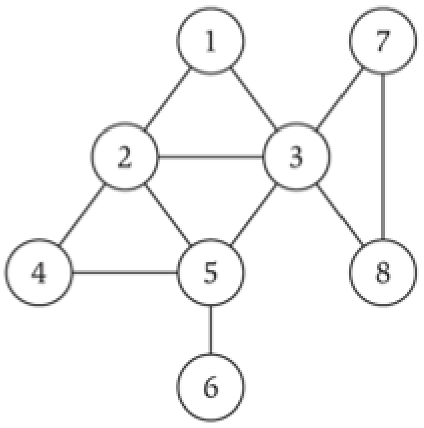
Connectivity

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 the shortest path between s and t?

Applications:

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



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

Breadth First Search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time

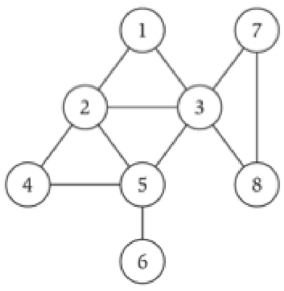
BFS algorithm.

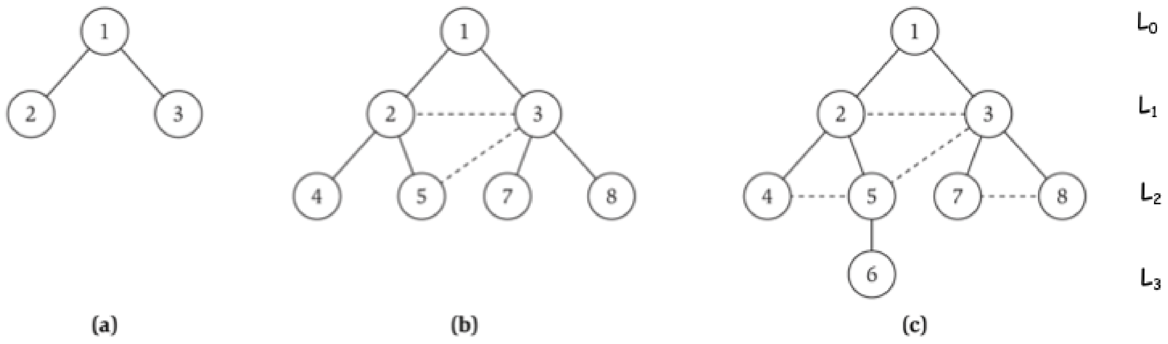
- $L_0 = s$

- $L_1 =$ all neighbours 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.

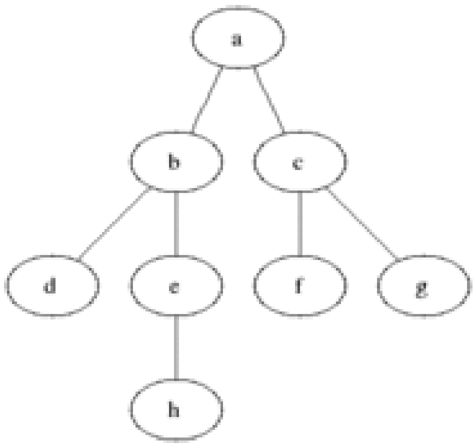
Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t if t appears in some layer.

Property. Let T be a BFS tree of $G = (V, E)$, and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.





Animated BFS:



Breadth First Search: Analysis

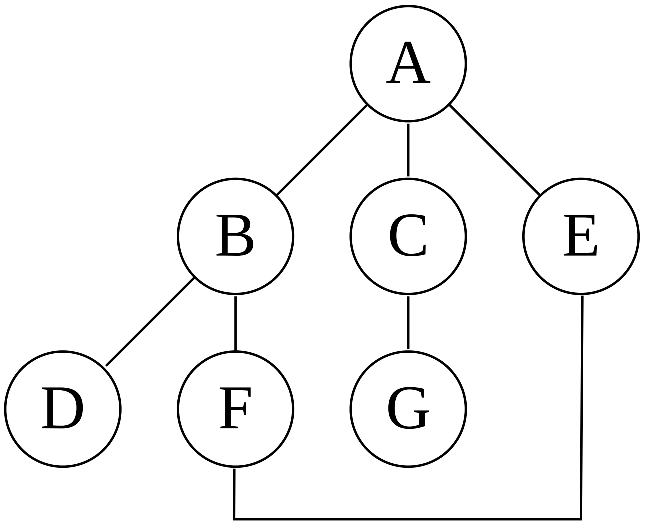
Theorem. The above implementation of BFS runs in O(m + n) time if the graph is given by its adjacency representation.

