# Graphs and Topological Sort

CSE 332 Spring 2021

**Instructor:** Hannah C. Tang

**Teaching Assistants:**

| | | |
|---|---|---|
| Aayushi Modi | Khushi Chaudhari | Patrick Murphy |
| Aashna Sheth | Kris Wong | Richard Jiang |
| Frederick Huyan | Logan Milandin | Winston Jodjana |
| Hamsa Shankar | Nachiket Karmarkar | |

# Announcements

- Please reach out to course staff if you are struggling for any reason

# Lecture Outline

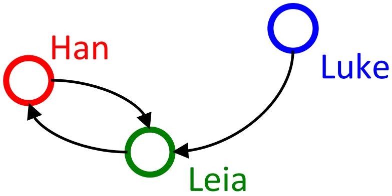
- Graphs
  - **Definitions**
  - Representation: Adjacency Matrix
  - Representation: Adjacency List
  - Algorithms over Graphs
- Topological Sort

# Graphs

- A ***graph*** is represents relationships among items
  - Very general definition because very general concept
- A ***graph*** is a pair: **G = (V, E)**
  - A set of ***vertices***, also known as ***nodes***

    $\mathbf{V = \{v_1, v_2, \ldots, v_n\}}$
  - A set of ***edges***, possibly ***directed***

    $\mathbf{E = \{e_1, e_2, \ldots, e_m\}}$
    - Each edge $\mathbf{e_i}$ is a pair of vertices $\mathbf{(v_j, v_k)}$
    - An edge "connects" the vertices




```
V = {Han, Leia, Luke}
E = {(Luke, Leia),
     (Han, Leia),
     (Leia, Han)}
```

# gradescope

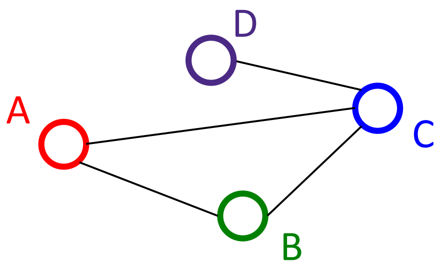
gradescope.com/courses/256241

- For one of the following, what are the ***vertices*** and the ***edges***?
  - Web pages with links
  - Facebook friends
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Family trees
  - Course pre-requisites

- **Wow!** Using the same algorithms for problems across so many domains sounds like "core computer science and engineering"

# Undirected Graphs

- In ***undirected graphs***, edges have no specific direction
  - Edges are always “two-way”

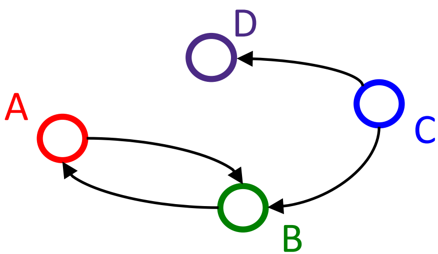



- Thus, `(u,v) ∈ E` implies `(v,u) ∈ E`
  - Only one of these edges needs to be in the set; the other is implicit

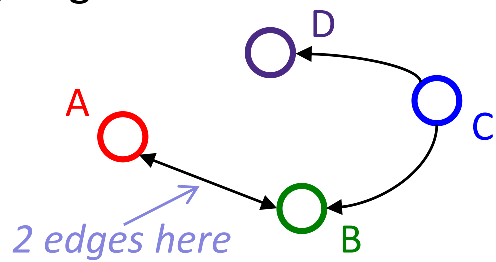
- ***Degree*** of a vertex: number of edges containing that vertex
  - i.e.: the number of adjacent vertices

# Directed Graphs

- In ***directed graphs*** (aka ***digraphs***), edges have a ***direction***




**or**




- Thus, `(u,v) ∈ E` does <u>*not*</u> imply `(v,u) ∈ E`
  - `(u,v) ∈ E` means `u → v`; u is the ***source*** and `v` the ***destination***

- ***In-Degree*** of a vertex: number of in-bound edges
  - i.e.: edges where the vertex is the destination
- ***Out-Degree*** of a vertex: number of out-bound edges
  - i.e.: edges where the vertex is the source

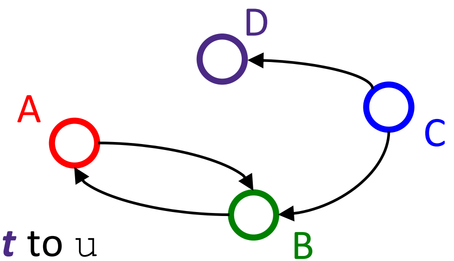
# Self-edges "selvedge"

- A ***self-edge*** (aka a ***loop***) is an edge of the form **`(u,u)`**

- Depending on the use/algorithm, a graph may have:
  - No self edges
  - Some self edges
  - All self edges (therefore often implicit, but we will be explicit)

- A node can have a degree / in-degree / out-degree of zero

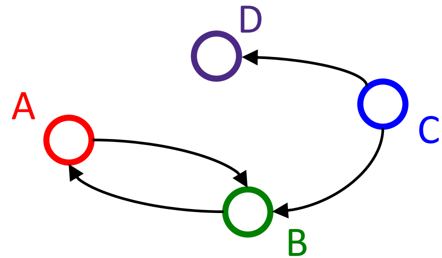
# Adjacency (1 of 2)



