L18 – Graphs

7/22/24

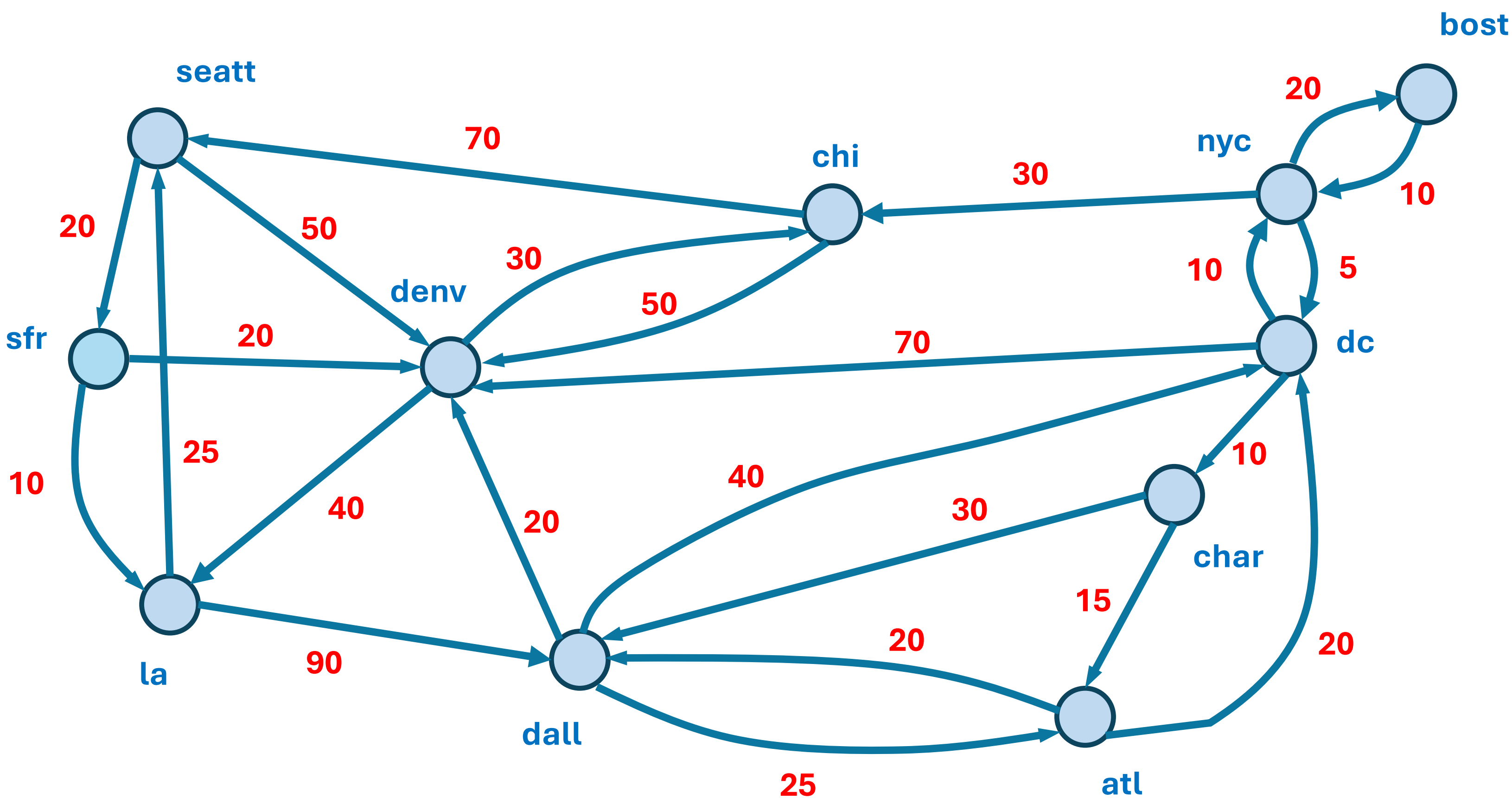
Announcements

- EX10 due tomorrow 7/23
- EX11 releases tonight, due LDOC
- QZ06 on Wednesday or Thursday (tentative)

Bipartite question

- Prove that trees (no parent pointer) are bipartite using induction
- Base case: single node is bipartite
- Inductive case: assuming tree with n nodes is bipartite, show that tree with n+1 nodes (i.e., add a leaf) is bipartite

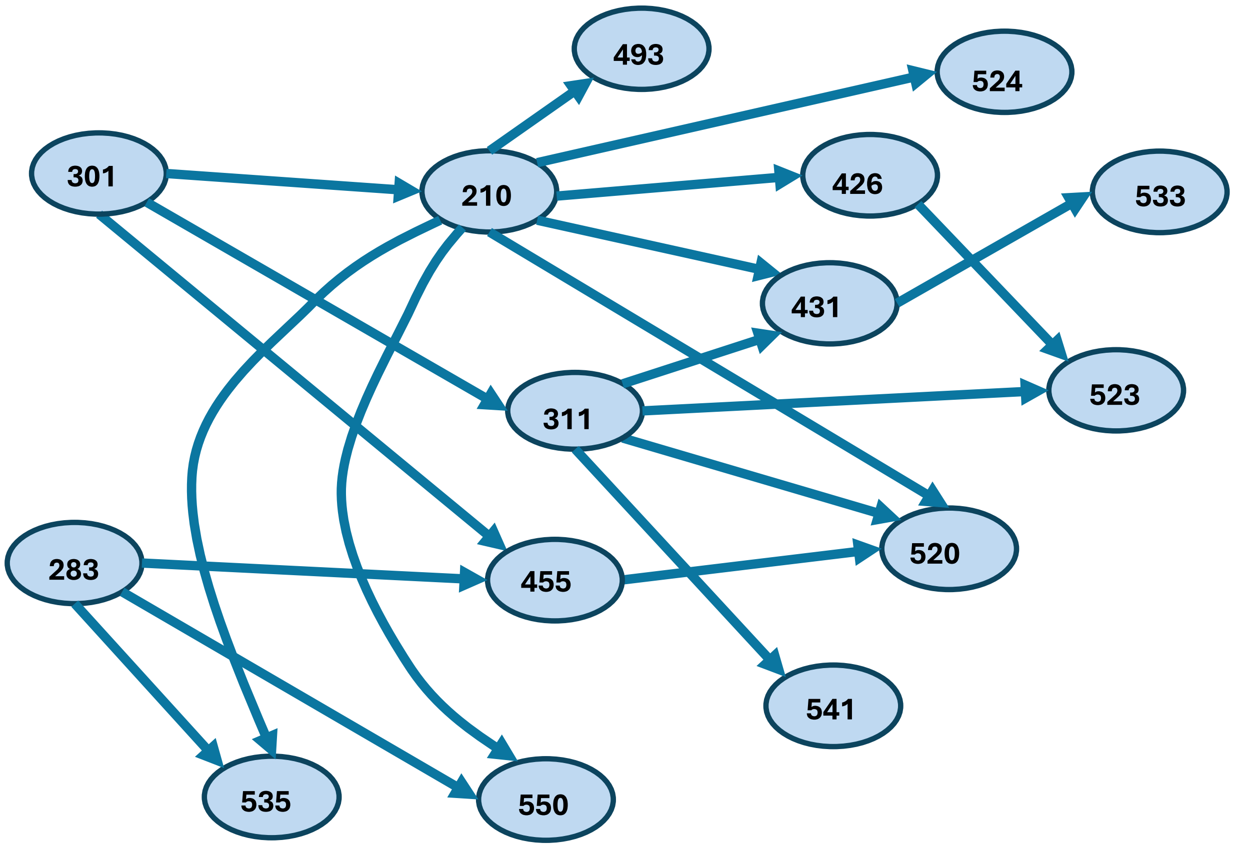
Modeling with graphs



Modeling with graphs, cont.