Pf.

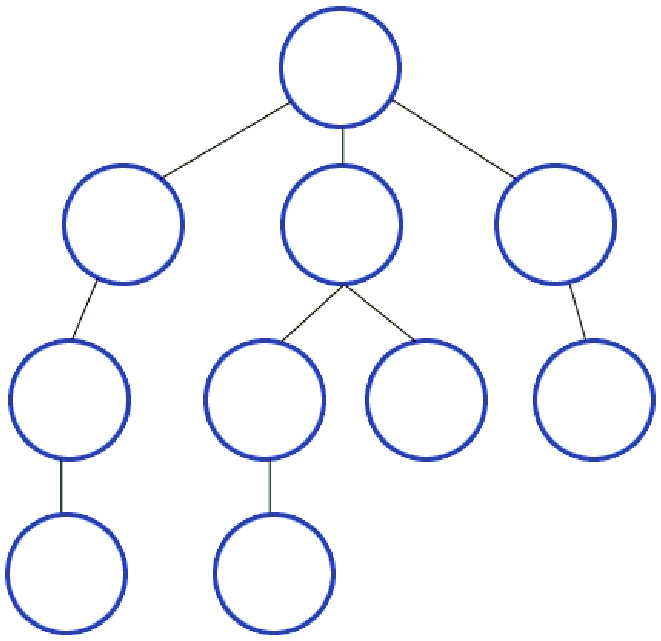
- Easy to prove $O(n^2)$ running time:
    - at most n lists $L[i]$
    - each node occurs on at most one list; for loop runs $\leq n$ times
    - when we consider node u, there are $\leq n$ incident edges (u, v), and we spend $O(1)$ processing each edge
- Actually runs in $O(m + n)$ time:
    - when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

Depth First Search

Depth-first search (**DFS**) is an algorithm for traversing or searching a tree or a graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

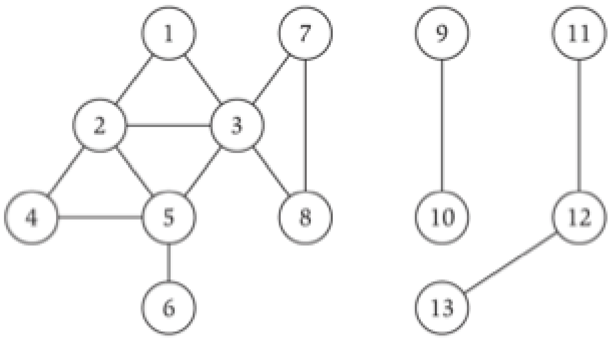


A depth-first search starting at the node A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree
, a structure with important applications in graph theory
. Performing the same search without remembering previously visited nodes results in visiting the nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.



Connected Component

Connected component. Find all nodes reachable from s.



Connected component containing node $1 = 1, 2, 3, 4, 5, 6, 7, 8$

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

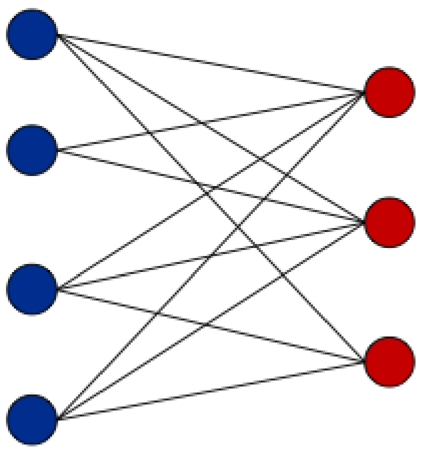
- BFS = explore in order of distance from s.
- DFS = explore in a different way.

Testing Bipartiteness

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

Applications.

- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

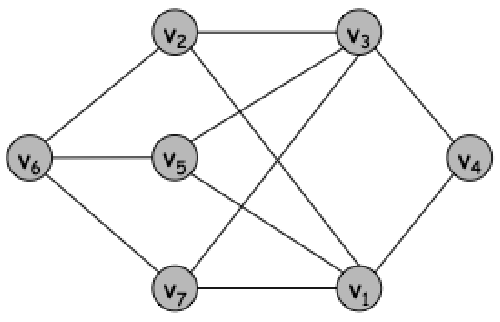


a bipartite graph

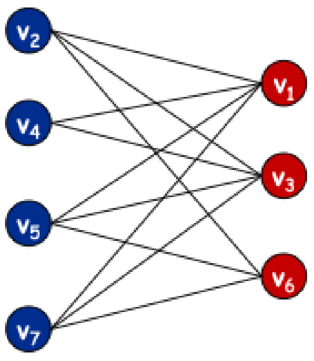
Testing Bipartiteness

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.



a bipartite graph G

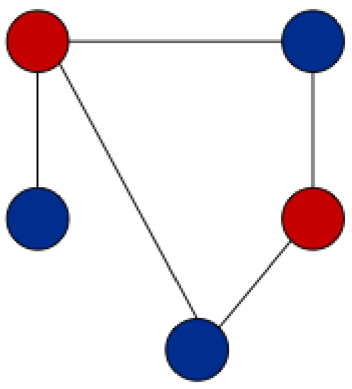


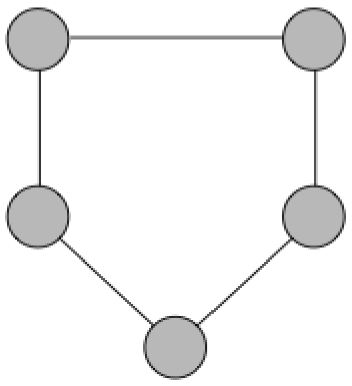
another drawing of G

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 cycle, let alone G.



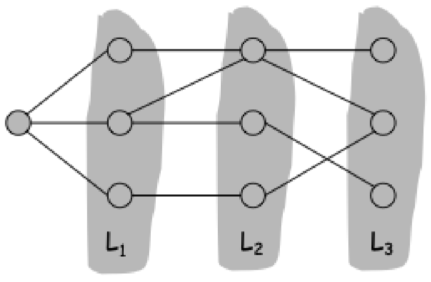


Bipartite Graphs

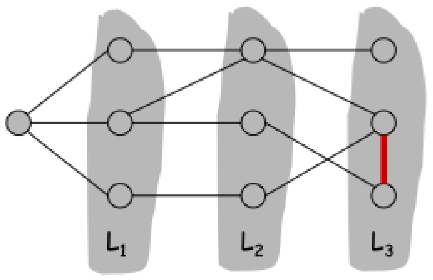
Lemma. Let G be a connected graph, and let $L_0, ..., L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(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)



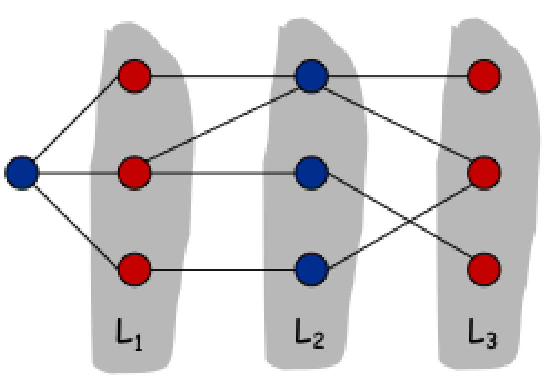
Case (ii)

Pf. (i)

- Suppose no edge joins two nodes in the same layer.
- By previous lemma, this implies all edges join nodes on adjacent

levels.