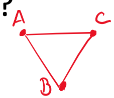

- If `(u,v) ∈ E`
  - Then `v` is a ***neighbor*** of `u`, i.e., `v` is ***adjacent*** to `u`
  - For directed edges, order matters
    - `u` is not adjacent to `v` unless `(v,u) ∈ E`

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

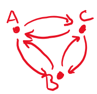
# Adjacency (2 of 2)




- For a graph G = (V,E):
  - |V| is the number of vertices
  - |E| is the number of edges
    - Minimum size?
      - *0*
    - Maximum size for an undirected graph with no self-edges?
      - |V||V-1|/2 ∈ O(|V|²)




    - Maximum for a directed graph with no self-edges?
      - |V||V-1| ∈ O(|V|²)




    - If self-edges are allowed, add |V| to the answers above (applies to both undirected and directed graphs)

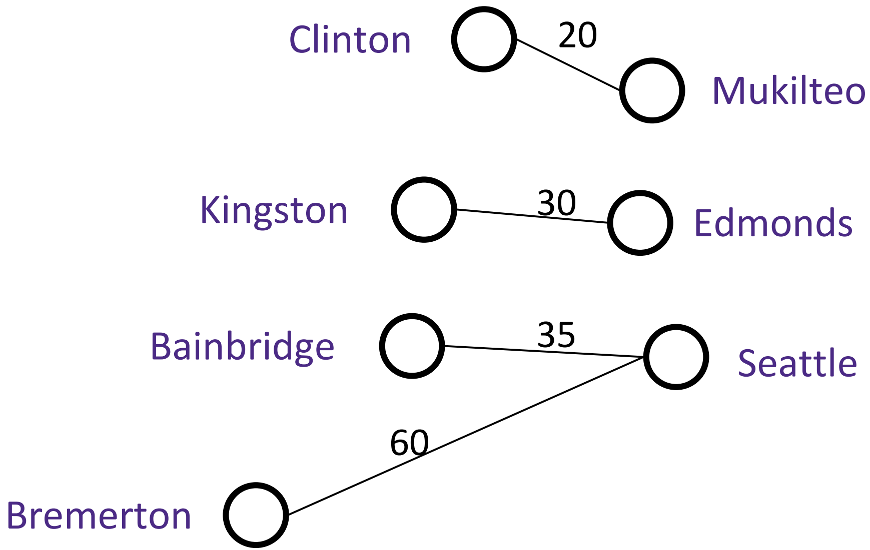
# gradescope

gradescope.com/courses/256241

- For one of the following, which would use ***directed edges***? Which would have ***self-edges***? Which might have ***0-degree nodes***?
  - Web pages with links
  - Facebook friends
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Family trees
  - Course pre-requisites

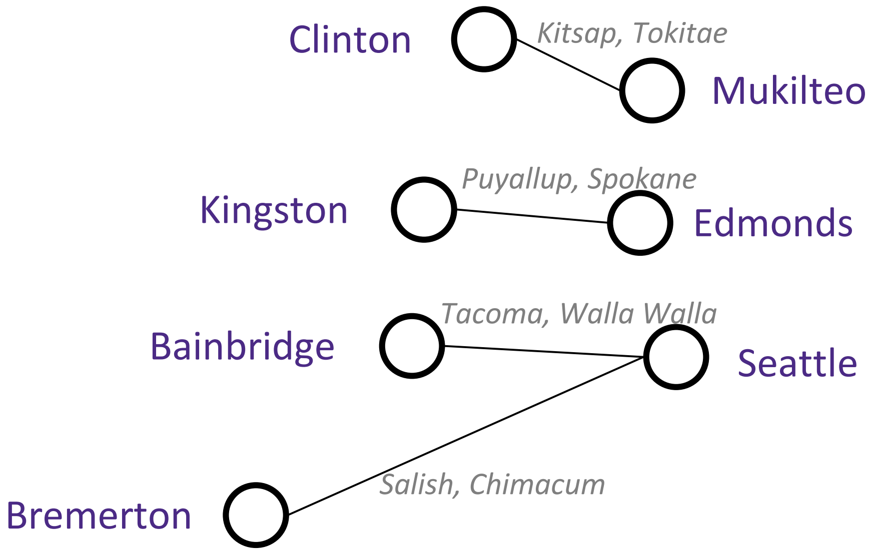
# Weighted Graphs

- In a weighed graph, each edge has a ***weight*** a.k.a. ***cost***
  - Typically numeric (most examples will use ints)
  - Some graphs allow *negative weights*; many don't



# Vertex and Edge Labels

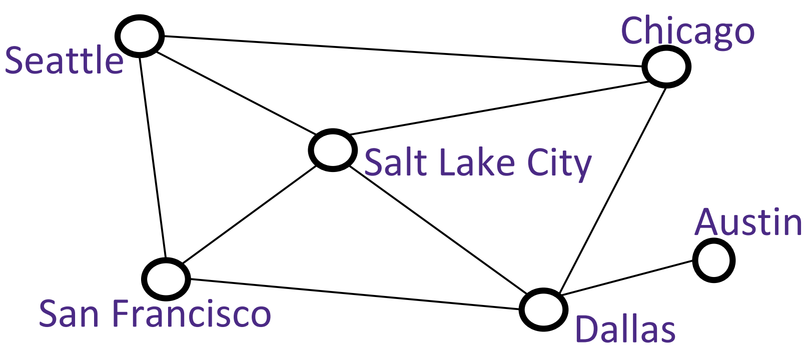
- More generally, both vertices and edges can have (possibly non-numeric) labels