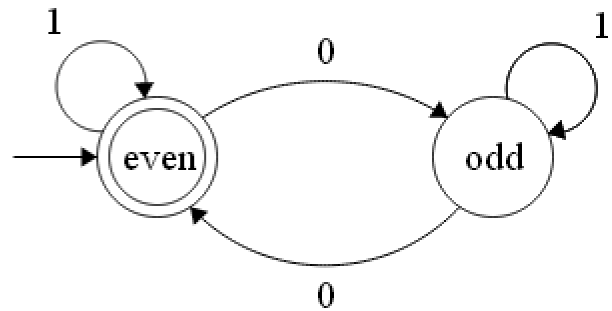
- What is the min cost from NYC to LA?
  - NYC –5> DC –10> Char –30> Dall –20> Denv –40> LA
  - 105
  - We'll learn some algorithms for determining this
- From any city, can I get to any other city?
  - I.e., is graph strongly connected?

Modeling with graphs, cont.



(numbers are outdated)

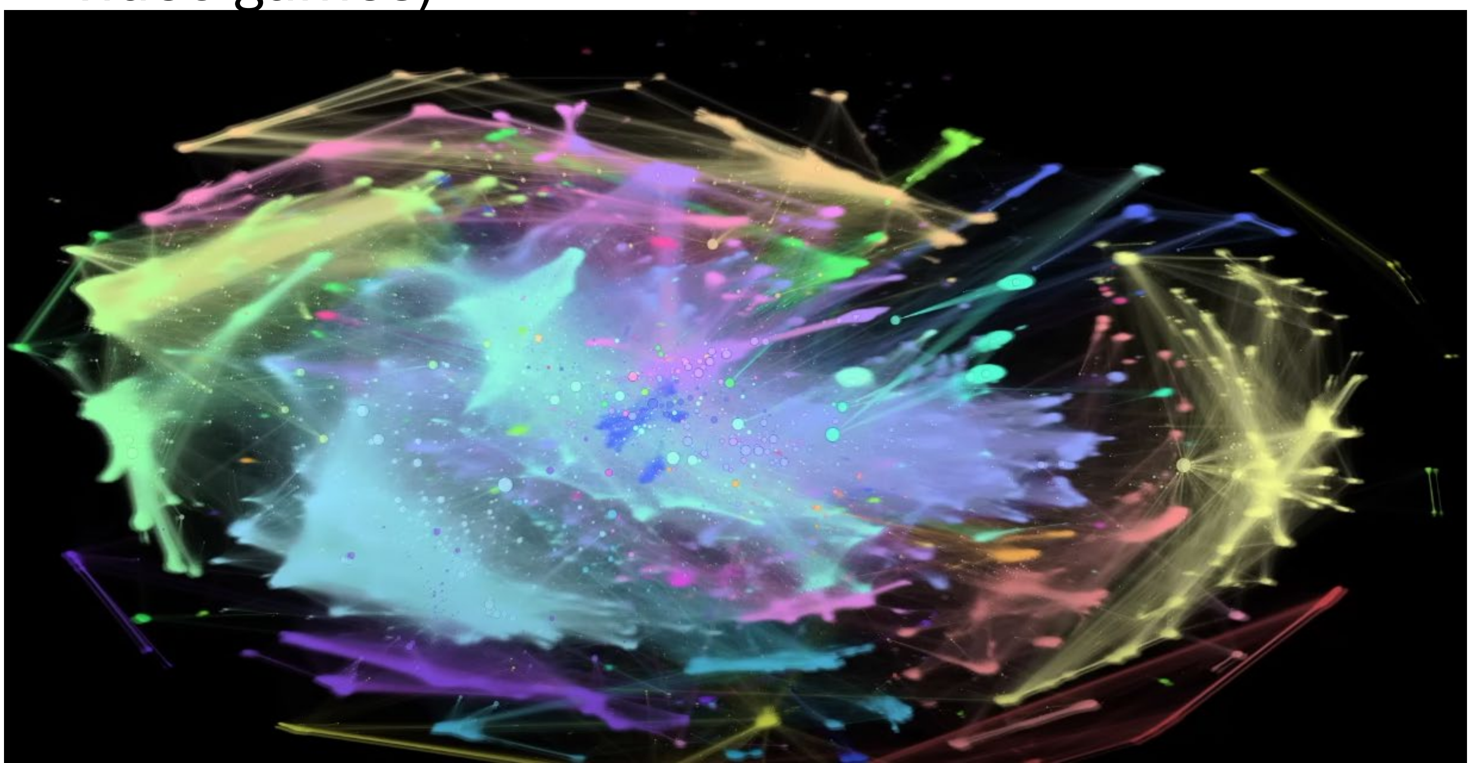
Modeling with graphs, cont.



- FSM (finite state automaton) – simple computer
  - Details in 455
- This one checks if a string has an even number of 0's
  - Vertices are "states"
  - Numbers on edges are not weights, they're input characters
- Find (cmd/ctrl + F) implemented with regex, implemented with FSM

Modeling with graphs, cont.

- Graph of Wikipedia
    - Each color is a community the author found, of which there are 28 (e.g., politics & law, football (soccer), video games)

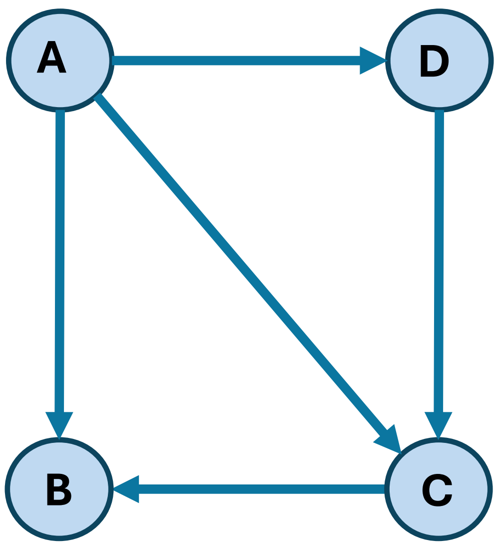


Graph representation

- 2 main approaches
- Adjacency matrix
  - Edge information tracked in 2D array
  - AM[i][j] represents an edge from vertex i to vertex j
- Adjacency list
  - Each vertex object maintains a list of edge objects

Adjacency matrix (unweighted)

- For N nodes, use N x N array of Boolean
- If AM[a,b] then (a,b)∈E
  - Edge from A to B



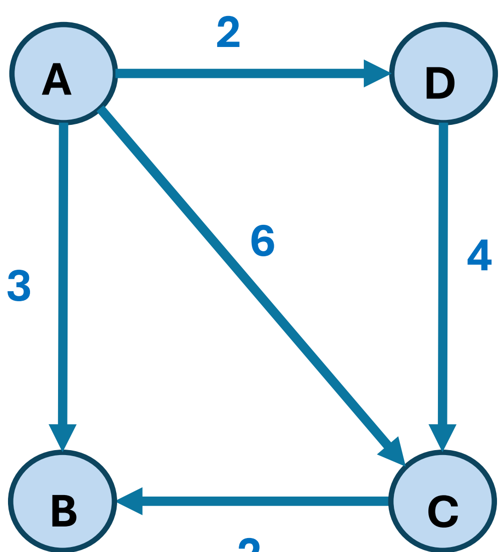
destination

source

	A	B	C	D
A	F	T	T	T
B	F	F	F	F
C	F	T	F	F
D	F	F	T	F

Adjacency matrix (weighted)

- Array elements now integer or float



destination

source	A	B	C	D
A	0	3	6	2
B	0	0	0	0
C	0	2	0	0
D	0	0	4	0

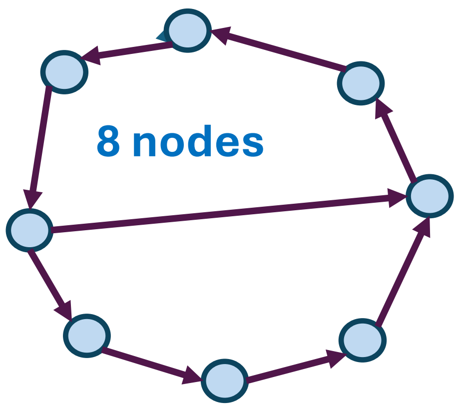
Adjacency matrix performance

- Pros and cons in terms of time and space complexity?
  - Use V = vertices, E = edges, |V| = # vertices, |E| = # edges
  1. How fast to determine if there's an edge from A to B?
  2. How fast to find all edges of A?
  3. Amount of space used?

