# CSE 332: Data Structures and Parallelism

Fall 2022

Anjali Agarwal

Lecture 18: Graph Theory

# Announcements

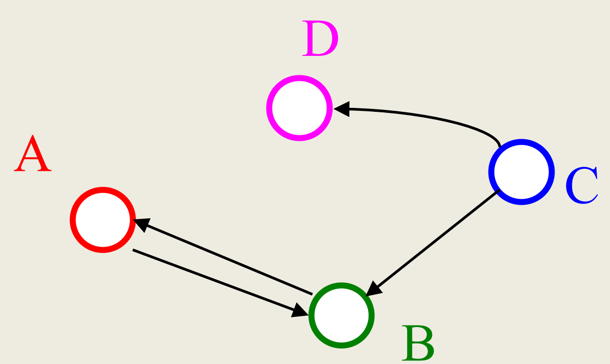
- Upcoming lectures
  - ~~Intro to graphs~~
  - Topological Sort
  - Parallelism (3 lectures)
  - Concurrency (2 lectures)

# Graphs

A formalism for representing binary relationships between objects

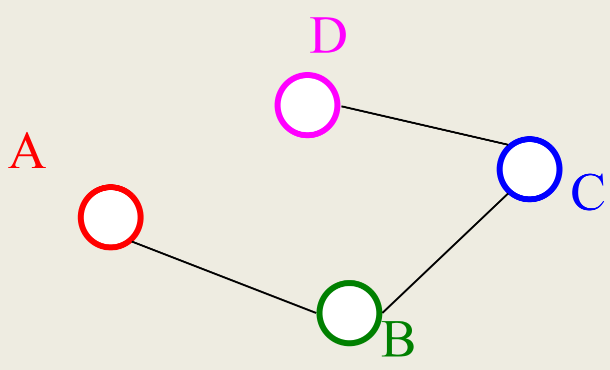
- Graph `G = (V,E)`
  - *Set of vertices*: $V = \{v_1, v_2, \ldots, v_n\}$
  - *Set of edges*: $E = \{e_1, e_2, \ldots, e_m\}$

## Directed




`V = {A, B, C, D}`
`E = {(C, B),(A, B),(B, A),(C, D)}`

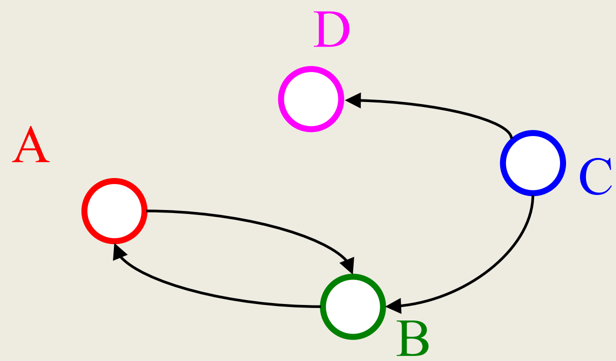
## Undirected

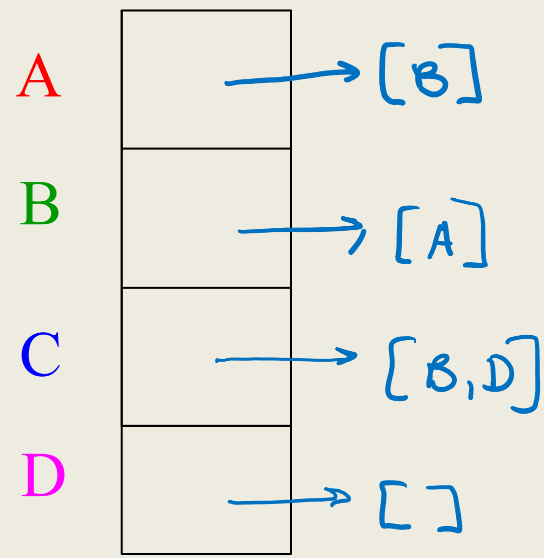



`V = {A, B, C, D}`
`E = {{C, B},{A, B},{C, D}}`

# Representation 1: Adjacency List

A list (array) of length `|V|` in which each entry stores a list (linked list) of all adjacent vertices