# Paths and Cycles (1 of 2)

- A ***path*** is a list of vertices $[v_0, v_1, \ldots, v_n]$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$
  - You'd call it a path from $v_0$ to $v_n$

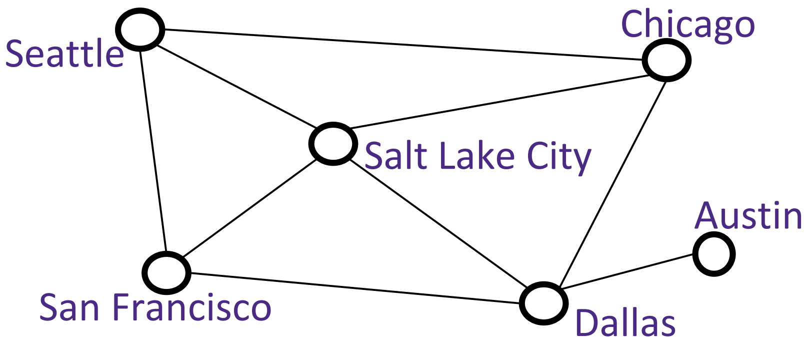
- A ***cycle*** is a path that begins and ends at the same node
  - i.e., $v_0 == v_n$

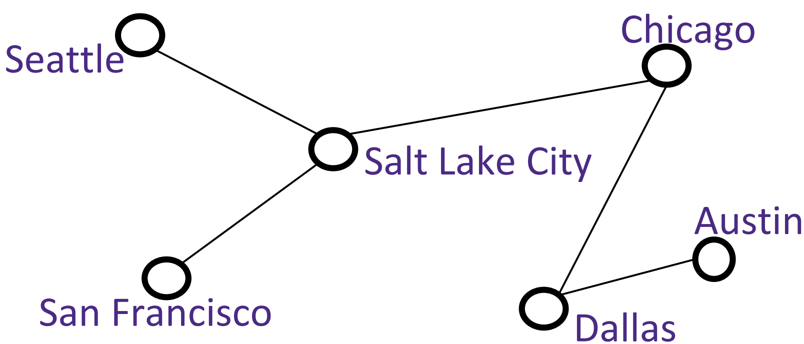



- Example path:
  - [Seattle, SLC, Chicago, Dallas, SF, Seattle]
  - Also happens to be a cycle!

# Paths and Cycles (2 of 2)

- A graph that does not contain any cycles is ***acyclic***

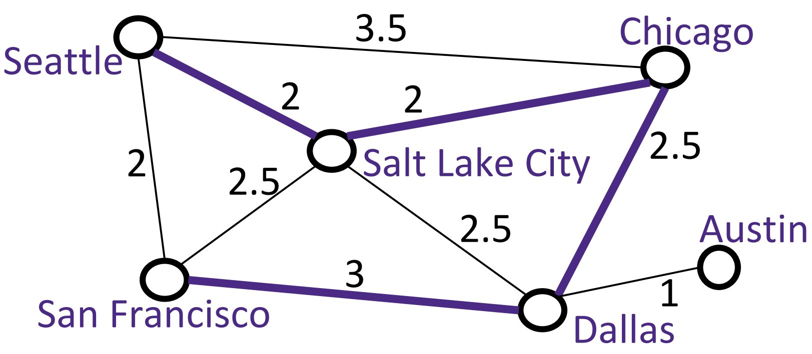


# Path Length and Cost

- ***Path length***: *Number* of edges in a path
  - Also called "unweighted cost"
- ***Path cost***: *Sum of the weights* of each edge in a path

- Example: P = [Seattle, SLC, Chicago, Dallas, SF]
  - length(P) = 4
  - cost(P) = 9.5



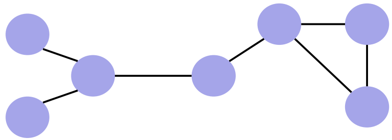
# gradescope

gradescope.com/courses/256241

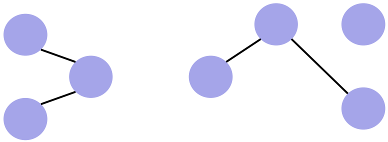
- Do ***weights*** make sense for each of the following graphs? What would they represent, and could those weights be ***negative***?
  - Web pages with links
  - Facebook friends
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Family trees
  - Course pre-requisites

# <u>Undirected</u> Graph Connectivity

- An undirected graph is ***connected*** if for all pairs of vertices **`u,v`**, there exists a *path* from **`u`** to **`v`**

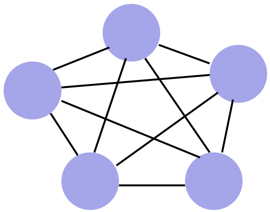


Connected graph



Disconnected graph

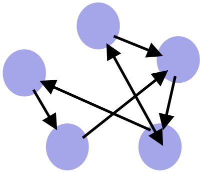
- An undirected graph is ***complete*** (aka ***fully connected***) if for all pairs of vertices **`u,v`**, there exists an *edge* from **`u`** to **`v`**