1. O(1)
2. O(|V|)
3. $O(|V|^2)$

Adjacency matrix space problem

- Problem is wasted space

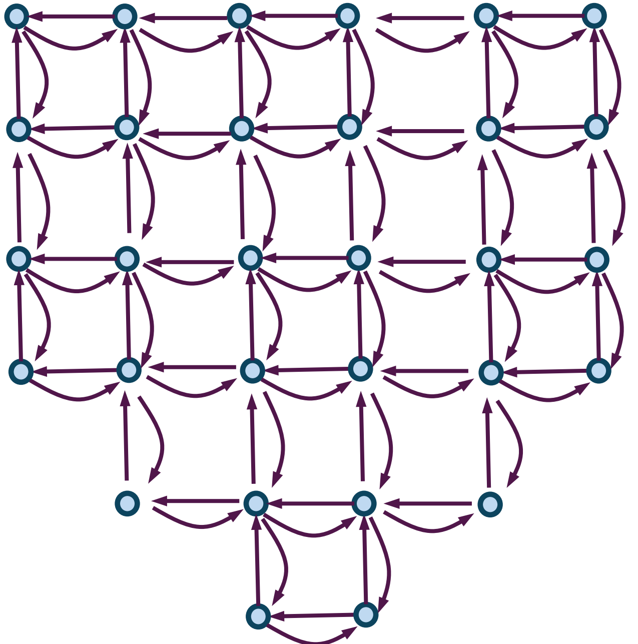


	T						
		T					
			T			T	
				T			
					T		
						T	
							T
T							

- 64 spaces, 55 not used
- $O(|V|^2)$ space
- If $|E| << |V|^2$, the graph is sparse
- If $|E| \approx |V|^2$, the graph is dense

Sparse or dense? Traffic example

- Let V be street intersections
- Let E be streets between V
- Model as digraph
  - Undirected might make more sense
- Is it sparse or dense?

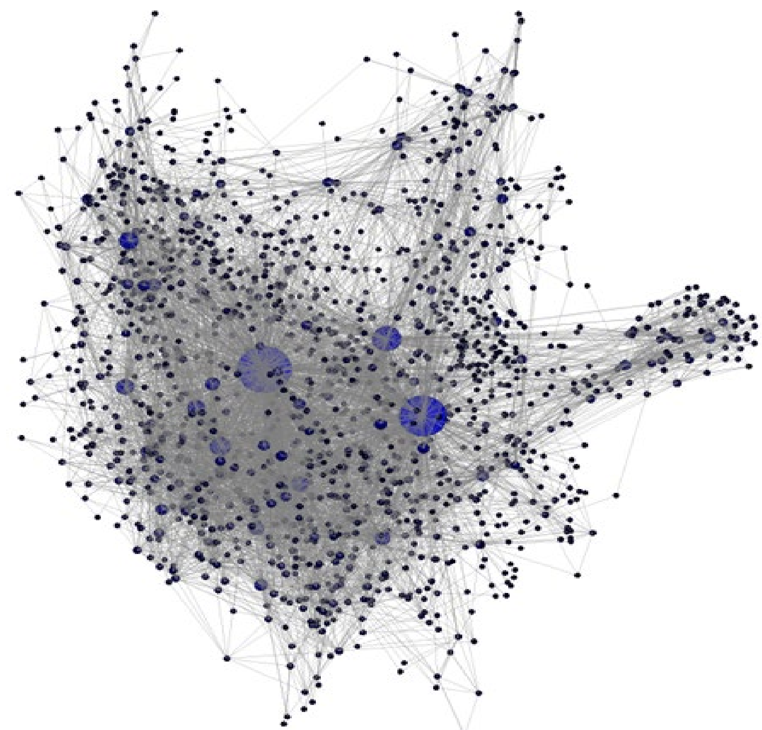


Traffic example explanation

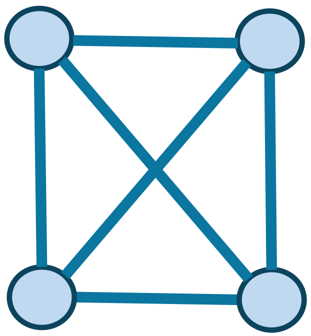
- Each node has maximum 4 edges out
  - Nodes on the perimeter have fewer than 4
- $|E| \approx 4|V|$
  - For large V, $|E| << |V|^2$
- Suppose there are 3000 intersections
- Matrix has $3000^2 = 9000000$ cells
- Number of cells used is 4(3000) = 12000
  - 0.13% used
- Sparse

Sparse or dense?

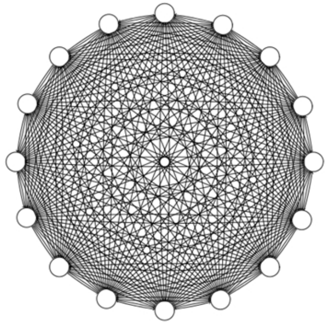
- 1538 nodes, 8032 edges
- $1538^2 = 2365444$
- 8032 << 2365444
- Sparse



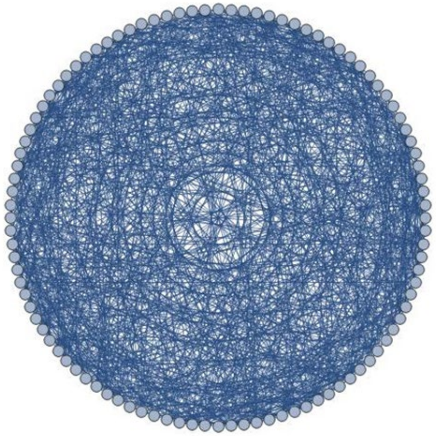
Complete graph density



- Complete graphs are always dense
- Recall that in a complete graph, node V has edges to every node but itself
- Clearly dense
- The math:
  - Complete undirected graph has $\binom{v}{2} = \frac{v(v-1)}{2}$ edges
  - O(|V|²)



K4	0	1	2	3
0	F	T	T	T
1	T	F	T	T
2	T	T	F	T
3	T	T	T	F



Adjacency matrix of undirected graph

- Represent adj. matrix of complete graph K4 as matrix where 1 denotes edge exists
- $$\begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}$$
- What is a matrix property that any adjacency matrix of an undirected graph has?

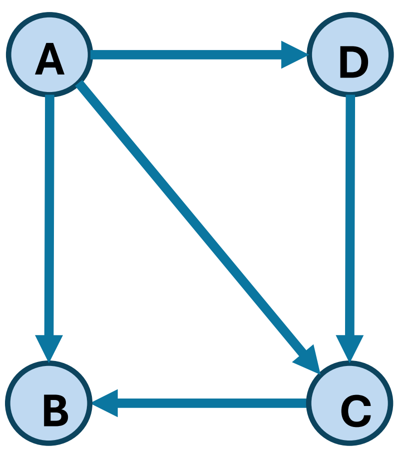
  - Symmetric
    - $A = A^T$
    - For every i, j, $a_{ij} = a_{ji}$

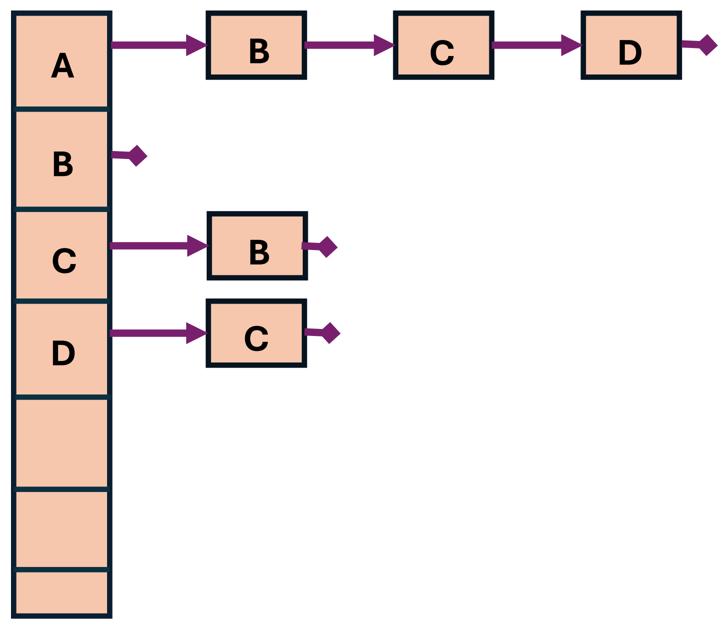
Adjacency matrix summary

- Easy to code
- Usually wastes most of its space
- When sparse, finding adjacent nodes is expensive
    - O(|V|) to find only a few adjacent vertices

Adjacency list

- Keep list of vertices
- Each vertex has a list of adjacent vertices
  - Remind you of anything we learned recently?