*Runtimes:*

*Iterate over vertices?* $O(|V|)$

*Iterate over edges?* $O(|V| + |E|)$

*Iterate edges adj. to vertex?* $O(d_v)$

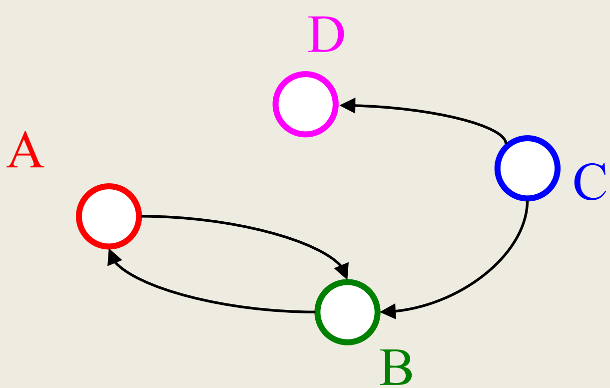
*Existence of edge?* $O(d_v)$

*Space requirements?* $O(|V| + |E|)$

*Best for what kinds of graphs?* SPARSE

# Representation 2: Adjacency Matrix

A `|V| x |V|` matrix **M** in which an element `M[u,v]` is true if and only if there is an edge from **u** to **v**




To

From

| | A | B | C | D |
|---|---|---|---|---|
| **A** | 0 | 1 | 0 | 0 |
| **B** | 1 | 0 | 0 | 0 |
| **C** | 0 | 1 | 0 | 1 |
| **D** | 0 | 0 | 0 | 0 |

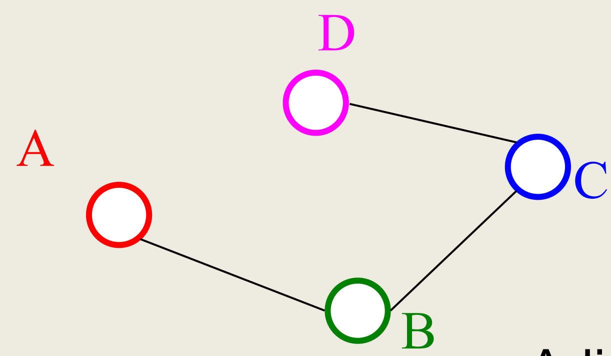
*Runtimes:*
*Iterate over vertices?* $O(|V|)$
*Iterate over edges?* $O(|V|^2)$
*Iterate edges adj. to vertex?* $O(|V|)$
*Existence of edge?* $O(1)$

*Space requirements?* $O(|V|^2)$

*Best for what kinds of graphs?*
DENSE

# Representing Undirected Graphs

What do these reps look like for an undirected graph?




Adjacency matrix:

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 0 | 0 |
| B | 1 | 0 | 1 | 0 |
| C | 0 | 1 | 0 | 1 |
| D | 0 | 0 | 1 | 0 |

* Symmetric across the diagonal

Adjacency list:

- A → [B]
- B → [A, C]
- C → [B, D]
- D → [C]

# |E| and |V|

- How many edges |E| in a directed graph with |V| vertices?

$0 \le |E| \le |V|(|V|-1)$

- How many edges |E| in a undirected graph with |V| vertices?

$0 \le |E| \le \frac{|V|(|V|-1)}{2}$

- How many edges |E| in a undirected, connected graph with |V| vertices?