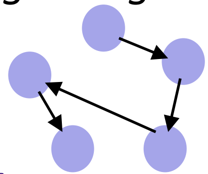
*(not pictured: self edges)*

# Directed Graph Connectivity

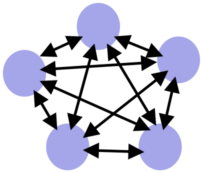
- A directed graph is ***strongly connected*** if for all pairs of vertices **`u,v`**, there exists a *path* from **`u`** to **`v`**



- A directed graph is ***weakly connected*** if for all pairs of vertices **`u,v`**, there exists a path from **`u`** to **`v`** *ignoring direction of edges*



- A directed graph is ***complete*** (aka ***fully connected***) if for all pairs of vertices **`u,v`**, there exists an *edge* from **`u`** to **`v`**

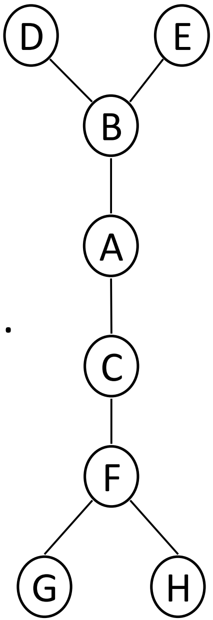


*(not pictured: self edges)*

# Trees as Graphs

- A ***tree*** is a graph that is:
  - acyclic
  - connected

Example:

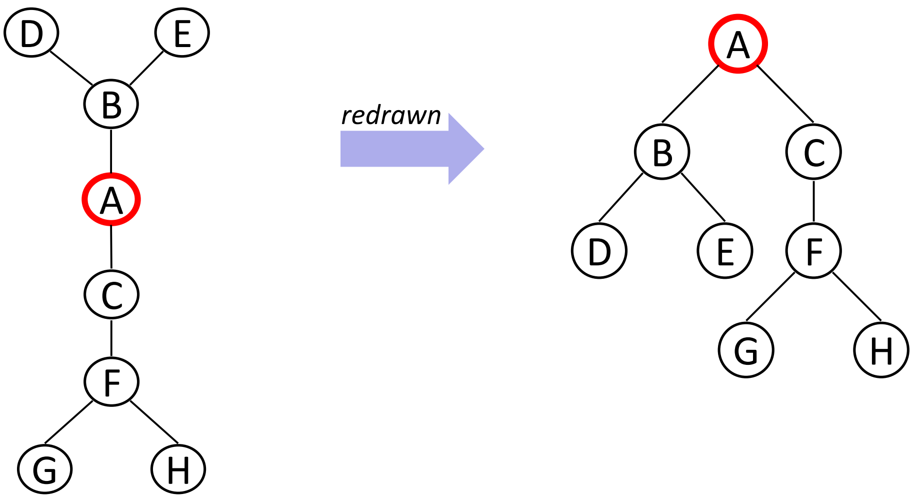



- So all trees are graphs, but not all graphs are trees
- How does this relate to the trees we know and love?...

# Rooted Trees (1 of 2)

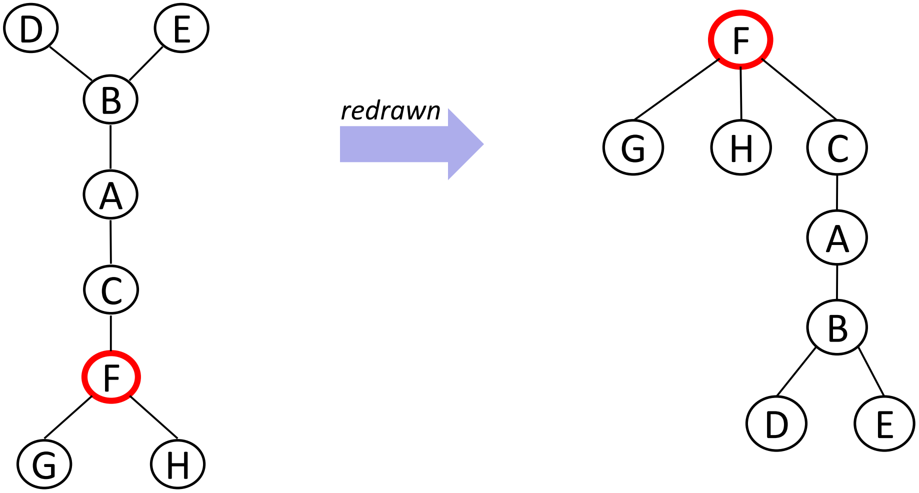
- We've previously worked with ***rooted trees***, where:
  - We identify a unique ("special") vertex: the root
  - We think of edges as **directed**: parent to children

- The same tree can be redrawn as multiple rooted trees depending on which node you pick as the root



# Rooted Trees (2 of 2)

- We've previously worked with ***rooted trees***, where:
  - We identify a unique ("special") vertex: the root
  - We think of edges as **directed**: parent to children
- The same tree can be redrawn as multiple rooted trees depending on which *node you pick as the root*



# gradescope

gradescope.com/courses/256241

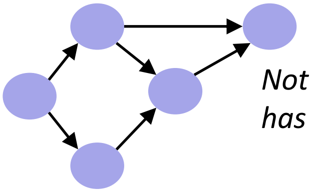
- For the undirected graphs, are they ***connected***?
- For the directed graphs, are they ***strongly connected***? ***weakly connected***?

