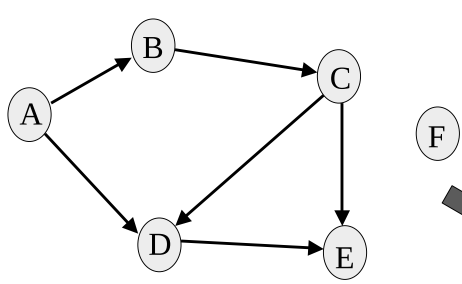
Lecture 23: Topo-Sort and Dijkstra's Greedy Idea

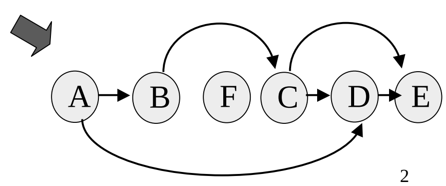
- What's the Buzz? Homework #5 is up on the web
  - Go to "Assignments" link on class web page
  - This is a programming assignment on graphs
    - Due June 1 (last day of class)
- Today's Topics:
  - Topological Sort (Take 2): Gunning for linear time…
  - Finding Shortest Paths
    - Breadth-First Search
    - Dijkstra's Method: Greed is good!
- Covered in Chapter 9 in the textbook

Recall from Last Time: Topological Sort

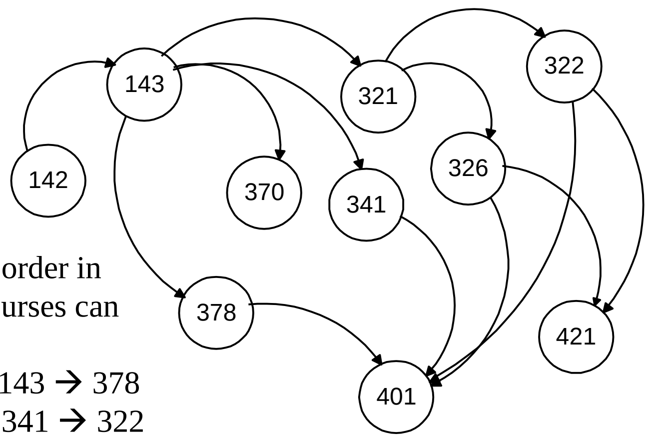
Topological sorting problem: given digraph $G = (V, E)$, output all the vertices in V such that no vertex is output before any other vertex with an edge to it



Any linear ordering in which all the arrows go to the right is a valid solution



Example Application of Topological Sort



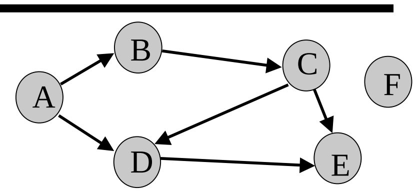
Problem: Find an order in which all these courses can be taken.

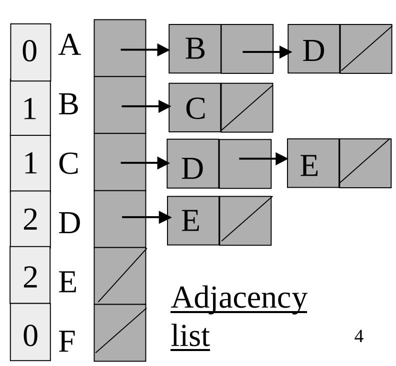
Example: 142 → 143 → 378 → 370 → 321 → 341 → 322 → 326 → 421 → 401

To take a course, all its prerequisites need to be taken first

Topo-Sort Algorithm #1 (from Last Time)

1. Store each vertex's **In-Degree** (# of incoming edges) in an array
2. While there are vertices remaining:
   - Find a vertex with In-Degree zero and output it
   - Reduce In-Degree of all vertices adjacent to it by 1
   - Mark this vertex (In-Degree = -1)





In-Degree array

Adjacency list

Topological Sort Algorithm #1: Analysis

For input graph G = (V,E), Run Time = ?