$|V|-1 \le |E| \le \frac{|V|(|V|-1)}{2}$

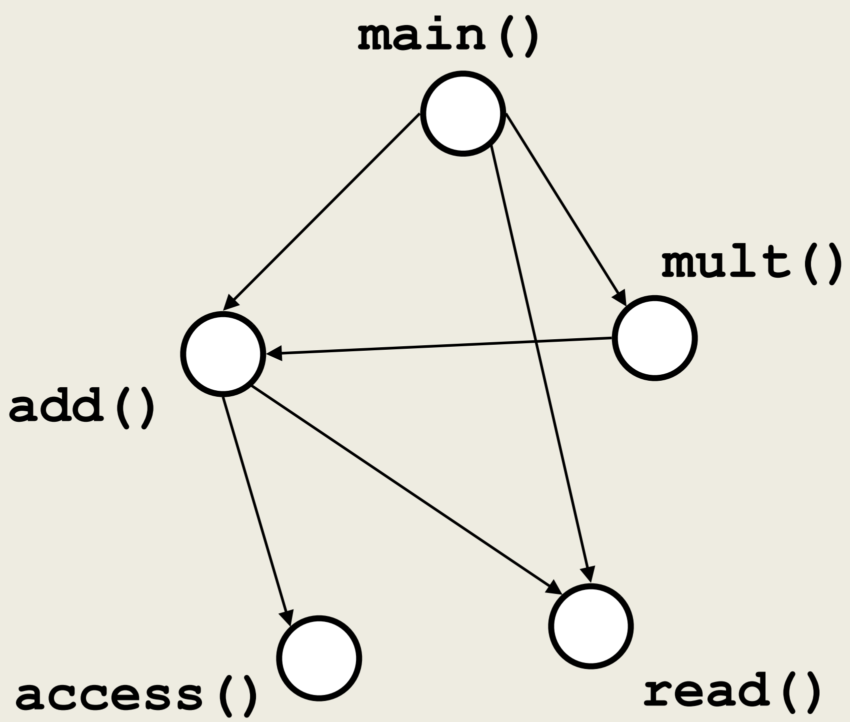
e.g. each vertex has 3 outgoing edges

- Some (semi-standard) terminology:
  - A graph is *sparse* if it has O(|V|) edges (upper bound).
  - A graph is *dense* if it has $\Theta(|V|^2)$ edges.

e.g. each vertex is neighbors with 1/2 the other edges

# Directed Acyclic Graphs (DAGs)

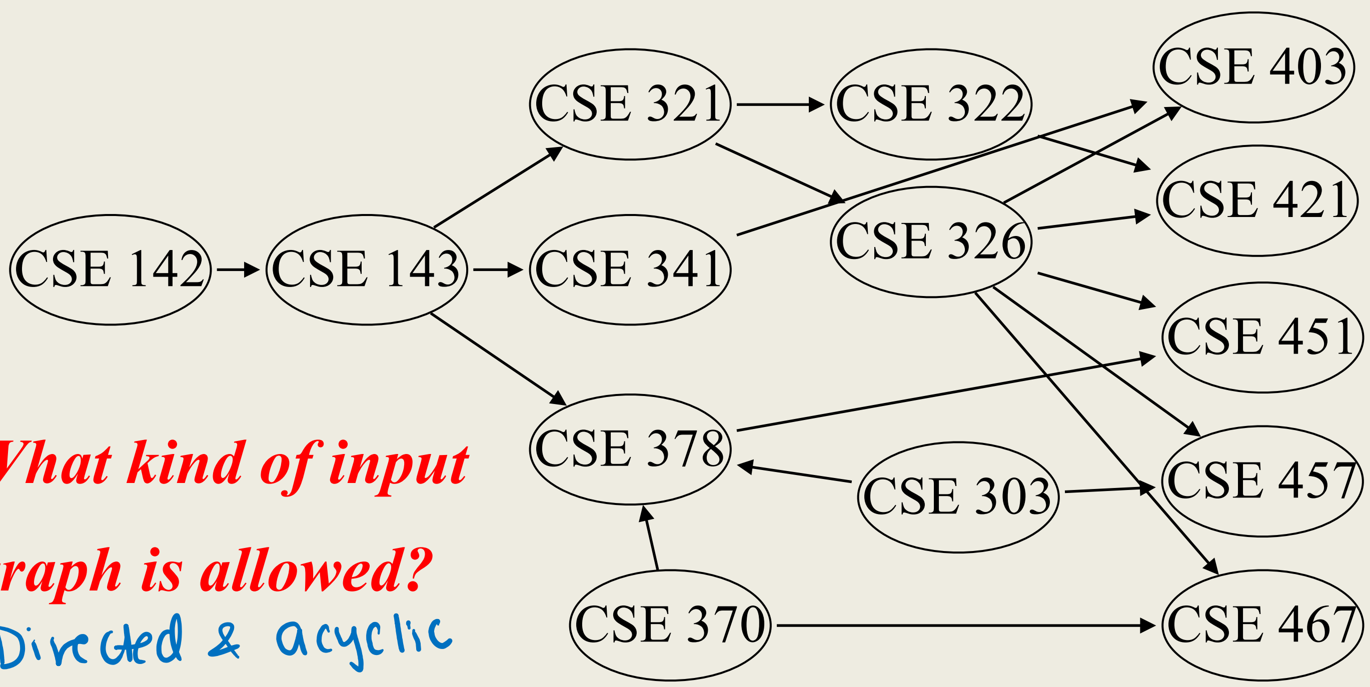
- DAGs are directed graphs with no (directed) cycles.



# Topological Sort

- Given a directed graph, **`G = (V,E)`**, output all the vertices in **`V`** sorted so that no vertex is output before any other vertex with an edge to it.