- Examples:
  - Web pages with links
  - Facebook friends
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Family trees
  - Course pre-requisites

# Directed Acyclic Graphs (aka DAGs)

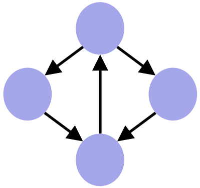
- A ***DAG*** is a directed graph with no directed cycles

- Every rooted directed tree is a DAG
  - But not every DAG is a rooted directed tree:



*Not a rooted directed tree; has an undirected cycle*

- Every DAG is a directed graph (by definition!)
  - But not every directed graph is a DAG:



*Not a DAG; has a directed cycle*

# Density / Sparsity (1 of 2)

- *Recall*:
  - In an undirected graph, $0 \le |E| < |V|^2$
  - In a directed graph: $0 \le |E| \le |V|^2$

So for any graph, $|E| \in O(|V|^2)$

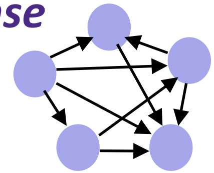
- One more fact:
  - In a *connected* undirected graph, $|E| \ge |V|-1$
  - In a *weakly connected* directed graph, $|E| \ge |V|-1$
  - In a *strongly connected* directed graph, $|E| \ge |V|$

So for any *connected* graph, $|E| \in \Omega(|V|)$

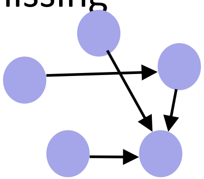
# Density / Sparsity (2 of 2)

- We do not always approximate as $|E|$ as $O(|V|^2)$
  - This is a *correct* bound, it's just oftentimes not *tight*

- If it is tight, i.e. $|E| \in \Theta(|V|^2)$, we say the graph is ***dense***
  - Intuitively: "lots of edges"



- If $|E| \in O(|V|)$ we say the graph is ***sparse***
  - Sparse: "most (of the possible) edges missing"



# gradescope

gradescope.com/courses/256241

- For the undirected graphs, are they ***dense*** or ***sparse***?
- For the directed graphs, are they a ***DAG***?

- Examples
  - Web pages with links
  - Facebook friends
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Family trees
  - Course pre-requisites

# Lecture Outline

- Graphs
  - Definitions
  - **Representation: Adjacency Matrix**
  - Representation: Adjacency List
  - Algorithms over Graphs
- Topological Sort

# Is a Graph an ADT or a Data Structure?

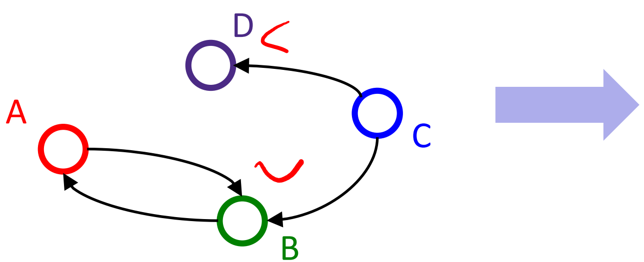
- tl;dr: 🤷‍♀️
  - They have operations like `hasEdge`$((v_j, v_k))$
  - But it is unclear what the "standard operations" are

- Instead, we develop algorithms over graphs and then use the "best" data structure for that algorithm. "Best" depends on:
  - Properties of the graph (e.g., dense versus sparse)
  - Common queries
    - e.g., "is **`(u,v)`** an edge?" vs "what are the neighbors of node **`u`**?"

- There are two standard graph representations:
  - ***Adjacency Matrix*** and ***Adjacency List***
  - Different trade-offs, particularly time vs space

# Adjacency Matrix: Representation

- Assign each node a number from `0` to `|V|-1`
- Graph is a `|V|x|V|` matrix (ie, 2-D array) of booleans
  - `M[u][v] == true` means there is an edge from `u` to `v`

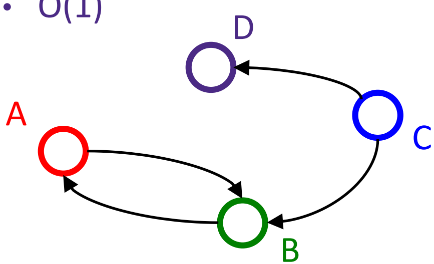



| | A | B | C | D |
|---|---|---|---|---|
| A | F | T | F | F |
| B | T | F | F | F |
| C | F | T | F | T |
| D | F | F | F | F |

# Adjacency Matrix: Properties (1 of 3)

- Running time to:
  - Get a vertex's out-edges:
    - O(|V|)
  - Get a vertex's in-edges:
    - O(|V|)
  - Decide if some edge exists:
    - O(1)
  - Insert an edge:
    - O(1)
  - Delete an edge:
    - O(1)