Break down into total time to:

→ Initialize In-Degree array: O(|E|)

→ Find vertex with in-degree 0: |V| vertices, each takes O(|V|) to search In-Degree array. Total time = O(|V|^2)

→ Reduce In-Degree of all vertices adjacent to a vertex: O(|E|)

→ Output and mark vertex: O(|V|)

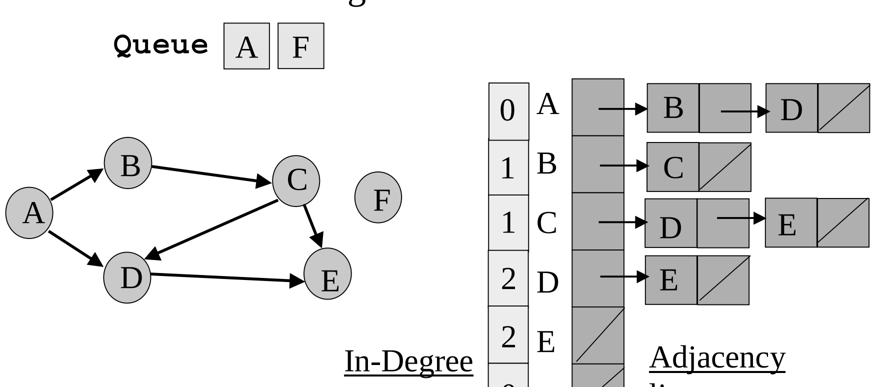
Total time = **O(|V|^2 + |E|)** → **Quadratic time!**

Can we do better than quadratic time?

Problem: Need a faster way to find vertices with in-degree 0?

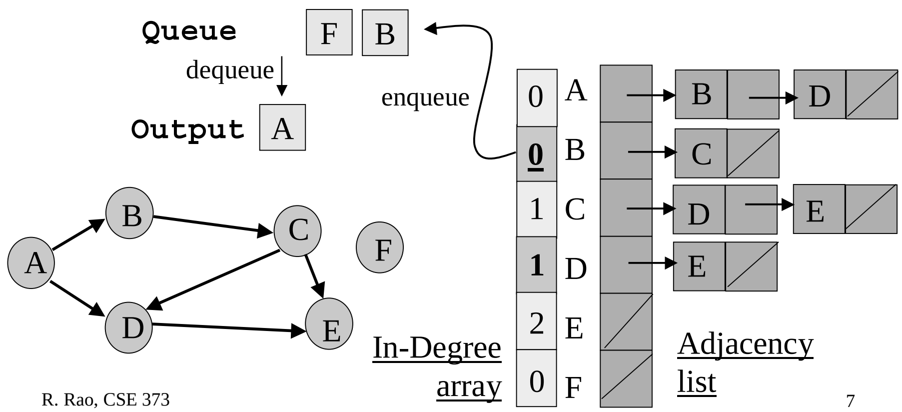
Topological Sort (Take 2)

Key idea: Initialize and maintain a *queue (or stack)* of vertices with In-Degree 0



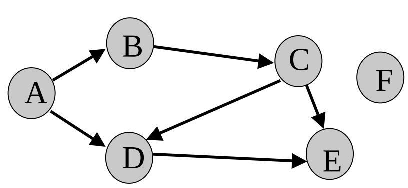
Topological Sort (Take 2)

After each vertex is output, when updating In-Degree array, *enqueue any vertex whose In-Degree has become zero*



Topological Sort Algorithm #2

1. Store each vertex's **In-Degree** in an array
2. Initialize a queue with all in-degree zero vertices
3. While there are vertices remaining in the queue:
   - Dequeue and output a vertex
   - Reduce In-Degree of all vertices adjacent to it by 1
   - Enqueue any of these vertices whose In-Degree became zero



Sort this digraph!

Topological Sort Algorithm #2: Analysis

For input graph G = (V,E), Run Time = ?

Break down into total time to:

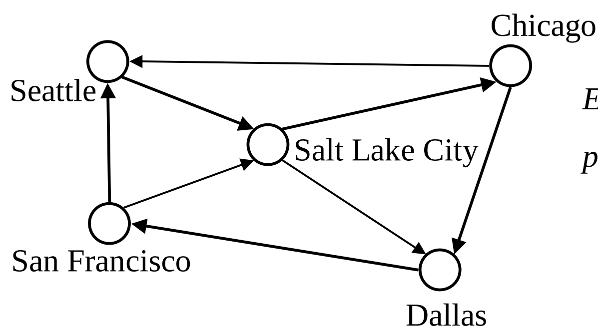
→ Initialize In-Degree array: O(|E|)
→ Initialize Queue with In-Degree 0 vertices: O(|V|)
→ Dequeue and output vertex: |V| vertices, each takes only O(1) to dequeue and output. Total time = O(|V|)
→ Reduce In-Degree of all vertices adjacent to a vertex and Enqueue any In-Degree 0 vertices: O(|E|)

Total time = **O(|*V*| + |*E*|)** → **Linear running time!**

Heads-up: You will be implementing this algorithm in HW #5

Paths

- Recall definition of a path in a tree – same for graphs
- A *path* is a list of vertices $\{v_1, v_2, \ldots, v_n\}$ such that (v_i, v_{i+1}) is in E for all $0 \leq i < n$.