142
143
321
341
303
370
378
322
326
403
421
451
457
467



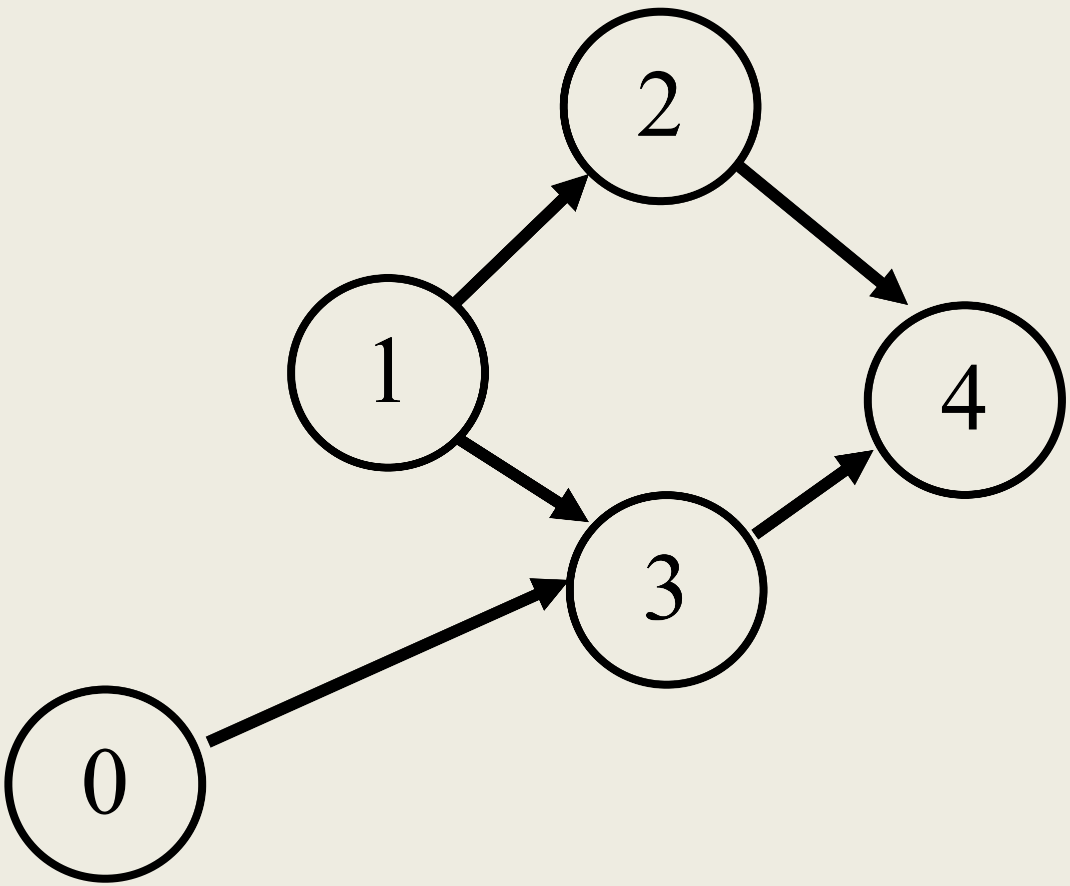

***What kind of input graph is allowed?***

Directed & acyclic

***Is the output unique?***

No.

# Find valid topological sorts

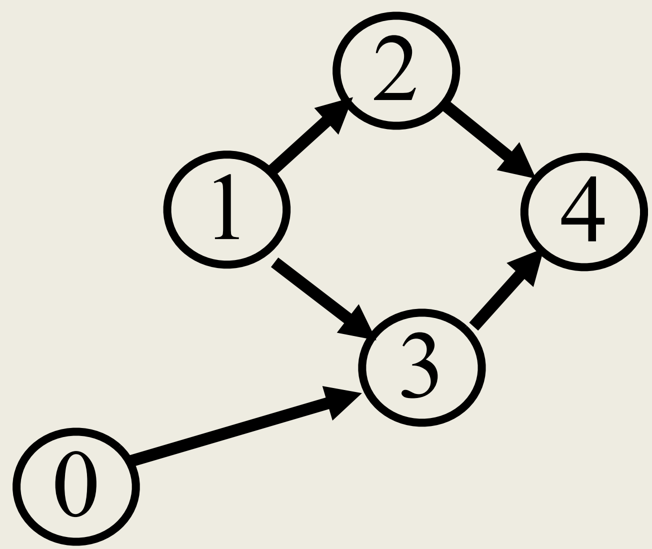



0 1 2 3 4
0 1 3 2 4
1 0 2 3 4
1 0 3 2 4
1 2 0 3 4

# Topological Sort: Take One

1. Label each vertex with its *in-degree* (# inbound edges)
2. **While** there are vertices remaining:
   a. Choose a vertex *v* of *in-degree zero*; output *v*
   b. Reduce the in-degree of all vertices adjacent to *v*
   c. Remove *v* from the list of vertices

| | In-degree | | |
|---|---|---|---|
| 0 | 0 | | → [3] |
| 1 | 0 | | → [2, 3] |
| 2 | ~~1~~ 0 | | → [4] |
| 3 | ~~2~~ ~~1~~ 0 | | → [4] |
| 4 | ~~2~~ ~~1~~ 0 | | → [] |