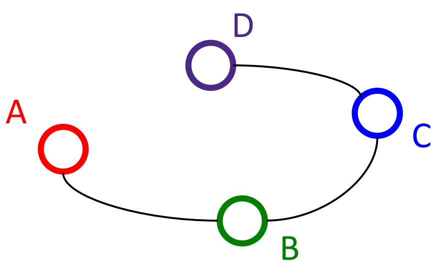
- Space requirements:
  - $|V|^2$ bits
- Best for sparse or dense graphs?
  - Best for dense graphs




| | A | B | C | D |
|---|---|---|---|---|
| A | F | T | F | F |
| B | T | F | F | F |
| C | F | T | F | T |
| D | F | F | F | F |

# Adjacency Matrix: Properties (2 of 3)

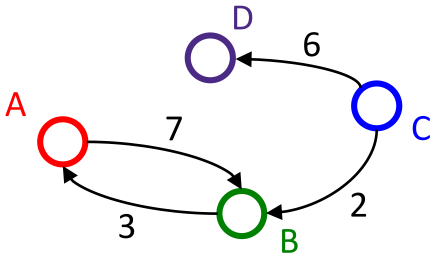
- How does the adjacency matrix vary for an undirected graph?
  - Undirected graphs are symmetric about diagonal axis
  - Languages with array-of-array matrix representations can save ½ the space by omitting the symmetric half
    - Languages with "proper" 2D matrix representations (eg, C/C++) can't do this




| | A | B | C | D |
|---|---|---|---|---|
| A | F | T | F | F |
| B | T | F | T | F |
| C | F | T | F | T |
| D | F | F | T | F |

# Adjacency Matrix: Properties (3 of 3)

- How can we adapt the representation for weighted graphs?
  - Store the weight in each cell
  - Need some value to represent “not an edge”
    - In some situations, 0 or -1 works



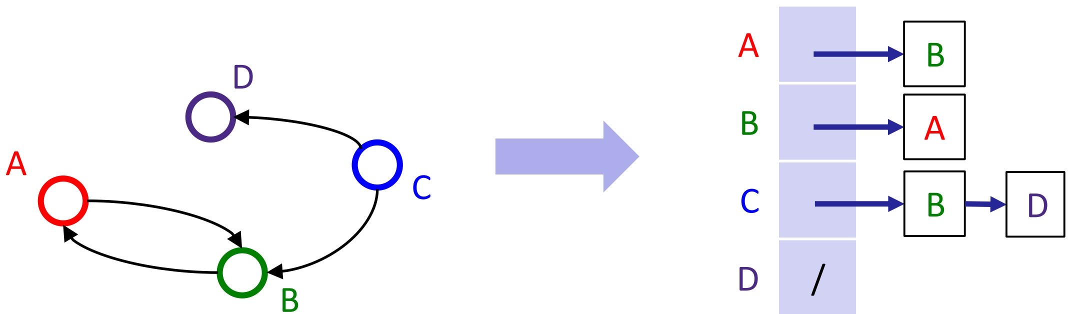

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | **7** | 0 | 0 |
| B | **3** | 0 | 0 | 0 |
| C | 0 | **2** | 0 | **6** |
| D | 0 | 0 | 0 | 0 |

# Lecture Outline

- Graphs
  - Definitions
  - Representation: Adjacency Matrix
  - **Representation: Adjacency List**
  - Algorithms over Graphs
- Topological Sort

# Adjacency List: Representation

- Assign each node a number from `0` to `|V|-1`
- Graph is an array of length `|V|`; each entry stores a list of all adjacent vertices
  - E.g. linked list

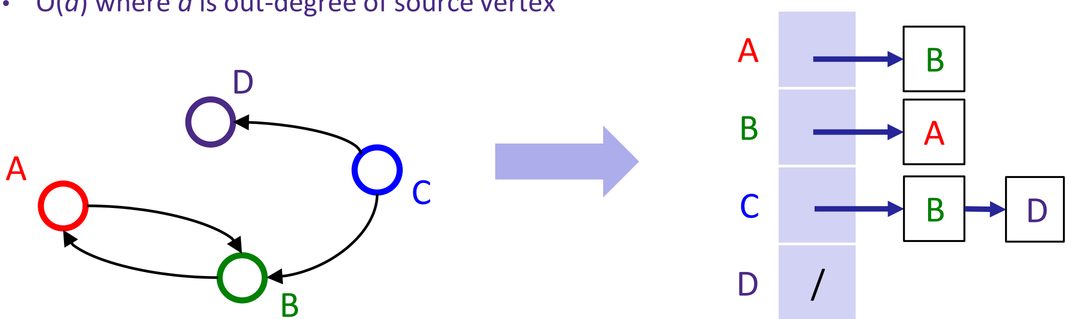


# Adjacency List: Properties (1 of 3)

- Running time to:
  - Get a vertex's out-edges:
    - O(*d*) where *d* is out-degree of vertex
  - Get a vertex's in-edges:
    - O(|V| + |E|)
    - (but could keep a second adjacency list for this!)
  - Decide if some edge exists:
    - O(*d*) where *d* is out-degree of source vertex
  - Insert an edge:
    - O(1)
    - (unless you need to check if it's there; then O(*d*))
  - Delete an edge:
    - O(*d*) where *d* is out-degree of source vertex

