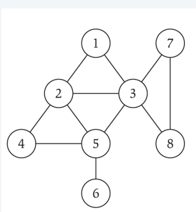
CSCI 355: Algorithm Design and Analysis
3. Graphs

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

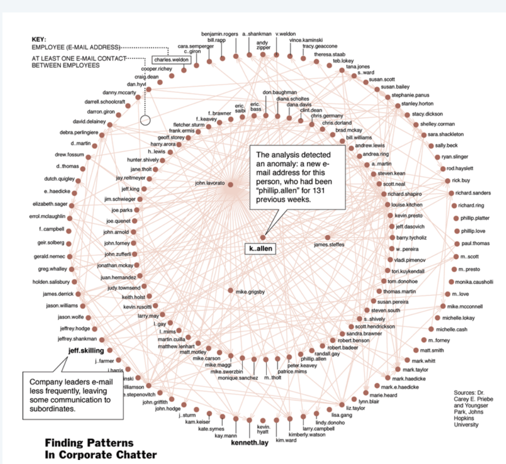


$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$

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

$m = 11, n = 8$

One week of Enron emails



Framingham heart study

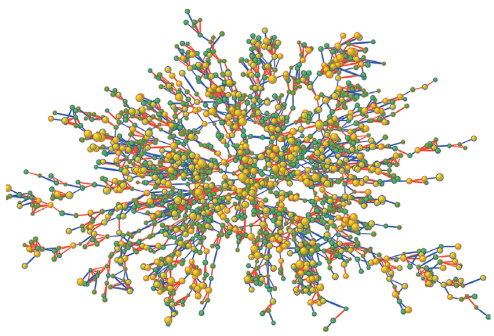


Figure 1. Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000. Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

Christakis and Fowler, "The Spread of Obesity in a Large Social Network over 32 Years", New England Journal of Medicine (2007)

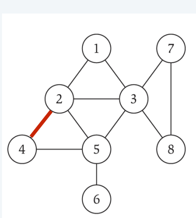
Some graph applications

graph	vertices	edges
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.



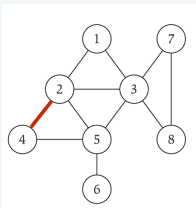
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	*1*	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	*1*	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

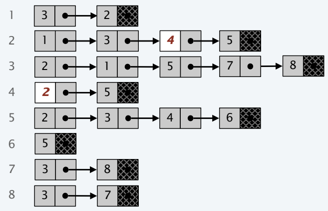
Graph representation: adjacency list

Adjacency lists. Vertex-indexed array of lists.

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

degree = number of neighbours of u



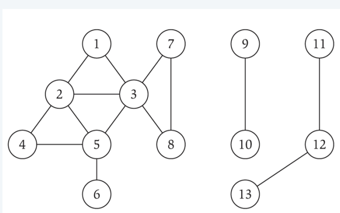


Paths and connectivity

Def. A path in an undirected graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, \ldots, v_k$ 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 vertices are distinct.

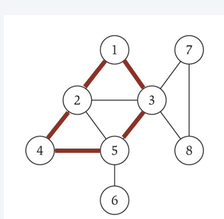
Def. An undirected graph is connected if for every pair of vertices u and v, there is a path between u and v.



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 vertices are distinct (except for v_1 and v_k).



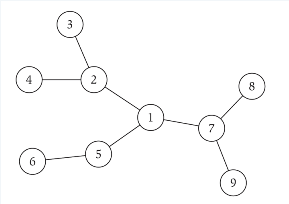
cycle C = 1-2-4-5-3-1

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 vertices. Any two of the following statements imply the third:

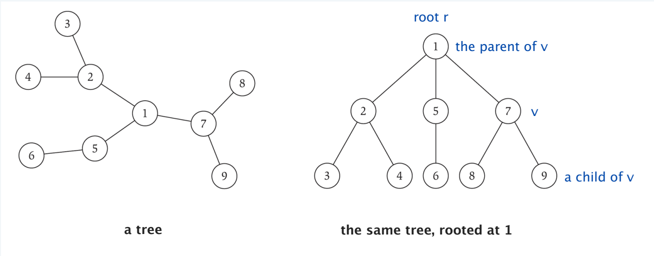
- G is connected.
- G does not contain a cycle.
- G has $n - 1$ edges.