0 1 2 3 4

```
void topsort(){
  labelEachVertexWithItsInDegree();
  for (int counter=0; counter < NUM_VERTICES; counter++){
        v = findNewVertexOfDegreeZero();
        output(v);
        for each w adjacent to v
              w.indegree--;
        mark_as_outputted(v);
  }
}
```

$O(|V|+|E|)$

$|V|$

naive! scan from top to bottom

$O(|V|)$

$O(1)$

$O(d_v)$

$O(1)$

*Runtime:*

$$O(|V|+|E|+|V|(|V|+d_v))$$

$$= O(|V|+|E|+|V|^2+|E|) \quad \text{since } |V|d_v = |E|$$

$$= O(|V|^2+|E|) = O(|V|^2)$$

# Topological Sort: Take Two

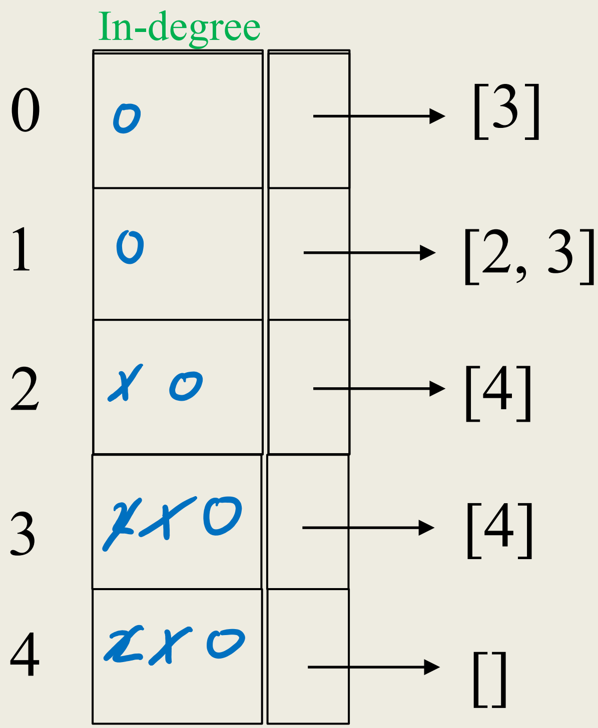
1. Label each vertex with its in-degree
2. Initialize a queue Q to contain all in-degree zero vertices
3. While Q not empty
   a. *v* = Q.dequeue; output *v*
   b. For each vertex *u* adjacent to v:
      - Reduce the in-degree of *u*
      - If new in-degree *u* is zero, Q.enqueue(*u*)

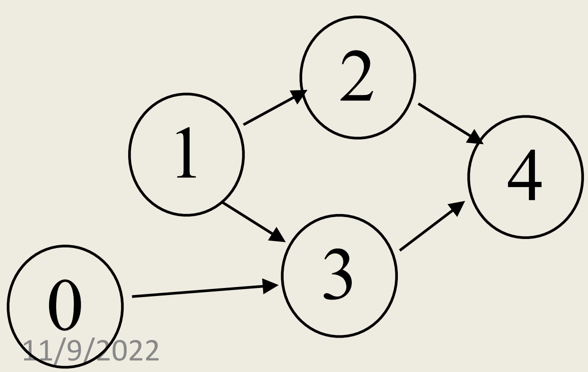
Better way to identify next 0-degree

* Relies on insight that the only ones' whose in-degree changes are the neighbors of u.