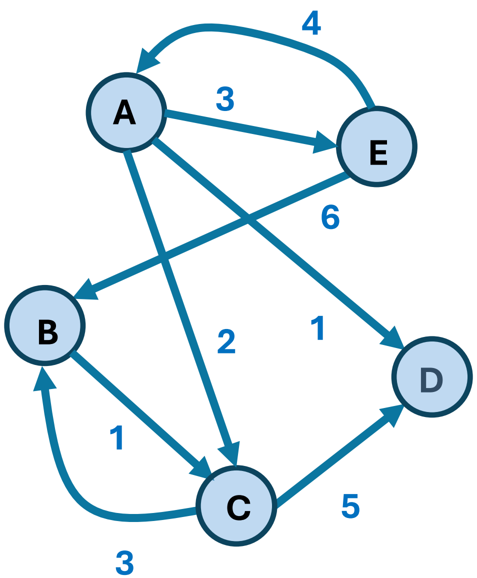


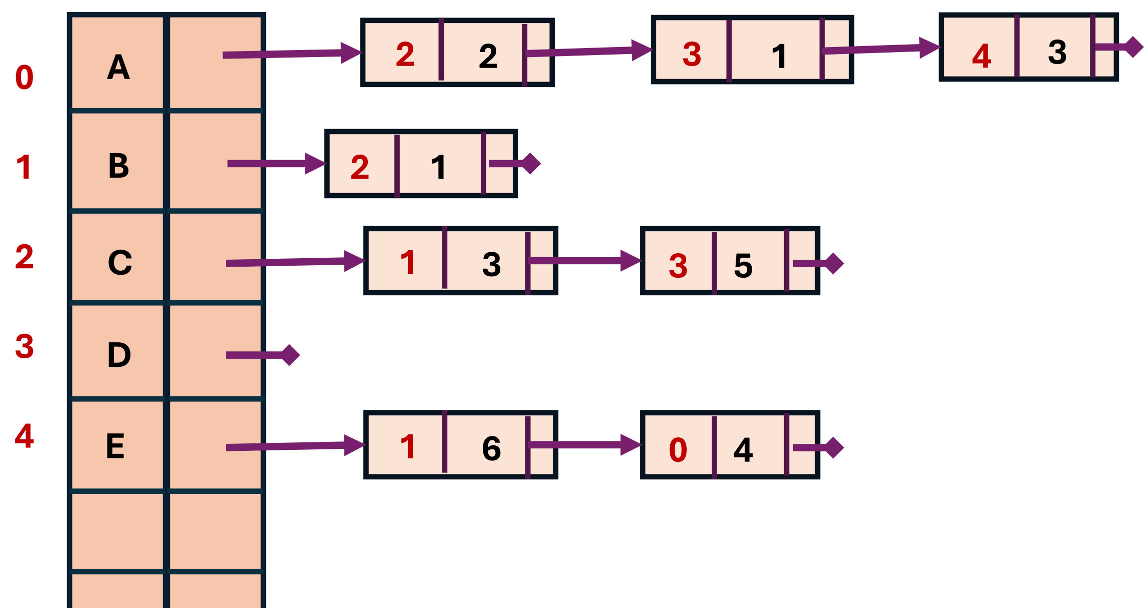
Adjacency list typing

- List of lists
  - List<List<Vertex>>
  - Get edges of vertex 0 with `al.get(0)`
- HashMap of lists
  - HashMap<String, List<Vertex>>
  - Suppose we represent node a with the String “a”
  - Then get its edges with `al.get(“a”)`
- Array of lists
  - List<Vertex>[]
  - `al[0]`
- Pros and cons?

Adjacency list (weighted)

- If edges are weighted, store the weight in the cell
- This example uses array for vertices and linked list for edges
  - What would HashMap for vertices look like?





Adjacency list performance

1. Amount of space used in terms of V and/or E?
2. Worst-case, find all vertices adjacent to some node v?
    1. Normally O(|list for v|)

1. O(|V| + |E|)
    1. We call this "linear" for graphs
    2. "size" of graph is |V| + |E|
2. O(|V|)
    1. Complete graph

Adjacency list efficiency

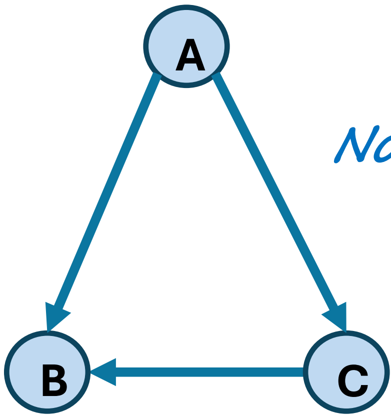
- Need a HashMap for vertices
  - See Adjacency list typing slide
- Many graph algorithms will be inefficient without
  - If you need to find a vertex, cannot afford to do O(|V|) search through all vertices list without making many algorithms become quadratic or worse
- May want similar hash structure for edges
  - E.g., HashMap<String, HashMap<String, Vertex>>
  - If a and b are nodes, to see if b is adjacent to a, do `map.get("a").containsKey("b")`
    - O(1)
    - Inner LinkedList for edges is instead O(|V|)
  - To get all edges of a, use a for-each loop, `keySet()`, etc., on `map.get("a")`

Topological sort (topo sort)

- First graph algorithm
- Computed for DAG G
- An ordering of all vertices in G such that if $(u,v) \in E$, then $u<v$ in the sort (u precedes v in the sequence)
- Every DAG has at least 1 topo sort
- Some have more than 1
- If a graph has a cycle, then it does not have a topo sort (why?)

Examples

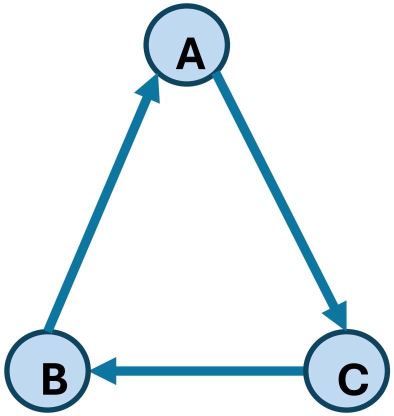
- In-degree of vertex b
  - Number of edges coming to b
  - Number of edges (a, b) ∈ E for distinct a
  - Number of vertices b is adjacent to



Note that A has no in-edges, and B has no out-edges

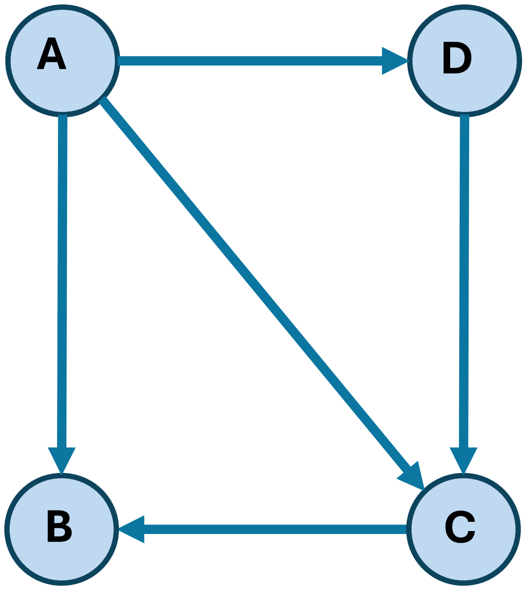
only 1 topo sort
A, C, B

Check:
(A,B) in E... A<B in sort
(A,C) in E... A<C in sort
(C,B) in E... C<B in sort