- Space requirements:
  - O(|V|+|E|)
- Best for sparse or dense graphs?
  - Best for sparse graphs, so usually just stick with linked lists for the buckets

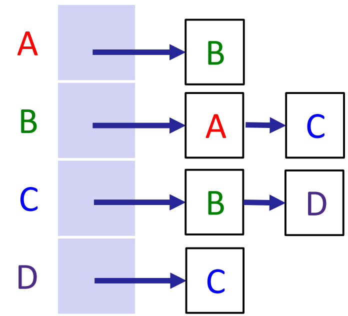
Let d(v) = out-degree of v



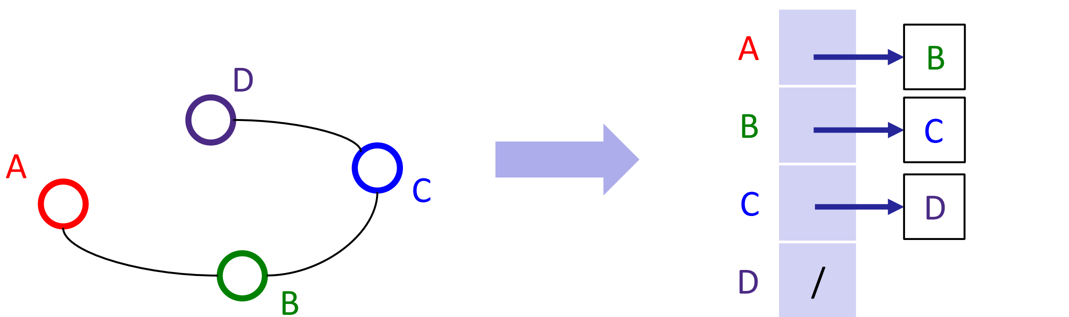
# Adjacency List: Properties (2 of 3)

- How does the adjacency list vary for an *undirected graph*?

- (Constant-time) improvements:
  - If vertices can be ordered, order (aka normalize) before insertion/lookup
    - Eg, only insert/find (A, B), *never* (B, A)
  - Double the edges
    - Eg, insert (A, B) *and also* (B, A)

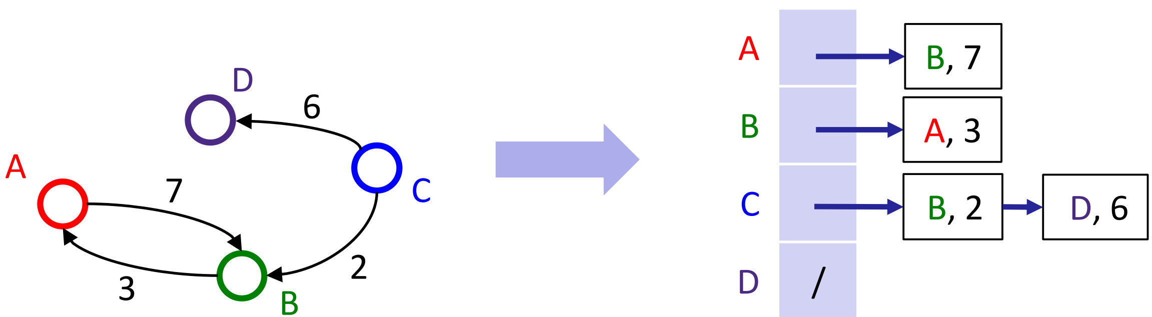



*… or …*



# Adjacency List: Properties (3 of 3)

- How can we adapt the representation for *weighted graphs*?
  - Store the weight alongside the destination vertex
  - No need for a special value to represent “not an edge”!



# Which Representation is Better?

- Graphs are often sparse:
  - Road networks are often grids
    - Every corner isn't connected to every other corner
  - Airlines rarely fly to all possible cities
    - Or if they do it is to/from a hub

- Adjacency lists should generally be your default choice
  - Slower performance compensated by greater space savings
  - Many graph algorithms rely heavily on getAllEdgesFrom(v)

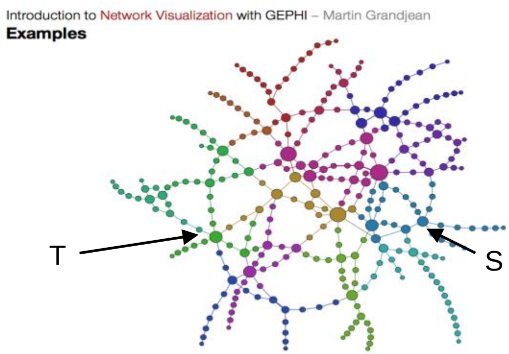
| | getAllEdgesFrom(v) | hasEdge(v, w) | getAllEdges() | Space |
|---|---|---|---|---|
| Adjacency Matrix | Θ(V) | Θ(1) | $\Theta(V^2)$ | $\Theta(V^2)$ |
| Adjacency List | Θ(d(v)) | Θ(d(v)) | Θ(E + V) | Θ(E + V) |

# Lecture Outline

- Graphs
  - Definitions
  - Representation: Adjacency Matrix
  - Representation: Adjacency List
  - **Algorithms over Graphs**
- Topological Sort

# Graph Queries

- Lots of interesting questions we can ask about a graph:
  - What is the shortest route from S to T? What is the longest route without cycles?
  - Are there cycles in this graph?
  - How can we disconnect this graph cheaply?
  - What is the cheapest way to connect this graph?