Rooted trees

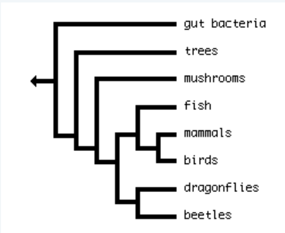
Given a tree T, choose a root vertex r and orient each edge away from r.

Importance. Models hierarchical structure.



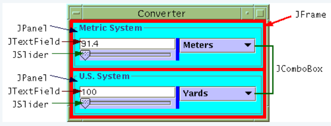
Phylogeny trees

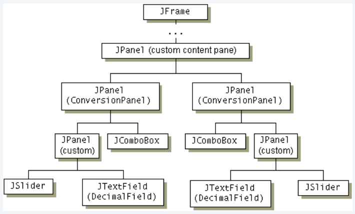
Describes the evolutionary history of species.



GUI containment hierarchy

Describes the organization of GUI widgets.





http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html

CSCI 355: ALGORITHM DESIGN AND ANALYSIS
3. GRAPHS

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

Connectivity

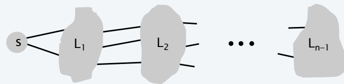
s-t connectivity problem. Given two vertices s and t, is there a path between s and t?

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

Applications.
- Facebook.
- Maze traversal.
- Erdős number.
- Kevin Bacon number.
- Fewest hops in a communication network.

Breadth-first search

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



BFS algorithm.

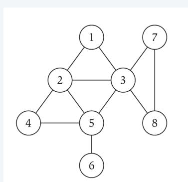
- $L_0 = \{ s \}$.
- L_1 = all neighbours of L_0.
- L_2 = all vertices that do not belong to L_0 or L_1, and that have an edge to a vertex in L_1.
- L_{i+1} = all vertices that do not belong to an earlier layer, and that have an edge to a vertex in L_i.

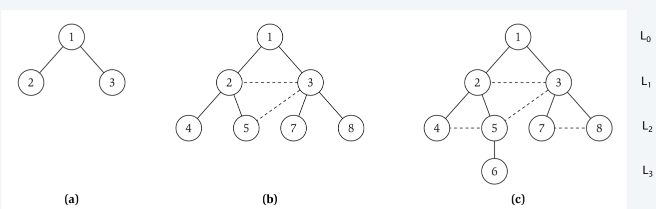
Theorem. For each i, L_i consists of all vertices at distance exactly i from s.

Theorem. There is a path from s to t iff t appears in some layer.

Breadth-first search

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.





Breadth-first search: analysis

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

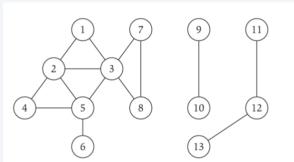
Pf.

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

each edge (u, v) is counted exactly twice in sum: once in degree(u) and once in degree(v)

Connected components

Connected component. Find all vertices reachable from s.

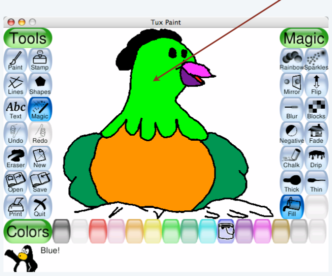


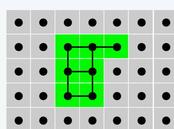
Connected component containing vertex $1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$.

Flood fill

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

- Vertex: pixel.
- Edge: two neighbouring lime green pixels.
- Blob: connected component of lime green pixels.