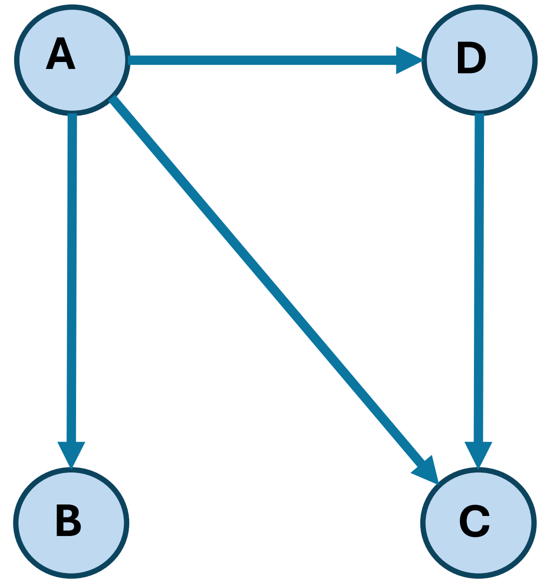


no topo sort
Note that every vertex has at least one in-edge and one out-edge

Examples



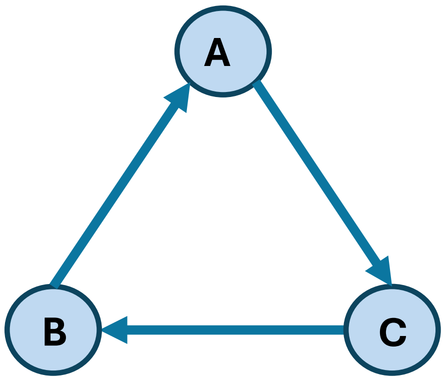
only 1 topo sort
A, D, C, B



3 topo sorts
A, B, D, C
A, D, C, B
A, D, B, C

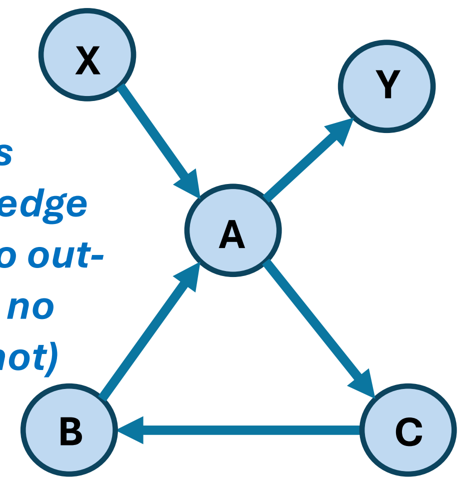
Properties

Topo sort -> at least one vertex with no in-edge and
at least one vertex with no out-edge

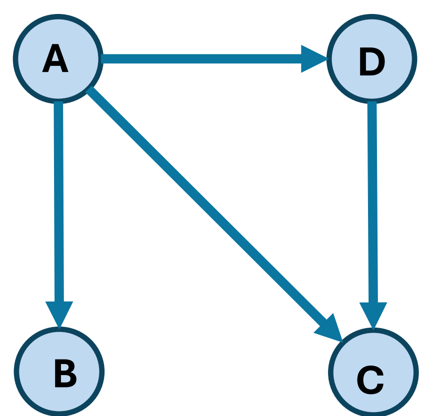


If all vertices have both in-edges and out-edges, there is no Topo Sort

Not sufficient:* *If there is some vertex with no in-edge and some vertex with no out-edge, there may still be no Topo Sort (may or may not)

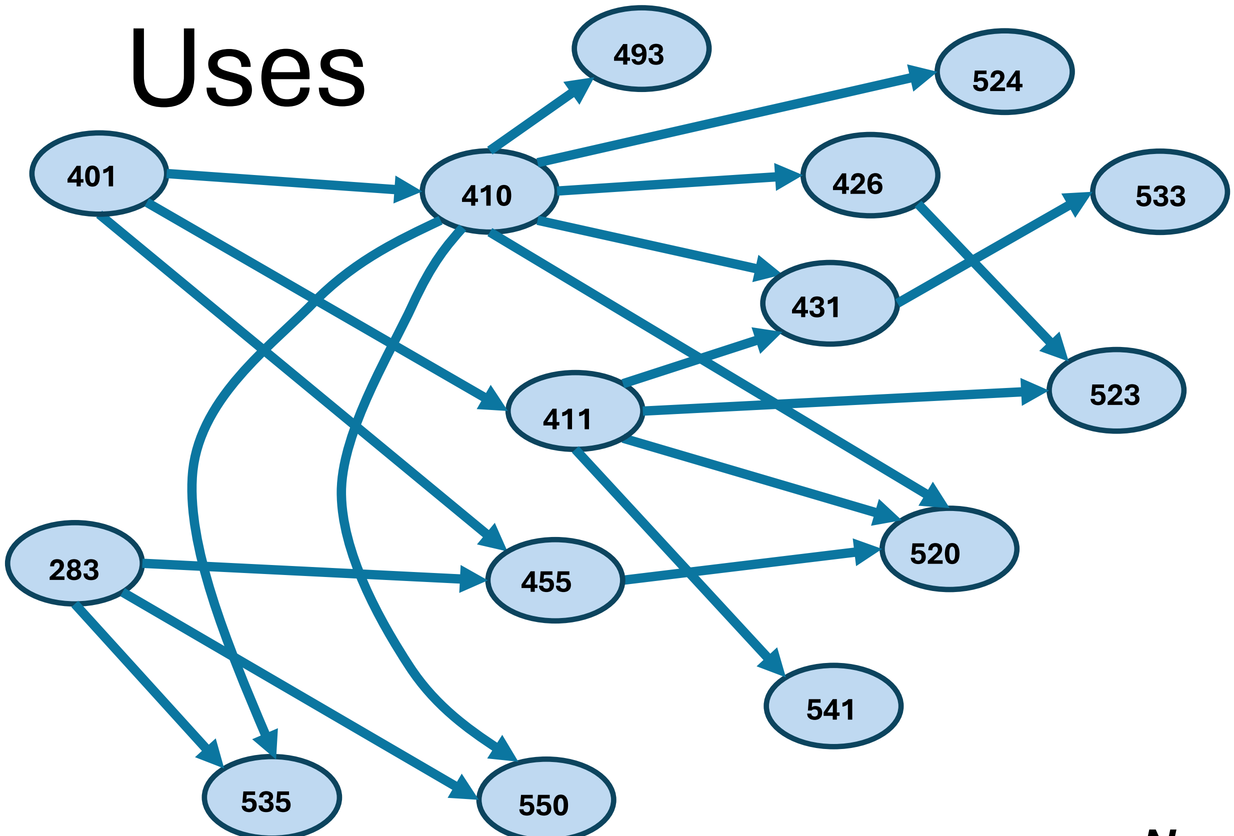


Necessary:* *If a graph has a Topo Sort, there is some vertex with no in-edge and some vertex with no out-edge