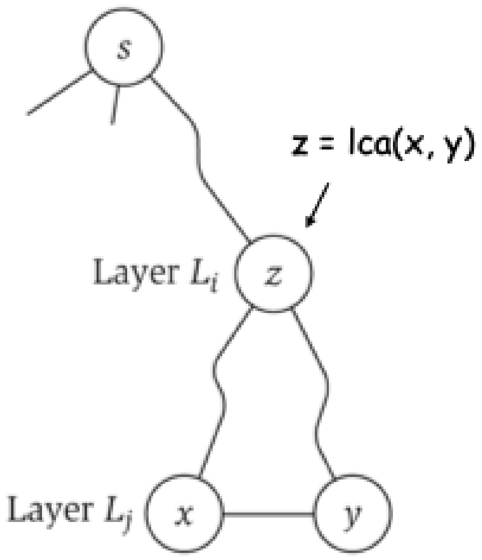
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Case (i)

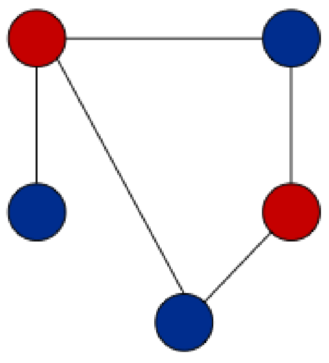
Pf. (ii)

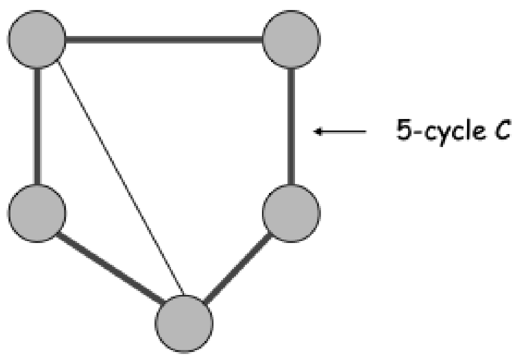
- Suppose (x, y) is an edge with x, y in same level L_j.
- Let $z = lca(x, y) =$ lowest common ancestor.
- Let L_i be level containing z.
- Consider cycle that takes edge from x to y,
  then path from y to z, then path from z to x.
- Its length is $1 + (j - i) + (j - i)$, which is odd



Obstruction to bipartiness

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



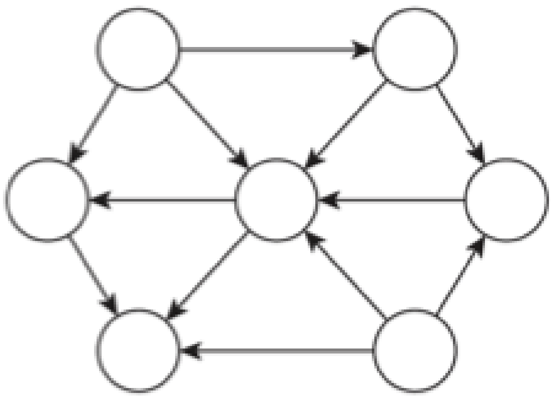


Connectivity in Directed Graphs

Directed Graph

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

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



Ex. Web graph - hyperlink points from one web page to another.

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

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 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.

Strong connectivity

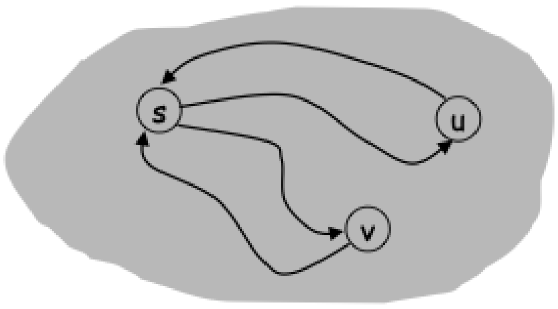
Def. Node u and v are **mutually reachable** if there is a path from u to v and also a path from v to u.

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-s path with s-v path. Path from v to u: concatenate v-s path with s-u path (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 Grev.
- Return true iff all nodes reached in both BFS executions.