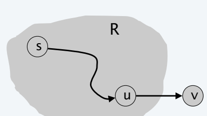




Connected components

Connected component. Find all vertices 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 ∈ R and v ∉ R
  Add v to R
Endwhile
```



adding 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 different way.

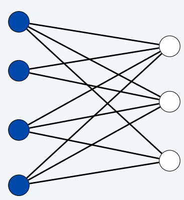
CSCI 355: Algorithm Design and Analysis
3. Graphs

- 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 vertices can be coloured 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.

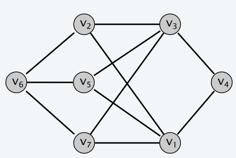


a bipartite graph

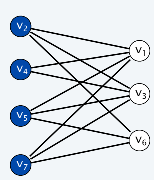
Testing bipartiteness

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 the structure of bipartite graphs.



a bipartite graph G

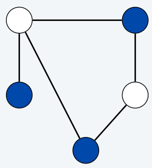


another drawing of G

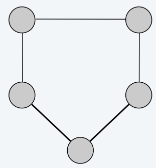
An obstruction to bipartiteness

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

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



bipartite (2-colourable)



not bipartite (not 2-colourable)

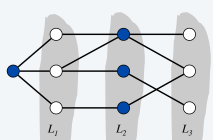
Testing bipartiteness

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

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

Pf. (i) No edge of G joins two vertices in the same layer, and G is bipartite.

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



Case (i)

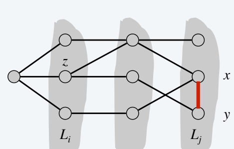
Testing bipartiteness

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

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

Pf. (ii) An edge of G joins two vertices in the same layer, and G contains an odd-length cycle (and hence is not bipartite).

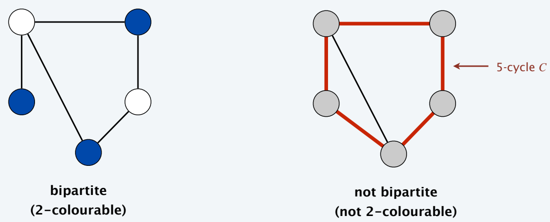
- Suppose (x, y) is an edge with x, y in the same layer L_j.
- Let $z = lca(x, y)$ denote the lowest common ancestor. Let L_i be the level containing z.
- Consider the cycle that takes the edge from x to y, then the path from y to z, then the path from z to x.
- Its length is $\underbrace{1}_{(x,\,y)} + \underbrace{(j-i)}_{\text{path } y\text{—}z} + \underbrace{(j-i)}_{\text{path } z\text{—}x}$, which is odd. ▪



Case (ii)

The *only* obstruction to bipartiteness

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



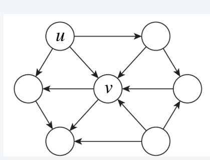
CSCI 355: Algorithm Design and Analysis
3. Graphs

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

Directed graphs

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

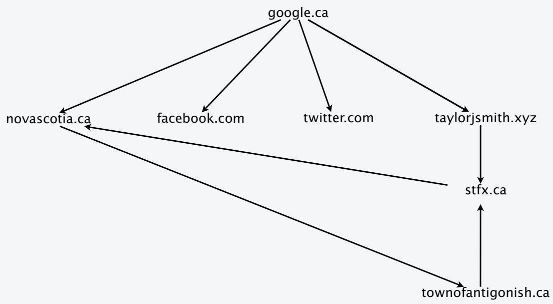
- Edge (u, v) leaves vertex u and enters vertex v.



World wide web

Web graph.

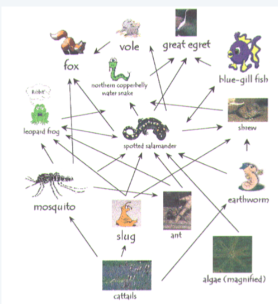
- Vertices: webpages.
- Edges: hyperlinks from one page to another (orientation is crucial).
- Modern search engines exploit hyperlink structure to rank web pages by importance.



Ecological food web

Food web graph.

- Vertices: species.
- Edges: connections from prey to predator.

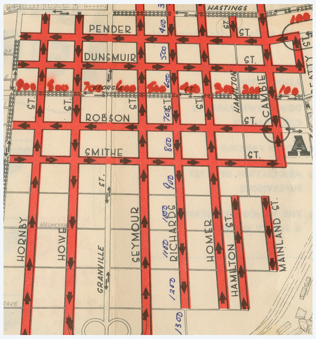


http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Road network

City map.

- Vertices: intersections.
- Edges: one-way streets.



City of Vancouver archives: Illustrated map of downtown Vancouver (1957)