Necessary and sufficient meaning

Uses



Use topo sort for structures like course pre-requisites

The sorts give OK orders of classes to take

Numerous topological sorts

283, 401, 455, 410, 426, 535, 493, 520, 550, 411, 431, 533, 541, 520, 523

401, 283, 455, 410, 426, 535, 493, 520, 550, 411, 431, 533, 541, 520, 523

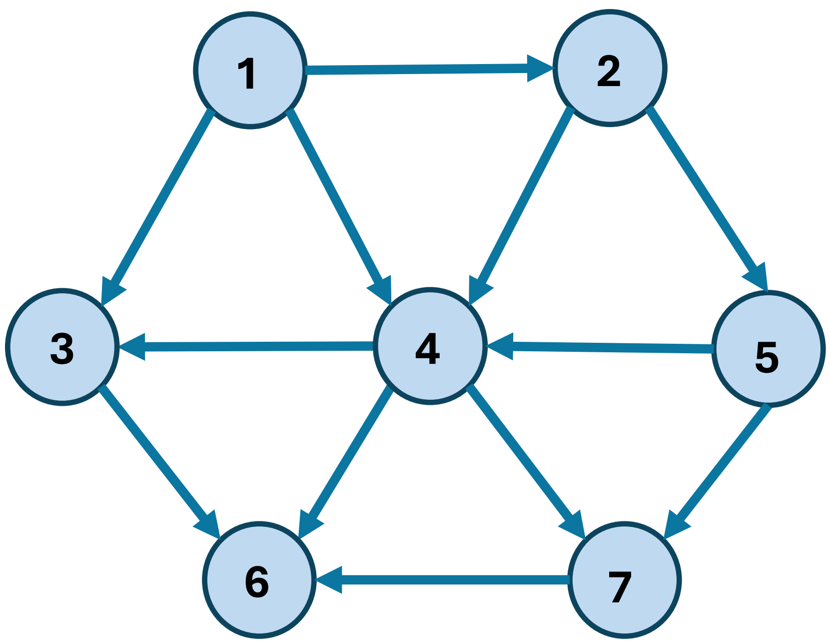
401, 283, 455, 410, 426, 535, 493, 520, 550, 411, 431, 533, 541, 523, 520

Outdated numbers, but good example

etc.

Algorithm to find a topo sort

- We'll use this graph for our example



Two topo sorts

S_1: 1, 2, 5, 4, 3, 7, 6

S_2: 1, 2, 5, 4, 7, 3, 6

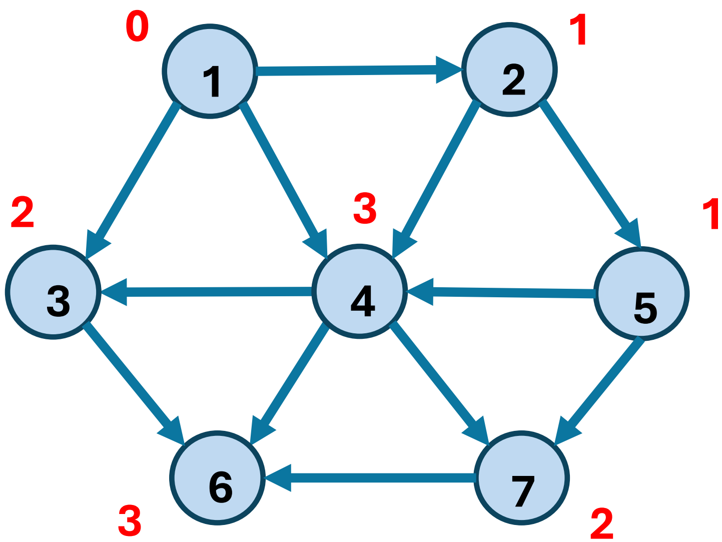
Algorithm

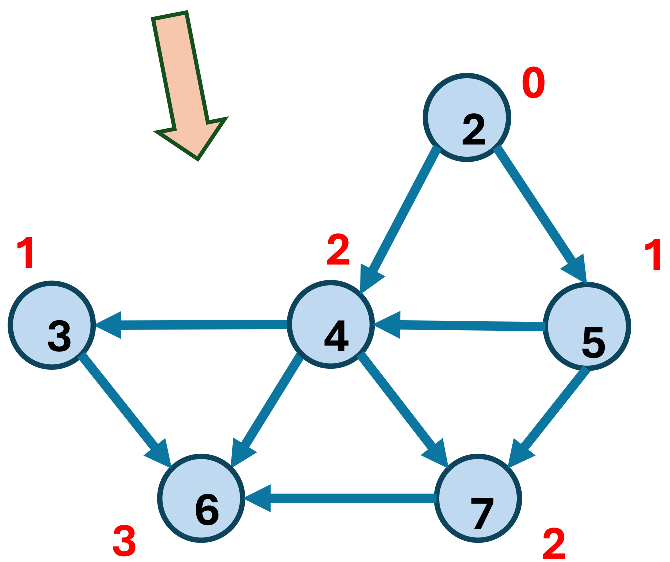
- Find a topo sort in $O(|V| + |E|)$ (linear time)
- Assume $G = (V, E)$ was built using adjacency list and that in-degree of each node was stored during the build

1. Any nodes with in-degree 0? If no, then cycle, done.
2. Pick any node v with in-degree 0. Put v into the TS
3. Decrement in-degree of any node w where $(v, w) \in E$ (essentially remove v and its out-edges from graph)
4. More nodes? Goto step 1, else done

Execute on Example

TS: 1 , 2



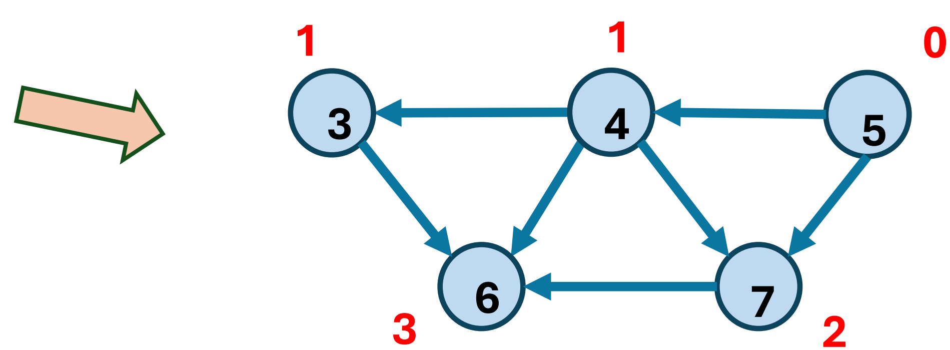


Node 1 has in-deg 0 so put it into TS

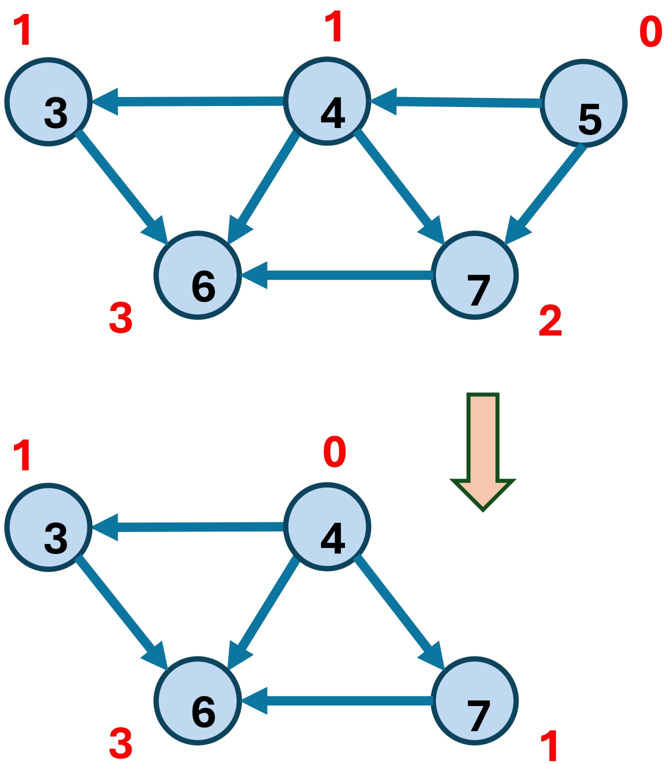
Remove node 1, out edges, and redo in-degrees