*Doesn't have to be a queue (stack, set, etc. all ok)

# Topological Sort: Take Two




1. Label each vertex with its in-degree
2. Initialize a queue Q to contain all in-degree zero vertices
3. While Q not empty
   a. *v* = Q.dequeue; output *v*
   b. For each vertex *u* adjacent to v:
      - Reduce the in-degree of *u*
      - If new in-degree *u* is zero, Q.enqueue(*u*)

Q: [~~0~~, ~~1~~, ~~2~~, ~~3~~, ~~4~~]

0 1 2 3 4

```
topsort(){
  Queue q(NUM_VERTICES);
  Vertex v, w;

  labelEachVertexWithItsIn-degree();

  q.makeEmpty();
  for each vertex v
    if (v.indegree == 0)
      q.enqueue(v);

  while (!q.isEmpty()){
    v = q.dequeue();
    output(v);
    for each w adjacent to v
      w.indegree--;
      if (w.indegree == 0)
        q.enqueue(w);
  }
}
```

$O(|V| + |E|)$

$O(|V|)$

$|V|$

$O(1)$

$O(1)$

$O(d_v)$

$O(1)$

initialize the queue

get a vertex with indegree 0

insert new eligible vertices

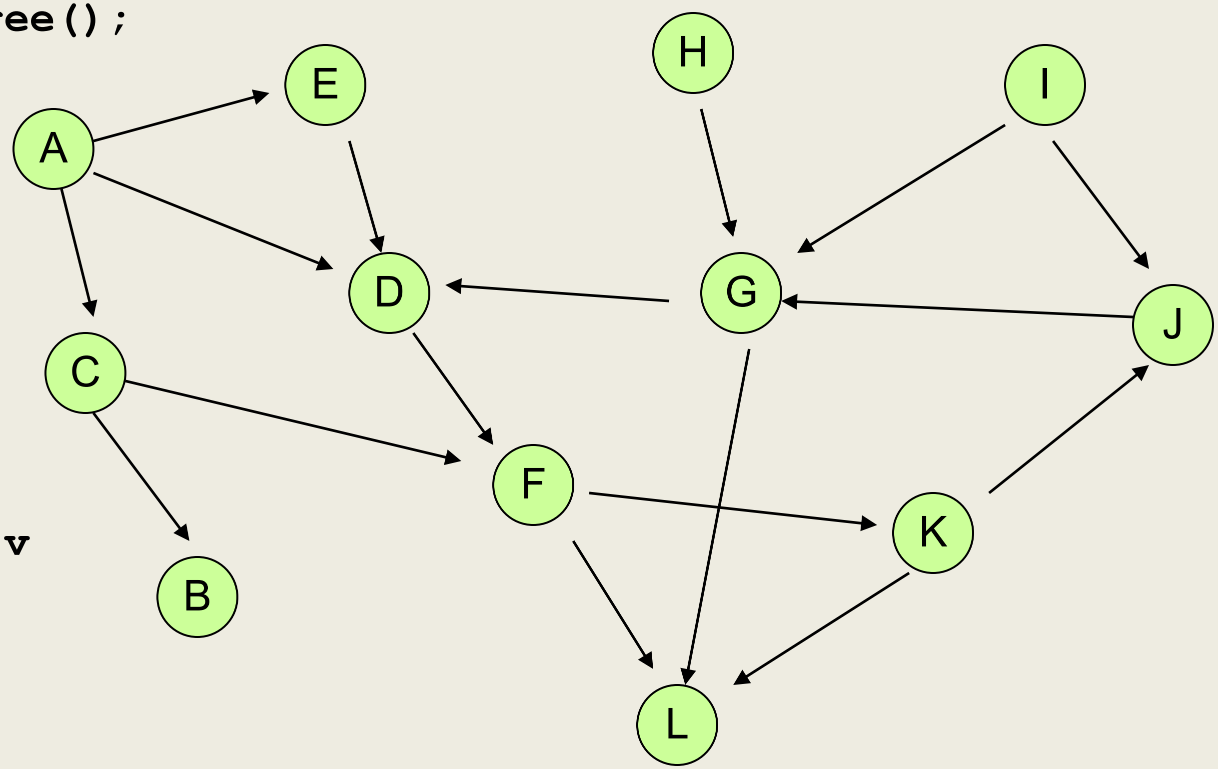
*Runtime?* $O(|V| + |E| + |V| + |V| \cdot d_v) = O(|E| + |V|)$

# Find a topological order for the following graph

```
Queue q(NUM_VERTICES);
labelEachVertexWithInDegree();

q.makeEmpty();
for each vertex v
  if (v.indegree == 0)
    q.enqueue(v);

while (!q.isEmpty()):
  v = q.dequeue();
  output(v);
  for each w adjacent to v
    w.indegree--;
    if (w.indegree == 0)
      q.enqueue(w);
```