- Correctness follows immediately from previous lemma

DAGs and Topological Ordering

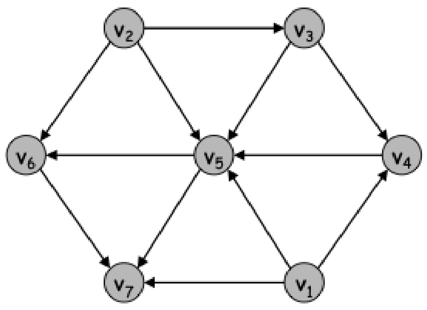
Directed Acyclic Graphs

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

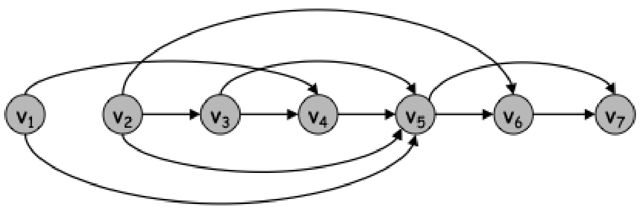
Ex. Precedence constraints: edge (v_i, v_j) means vi must precede v_j.

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

> **Topological sorting** is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological is only possible if the graph is a **Directed Acyclic Graph** (DAG). The first vertex in a topological sorting is always a vertex with in in-degree of 0 (a vertex with no incoming edges).



a DAG



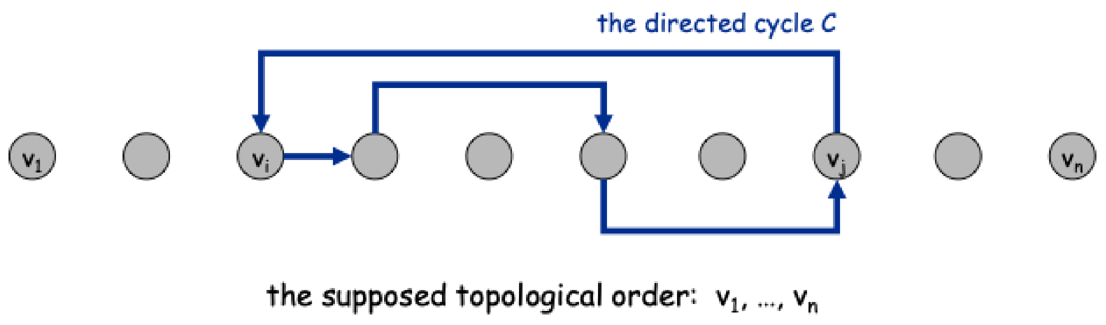
a topological ordering

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

Pf. (by contradiction)

- Suppose that G has a topological order $v_1, ..., v_n$ and that G also has a directed cycle C. Let's see what happens.
- Let vi 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.

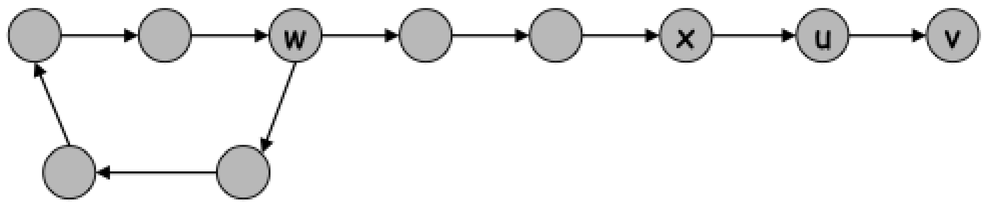
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, ..., v_n$ is a topological order, we must have $j < i$, a contradiction.



Lemma. 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 incoming edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
- Then, since u has at least one incoming 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.



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 incoming edges.

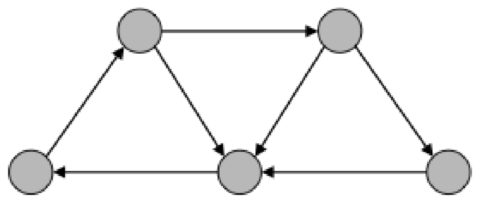
- G - { v } is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, G - { v } has a topological ordering.
- Place v first in topological ordering; then append nodes of G - { v }
- in topological order. This is valid since v has no incoming edges.

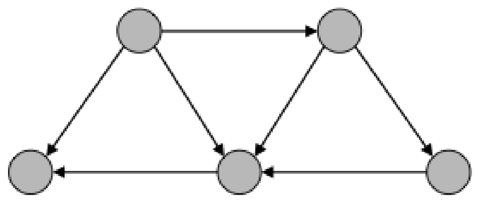
Precedence Constraints

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

Applications.

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


strongly connected


not strongly connected

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
- Initialisation: $O(m+n)$ via single scan through the 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