Example of a path:

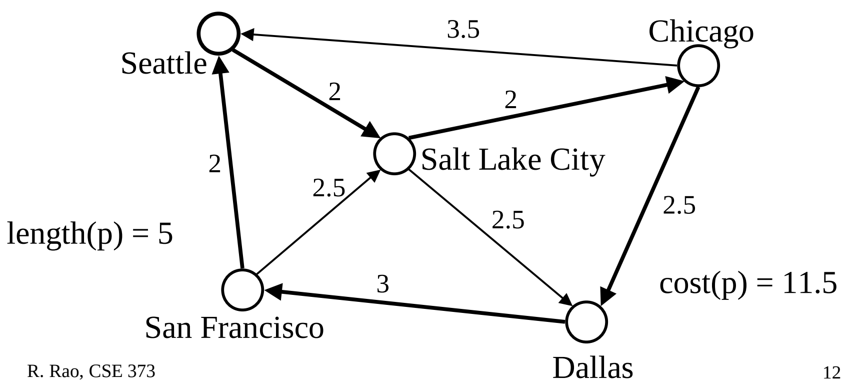
p = {Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle}

Simple Paths and Cycles

- A *simple path* repeats no vertices (except the 1st can be the last):
  - p = {Seattle, Salt Lake City, San Francisco, Dallas}
  - p = {Seattle, Salt Lake City, Dallas, San Francisco, Seattle}
- A *cycle* is a path that starts and ends at the same node:
  - p = {<u>Seattle</u>, Salt Lake City, Dallas, San Francisco, <u>Seattle</u>}
- A *simple cycle* is a cycle that repeats no vertices except that the first vertex is also the last
- A directed graph with no cycles is called a DAG (directed acyclic graph) E.g. All trees are DAGs
  - A graph with cycles is often a DRAG…(okay, that's a bad joke)

Path Length and Cost

- *Path length*: the number of edges in the path
- *Path cost*: the sum of the costs of each edge
  - Path length is simply the unweighted path cost (edge weight = 1)



Single Source, Shortest Path Problems

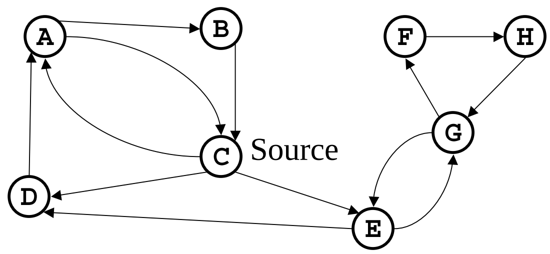
- Given a graph G = (*V*, *E*) and a "source" vertex *s* in *V*, find the minimum cost paths from *s* to every vertex in *V*
- Many variations:
  - unweighted vs. weighted
  - cyclic vs. acyclic
  - positive weights only vs. negative weights allowed
  - multiple weight types to optimize
  - Etc.
- We will look at only a couple of these…
  - See text if you are interested in the others

Why study shortest path problems?

- Plenty of applications
- Traveling on a budget: What is the cheapest multiple-stop airline schedule from Seattle to city X?
- Optimizing routing of packets on the internet:
  - Vertices are routers and edges are network links with different delays
  - What is the routing path with smallest total delay?
- Hassle-free commuting: Finding what highways and roads to take to minimize total delay due to traffic
- Finding the fastest way to get to coffee vendors on campus from your classrooms

Unweighted Shortest Paths Problem

<u>Problem</u>: Given a "source" vertex *s* in an unweighted graph G = (*V*,*E*), find the shortest path from *s* to all vertices in G

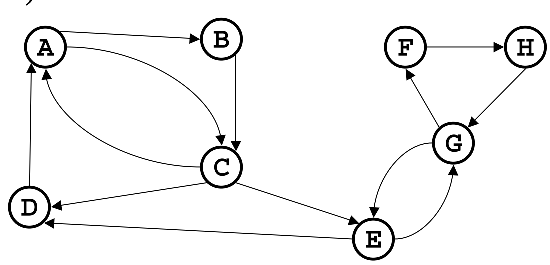


Find the shortest path from **C** to: **A B C D E F G H**

Solution based on Breadth-First Search

- <u>Basic Idea</u>: Starting at node s, find vertices that can be reached using 0, 1, 2, 3, …, N-1 edges (works even for cyclic graphs!)