Some directed graph applications

directed graph	vertices	directed edges
web	web page	hyperlink
food web	species	predator-prey relationship
transportation	street intersection	one-way street
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 vertex s, find all vertices reachable from s.

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

Graph search. BFS extends naturally to directed graphs.

Application.
- Web crawler: start from web page s. Find all web pages linked from s, either directly or indirectly.

Strong connectivity

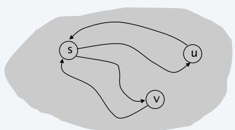
Def. Vertices u and v are mutually reachable if there is both a path from u to v and also a path from v to u.

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

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

Pf. ⇒ Follows from definition.
Pf. ⇐ 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

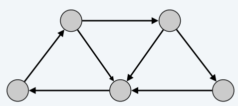


Strong connectivity: algorithm

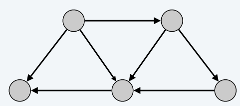
Theorem. We can determine if G is strongly connected in $O(m + n)$ time.

Pf.

- Pick any vertex 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 vertices are reached in both BFS executions.
- Correctness follows immediately from previous lemma. ▪



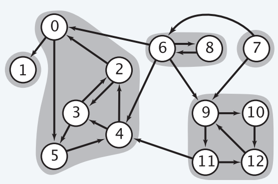
strongly connected



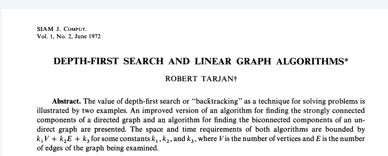
not strongly connected

Strong components

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



Theorem. [Tarjan 1972] We 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.

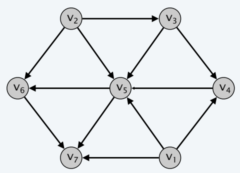
CSCI 355: Algorithm Design and Analysis
3. Graphs

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

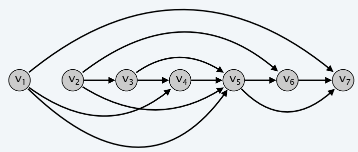
Directed acyclic graphs

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 vertices 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. An edge (v_i, v_j) means task v_i must occur before v_j.

Applications.

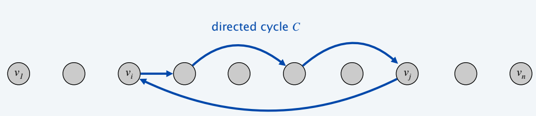
- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j.
- Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

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 v_i be the lowest-indexed vertex in C, and let v_j be the vertex 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. ▪



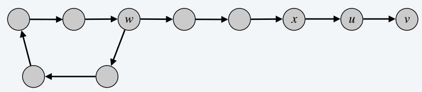
the supposed topological order: $v_1, \ldots, v_n$

Directed acyclic graphs

Lemma. If G is a DAG, then G has a vertex with no incoming edges.

Pf. [by contradiction]

- Suppose that G is a DAG and every vertex has at least one incoming edge.
- Pick any vertex 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 vertex, say w, twice.
- Let C denote the sequence of vertices encountered between successive visits to w.
- C is a cycle. ▪



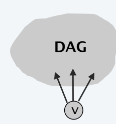
Directed acyclic graphs

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

Pf. [by induction on n]

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

Theorem. Our 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 vertices with no incoming edges
- Initialization: $O(m + n)$ via a 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 ▪