Node 2 has in-deg 0 so put it into TS

Remove node 2, out edges, and redo in-degrees



Execute on Example



TS: 1 , 2 , 5 , 4 , 7 , 3 , 6

Node 5 has in-deg 0 so put it into TS

Remove node 5, out edges, and redo in-degrees

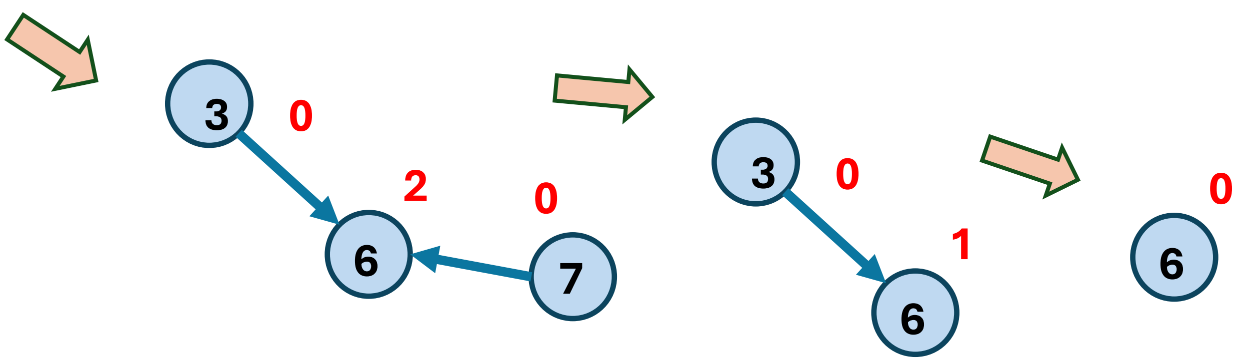
Node 4 has in-deg 0 so put it into TS

Remove node 4, out edges, and redo in-degrees

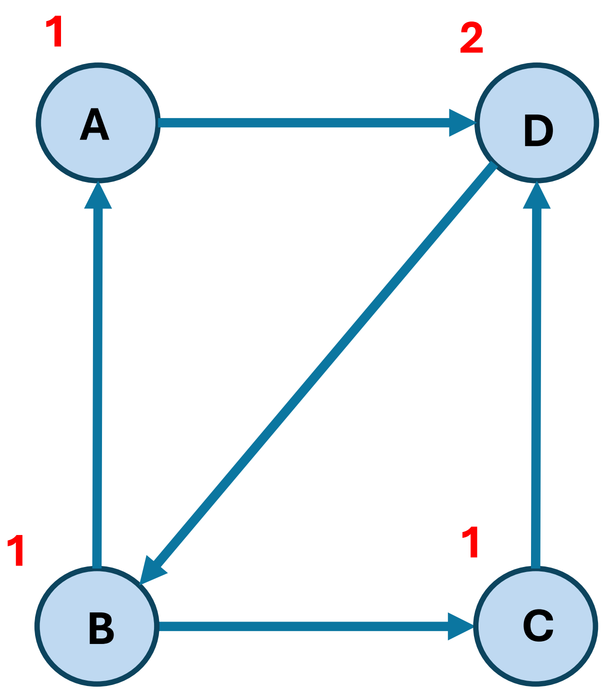
Nodes 3 and 7 have in-deg 0 so pick one, put in TS

Node 3 into TS, remove it

Node 6 into TS,
remove, done



What happens with cycles?

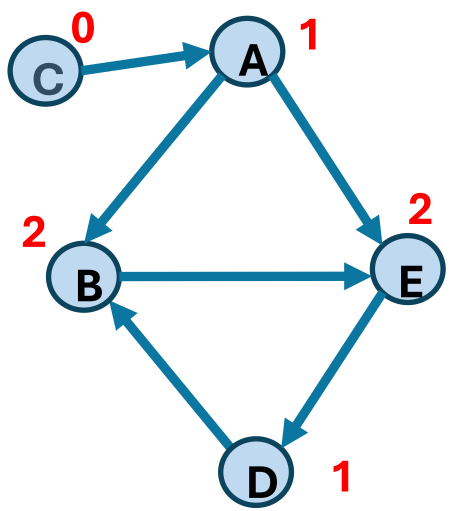


TS:

Can't even start this one

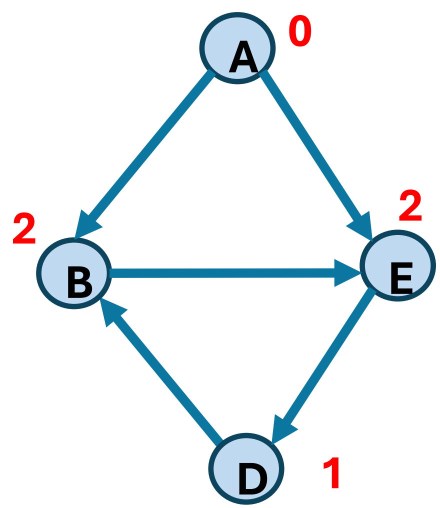
*There are **no** vertices with in-degree 0*

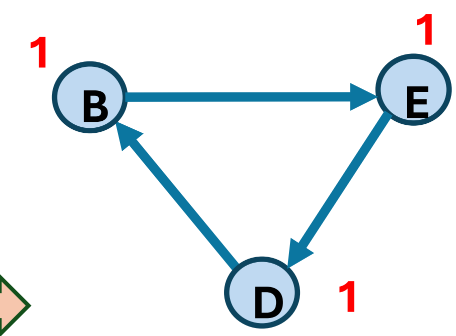
What happens with cycles?



TS: C A

Length is not |V|, invalid

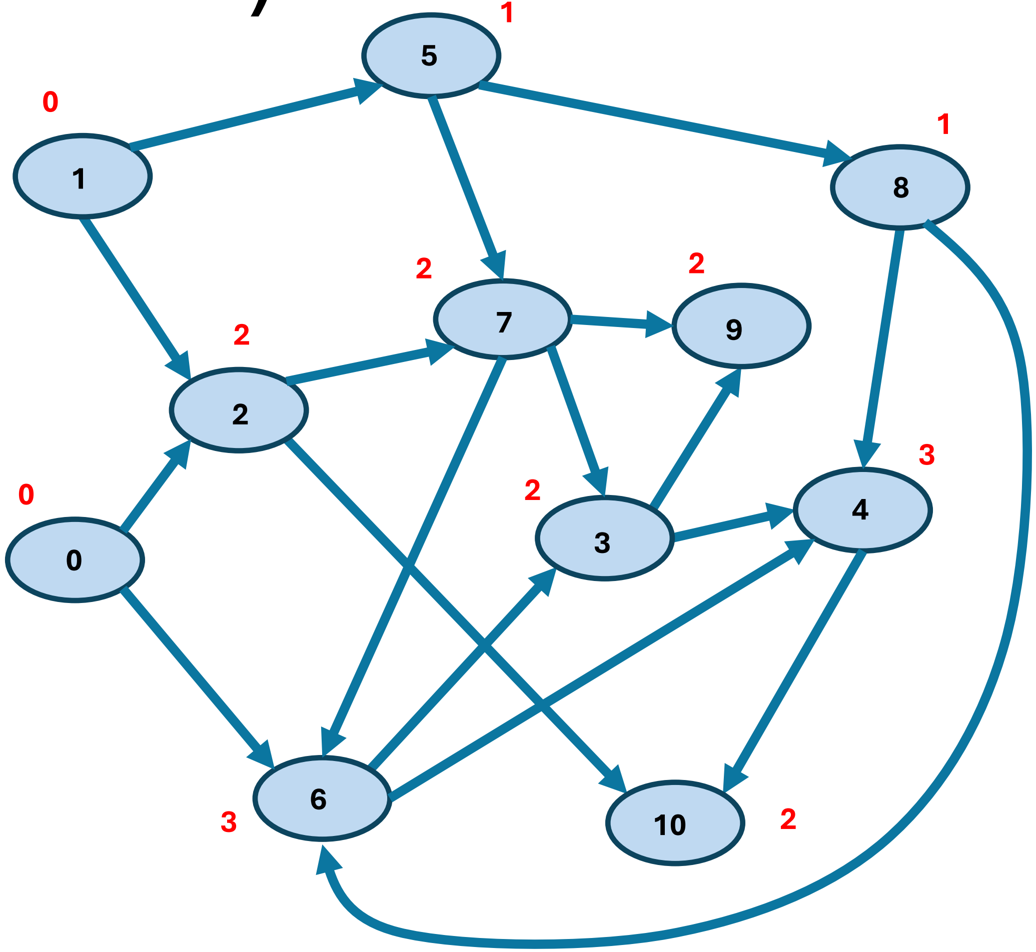




There are no vertices with in-degree 0

However there are unprocessed nodes
So, we have found cycle(s)
So, no Topo Sort possible