On-board example:



Find the shortest path from **C** to: **A B C D E F G H**

Breadth-First Search (BFS) Algorithm

- Uses a queue to track vertices that need to be expanded
- Pseudocode (source vertex is `s`):

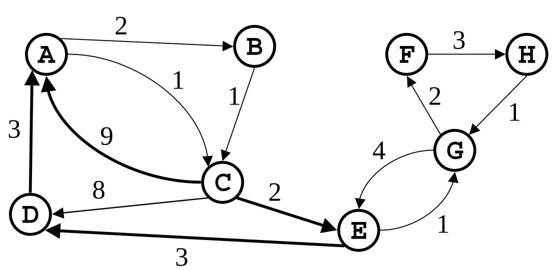
```
1. Dist[s] = 0
2. Enqueue(s)
3. While queue is not empty
   1. X = dequeue
   2. For each vertex Y adjacent to X and not
      previously visited
      • Dist[Y] = Dist[X] + 1
      • Prev[Y] = X
      • Enqueue Y
```

- Running time (same as topological sort) = **O(|*V*| + |*E*|)** (why?)

That was easy…what if edges have weights?

- BFS does not work anymore – minimum cost path may have additional hops

Shortest path from C to A:
BFS: C→A
(cost = 9)
Minimum Cost Path = C→E→D→A
(cost = 8)



Dijkstra to the rescue…

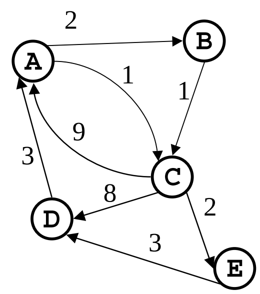
- ✦ Legendary figure in computer science; now a professor at University of Texas at Austin.
- ✦ Some gossip about D. from CSE 326 (2000)…
- ✦ Rumor #1: Supports teaching introductory computer courses without computers (pencil and paper programming).
- ✦ Rumor #2: Supposedly wouldn't (until recently) read his e-mail; so, his staff had to print out his e-mails and put them in his mailbox.

Dijkstra's Algorithm for Weighted Shortest Path

- ✦ Classic algorithm for solving shortest path in weighted graphs (without negative weights)
- ✦ A *greedy* algorithm (irrevocably makes decisions without considering future consequences)
- ✦ <u>Basic Idea:</u>
  - Similar to BFS
    - Each vertex has a cost for path from source
    - Vertices to be expanded have least cost seen so far
      - Greedy choice – always expand least cost vertex
    - But unlike BFS, a vertex already visited may be updated if a better path to it is found

Pseudocode for Dijkstra's Algorithm

1. Initialize the cost of each node to ∞
2. Initialize the cost of the source to 0
3. While there are unknown nodes left in the graph
   1. Select the unknown node *N* with the *lowest cost*
   2. Mark *N* as known
   3. For each node *A* adjacent to *N*
      If (*N*'s cost + cost of (*N*, *A*)) < A's cost
      A's cost = *N*'s cost + cost of (*N*, *A*)
      Prev[*A*] = *N* *//store preceding node*

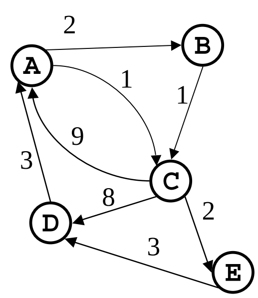


(Prev allows paths to be reconstructed)

Dijkstra's Algorithm (greed in action)

Work through this example…

vertex	known	cost	Prev
A			
B			
C			
D			
E			



Next Class:

Does Dijkstra's method always work?

How fast does it run?

To Do:

Start Programming Assignment #2

(Don't wait until the last few days!!!)

Continue reading and enjoying chapter 9