# When can we find a topological sort of a directed graph?

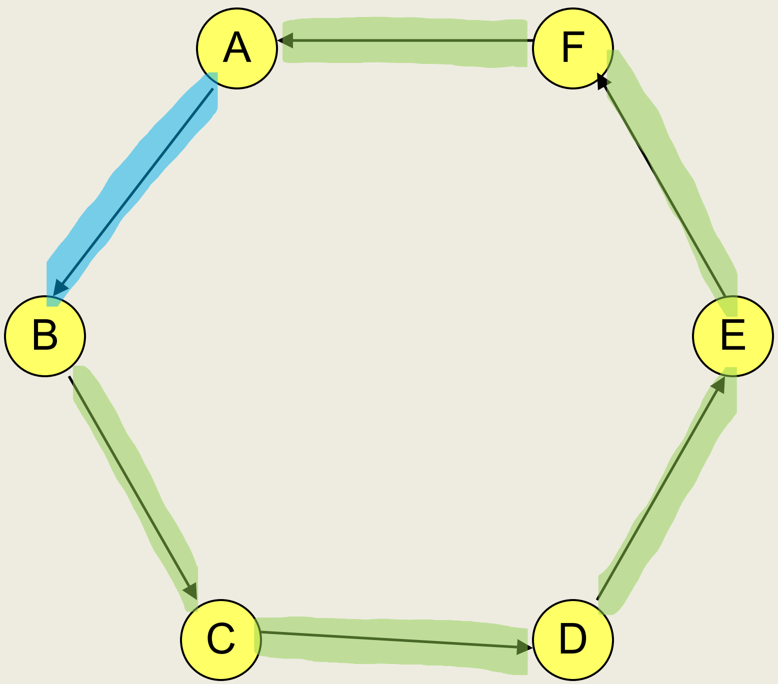
1. If the graph has a cycle, there is no topological sort
2. If the graph is acyclic, there is a topological sort

In other words:

A directed graph has a topological sort if and only if it is acyclic.

# 1. If a graph has a cycle, there is no topological sort

- Suppose there is a cycle $(A, B, C, \ldots, F, A)$
- Then $A$ must come before $B$ in any valid topological sort
- But $B$ must also come before $A$ in any valid topological sort
- So there is no valid sort!



# 2. If the graph is acyclic, there is a topological sort

We won't prove the entire statement. Instead…

Lemma: If a graph is acyclic, it has a vertex with in-degree 0

Proof:

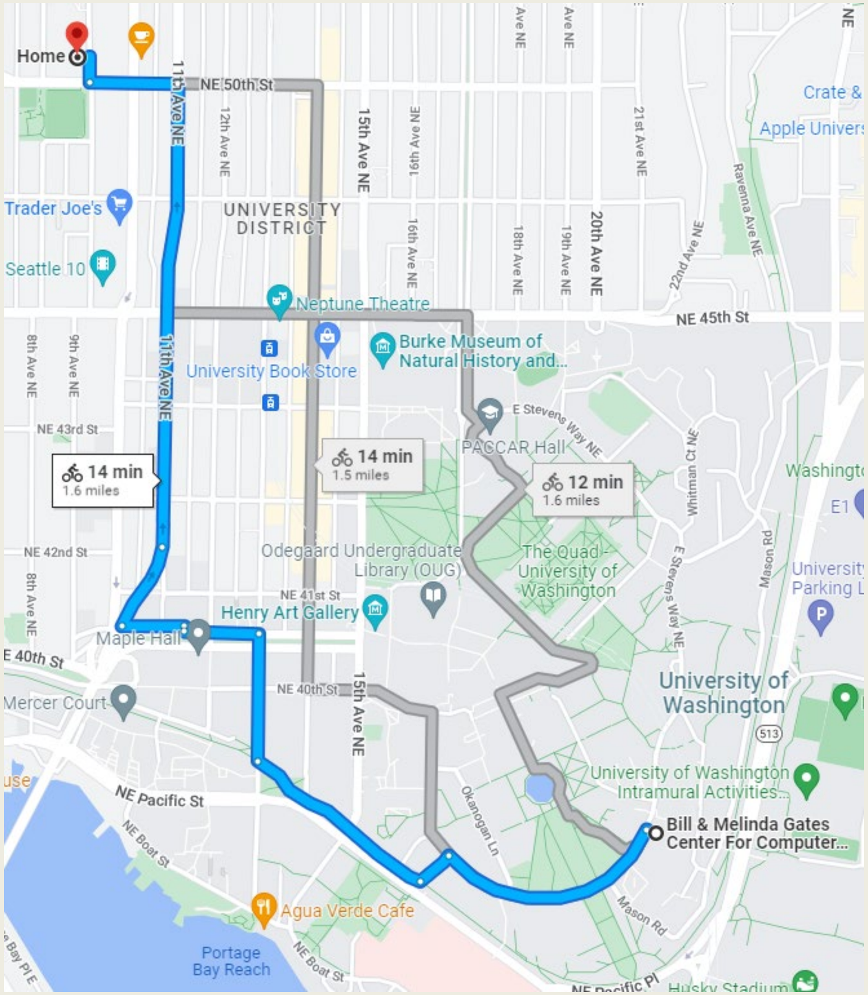
- Pick a vertex $v_1$, if it has in-degree 0 then done
- If not, let $(v_2, v_1)$ be an edge, if $v_2$ has in-degree 0 then done
- If not, let $(v_3, v_2)$ be an edge . . .
- If this process continues for more than $|V|$ steps, we have a repeated vertex, so we have a cycle