# Graph Queries More Theoretically

- Some well known graph problems and their common names:
  - **s-t Path**. Is there a path between vertices s and t?
  - **Connectivity.** Is the graph connected?
  - **Biconnectivity.** Is there a vertex whose removal disconnects the graph?
  - **Shortest s-t Path.** What is the shortest path between vertices s and t?
  - **Cycle Detection.** Does the graph contain any cycles?
  - **Planarity**. Can you draw the graph on paper with no crossing edges?
  - **Isomorphism**. Are two graphs the same graph (in disguise)?
  - **Euler Tour.** Is there a cycle that uses every *edge* exactly once?
  - **Hamilton Tour.** Is there a cycle that uses every *vertex* exactly once?

- Often can't tell how difficult a graph problem is without very deep consideration.

# Graph Problem Difficulty

- Some well known graph problems:
  - **Euler Tour:** Is there a cycle that uses every *edge* exactly once?
  - **Hamilton Tour:** Is there a cycle that uses every *vertex* exactly once?

- Difficulty can be deceiving
  - O(|E|) Euler tour algorithm was found as early as 1873 [Link]
  - Despite decades of intense study, no efficient algorithm for a Hamilton tour exists. Best algorithms are exponential time

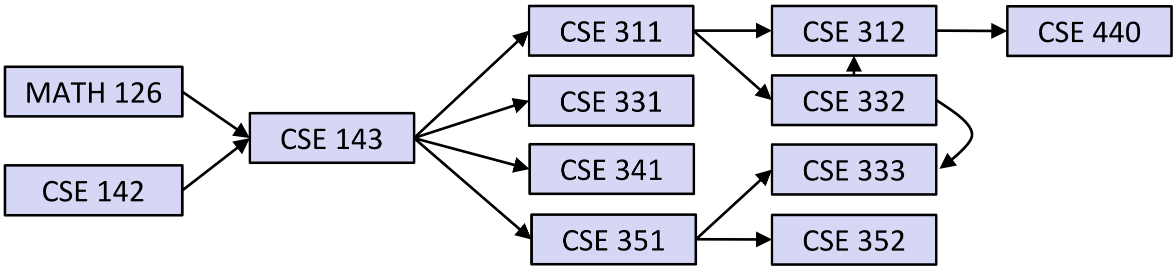
- Graph problems are among the most mathematically rich areas of CS theory

# Lecture Outline

- Graphs
  - Definitions
  - Representation: Adjacency Matrix
  - Representation: Adjacency List
  - Algorithms over Graphs
- **Topological Sort**

# Topological Sort: Applications

- Figuring out how to finish your degree



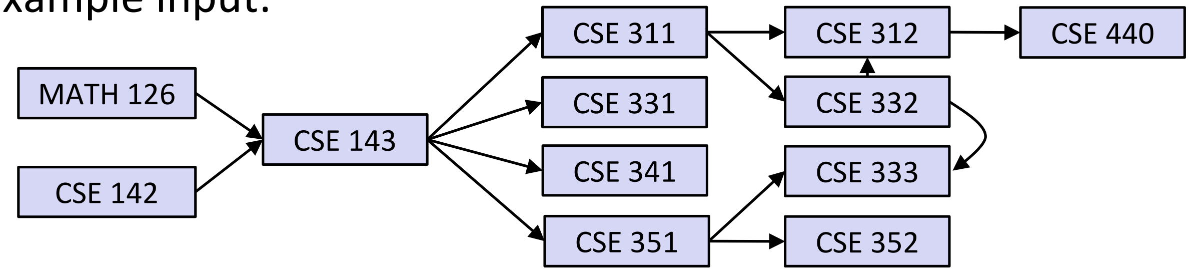

- Determining the order for recomputing spreadsheet cells
- Computing the order to compile files using a Makefile
- Scheduling jobs in a big data pipeline

# Topological Sort

Disclaimer: Do not use for official advising purposes!
Falsely implies CSE 332 is a prereq for CSE 312, etc.

- Output all the vertices of a DAG in an order such that no vertex appears before any other vertex that has a path to it
  - A DAG represents a ***partial order***, and a topological sort produces a ***total order*** that is consistent with it

- Example input:

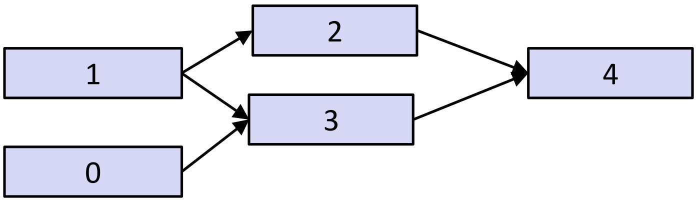



- Example output:
  - 126, 142, 143, 311, 331, 332, 312, 341, 351, 333, 352, 440

# gradescope

gradescope.com/courses/256241

- Provide two valid topological sorts for this digraph:




- Why do we perform topological sorts only on DAGs?
- Does a DAG always have a unique topological sort?
- What DAGs have exactly 1 topological sort?
- Provide a real-world application of topological sort
  - Eg, determining what order to watch Marvel movies in