Try on this graph (several topo sorts)



Topo Sort:

1 ***choice: 0, 1***
5 ***choice: 0, 5***
0 ***choice: 0, 8***
8
2
7
6
3
4
10 ***choice: 9, 10***
9

Topo sort algorithm analysis

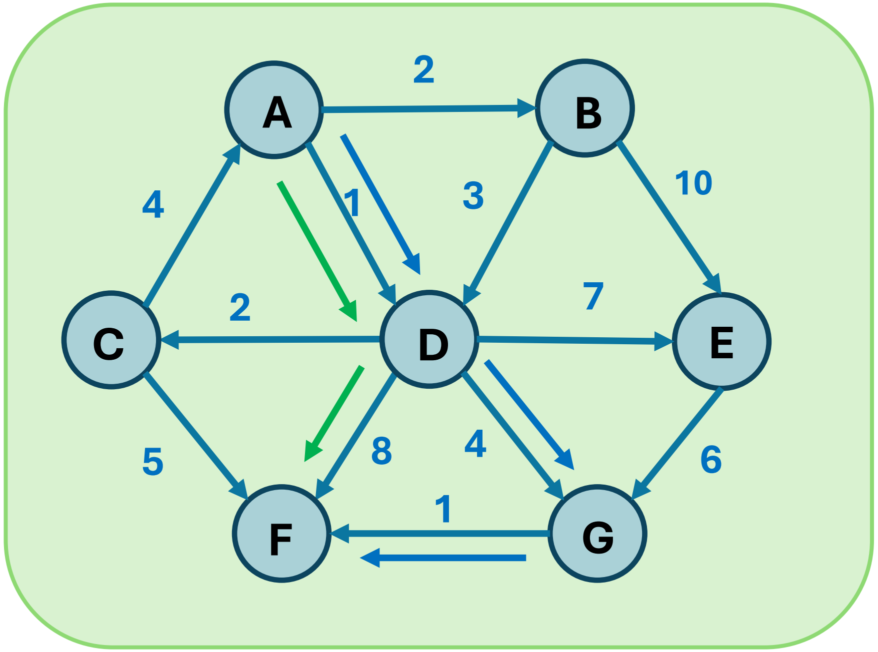
- $O(|V| + |E|)$
  - Examine and remove each vertex, each removal $O(1)$ so $O(|V|)$
  - While examining, operate on each out edge once by decrementing in-degree of destination vertex, so $O(|E|)$
- This is what we hope for, actually depends on several factors
  - Remove each vertex $O(1)$
  - Decrement in-degree of destination vertex $O(1)$
  - Find vertex with in-degree 0 $O(1)$
    - If you have to do a linear search through all vertices, then $O(|V|)$, and the algorithm's complexity is then $O(|V|^2 + |E|)$

Efficient topo sort impl

- Compute initial in-degree of each vertex while graph is built
  - O(|E|)
  - While building graph, while making the entry for vertex u, for each of its edges (u, v_i), access Vertex v_i in O(1) and increment its in-degree (store it as field in Vertex class) in O(1)
- Scan through all vertices and identify all with in-degree 0, add to queue
  - O(|V|)
- While the 0-in-degree queue is not empty
  - Take a vertex off the queue and add it to the topo list
  - Examine each edge and "remove" it by decreasing the in-degree associated with the edge's destination
  - If in-degree of a destination vertex falls to 0, add it to the 0-in-degree queue
- When 0-in-degree queue is empty, if topo list does not contain |V| vertices, then must have found a cycle, no topo sort possible
- Otherwise, topo list is valid
- Now O(|V| + |E|)

Shortest path

- Many problems require us to find shortest path from vertex v to vertex w
  - Simple example, road navigation
- Look at 2 situations
  - Digraph with unweighted edges
    - Weight 1 on all (want shortest path length)
  - Digraph with weighted edges
    - Want lowest cost



Digraph Example

Going from A to F

Weighted: shortest path is A, D, G, F with a cost of 6

Unweighted: shortest path is A, D, F with a length/cost of 2

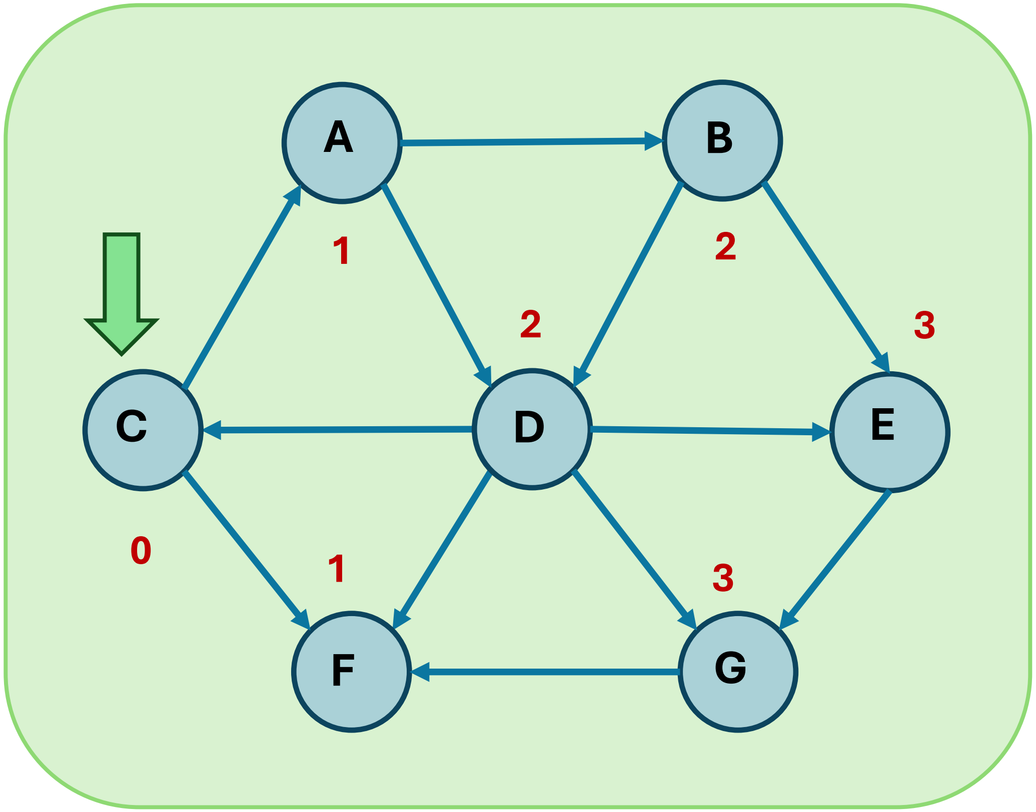
Unweighted shortest path

- Input: Unweighted digraph G = (V, E), start vertex s where $s \in V$
- Output: shortest path(s) from s to every other vertex
- Unweighted algorithm in $O(|V|^2)$ fairly simple
  - Adding weights complicates things, Dijkstra's algorithm

Bad unweighted shortest path

- Recognize that no shortest path can be longer than |V| - 1
- Run a loop with len going from 0 to |V| - 1 (inclusive)
- In loop, go through all nodes and when we find one with distance “len”, we mark all unmarked adjacent nodes with distance “len + 1”
- Double nested loops $O(|V| - 1) * O(|V|)$ is $O(|V|^2)$

(Bad) Unweighted Shortest Path



Output is a graph with each node being labeled with the shortest distance from C

C is marked 0 to start

currDist = 0,
find all nodes marked 0

- **We find C, and mark adjacencies 0+1**

currDist = 1,
find all nodes marked 1

- **We find A, and mark adjacencies 1+1**
- **We find F, no adjacencies**

currDist = 2,
find all nodes marked 2

- **We find B, and mark unmarked adjacencies 2+1**
- **We find D, and mark unmarked adjacencies 2+1**