$v_2 \to v_n \to \cdots \to v_3 \to v_2 \to v_1$ Cycle!!

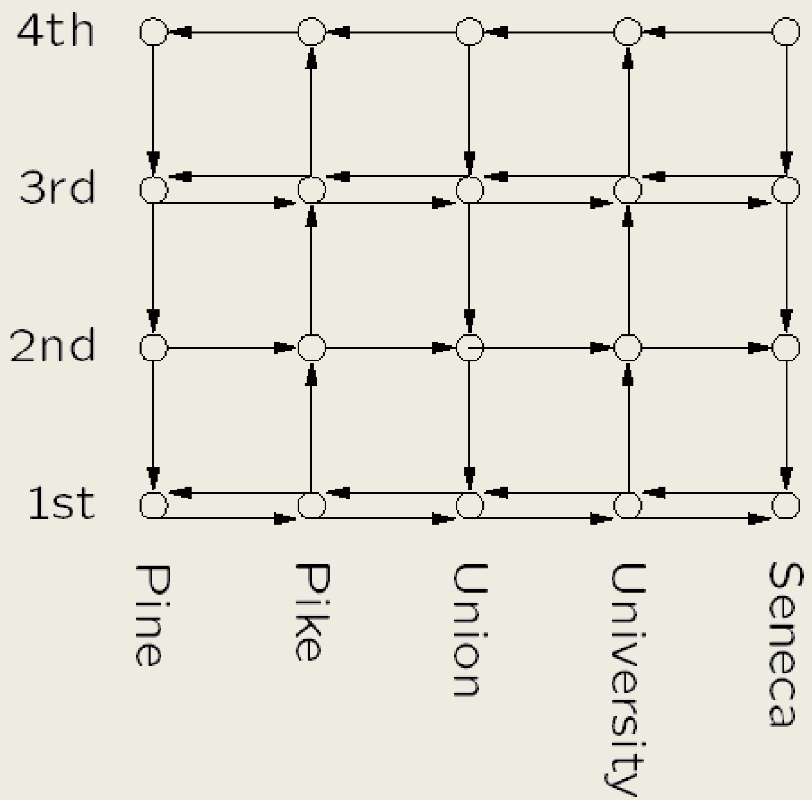
# Shortest Paths Problem

- Given a directed graph with edge costs and a starting vertex s, find the minimum cost path from s to every other vertex in the graph.
- Future results
  - Dijkstra’s algorithm solves the shortest paths problems if all costs are non-negative
  - Bellman-Ford’s algorithm solves the shortest paths problem if costs are allowed to be negative
    - Project 3 implements, and parallelizes Bellman-Ford

# Example: Find the shortest path



# Example: Bus Routes in Downtown Seattle




If we’re at 3rd and Pine, how can we get to

1st and University using Metro?

How about 4th and Seneca?

# The Shortest Path Problem

Given a graph $G$, and vertices $s$ and $t$ in $G$, find the shortest path from $s$ to $t$.

Two cases: weighted and unweighted.

For a path $p = v_0\ v_1\ v_2 \dots v_k$

- *unweighted length* of path $p = k$ (a.k.a. *length*)
- *weighted length* of path $p = \sum_{i=0..k-1} c_{i,i+1}$ (a.k.a. *cost*)

We will assume the graph is directed

# Single Source Shortest Paths (SSSP)

Given a graph *G* and vertex *s*, find the shortest paths from *s* to all vertices in G.

– How much harder is this than finding single shortest path from s to t? not much!
  - Most algorithms will have to find the shortest path to every vertex in the graph in the worst case
    – Although may stop early in some cases