Graphs

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

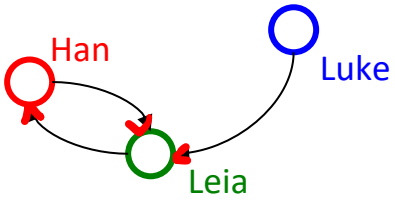
- Exercises 7, 8, and midterm grades releasing later today
  - Some complications on grading ex6, releasing later this weekend
  - Will make more official Ed announcement later
  - Reminder: regrade requests have been opening the next day noon and closing 2 days later at noon
- Reminder to reserve your slot for final (DUE SUNDAY)
- Deadline pushed by 24 hours: Ex11, 12
  - EC is different this quarter than previous.. Oops
  - Happy National root beer float day / International beer day..?

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***

    V = {v_1, v_2, …, v_n}
  - A set of ***edges***, possibly ***directed***

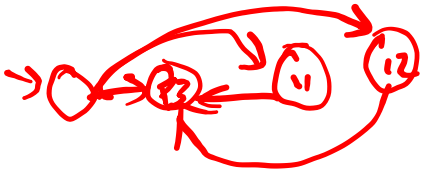
    E = {e_1, e_2, …, e_m}
    - Each edge **e_i** is a pair of vertices **(v_j,v_k)**
    - An edge "connects" the vertices




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

- We've seen these before: DAGs

Is a Graph an ADT?



- Kind of, but not in the way we're used to
  - They have operations like `hasEdge(`(v_j, v_k)`)`
  - But it is unclear what the "standard operations" are

- Instead:
  - We tend to develop algorithms over graphs and then use whatever data structure is efficient for that algorithms

- Many important problems can be solved by:
  1. Formulating them in terms of graphs
  2. Applying a standard graph algorithm

Some Graph Examples

- For each 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 domain… Almost as if… It's a core concept…?

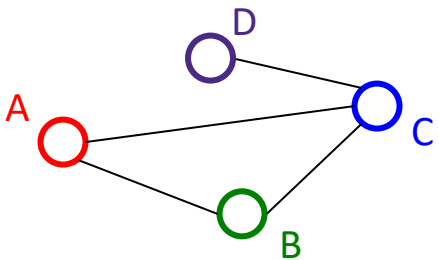
gradescope

gradescope.com/courses/275833

- What is another graph example you can think of?
  - Put it in chat after countdown & respond on gradescope

Undirected Graphs

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

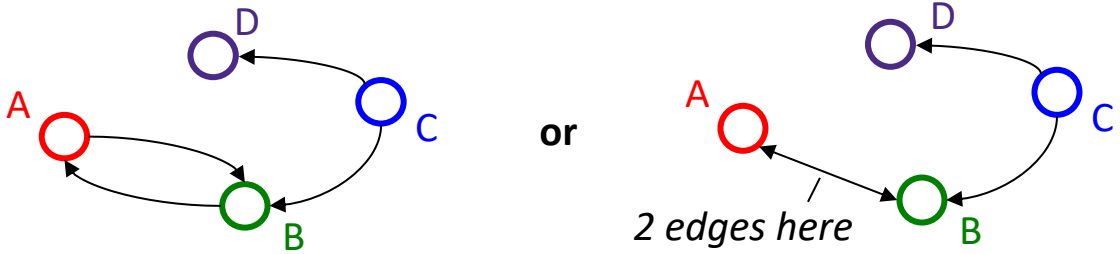


- Thus, `(u,v) ∈ E` implies `(v,u) ∈ E`
  - Only one of these edges needs to be in the set; the other is implicit

- ***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***



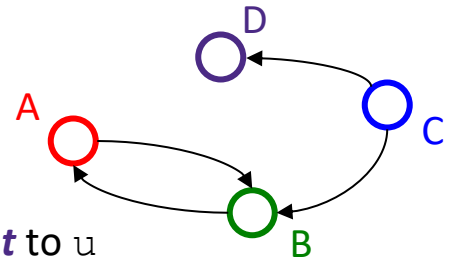
- Thus, `(u,v) ∈ E` does *not* imply `(v,u) ∈ E`
  - `(u,v) ∈ E` means `u → v`; u is the ***source*** and `v` the ***destination***

- ***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

- 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

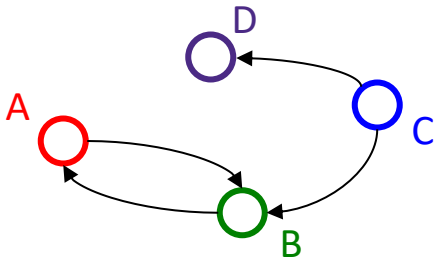
Adjacency (1 of 2)



- 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?
      - $|V||V-1|/2 \in O(|V|^2)$
    - Maximum for a directed graph with no self-edges?
      - $|V||V-1| \in O(|V|^2)$
    - If self-edges are allowed, add `|V|` to the answers above (applies to both undirected and directed graphs)

gradescope

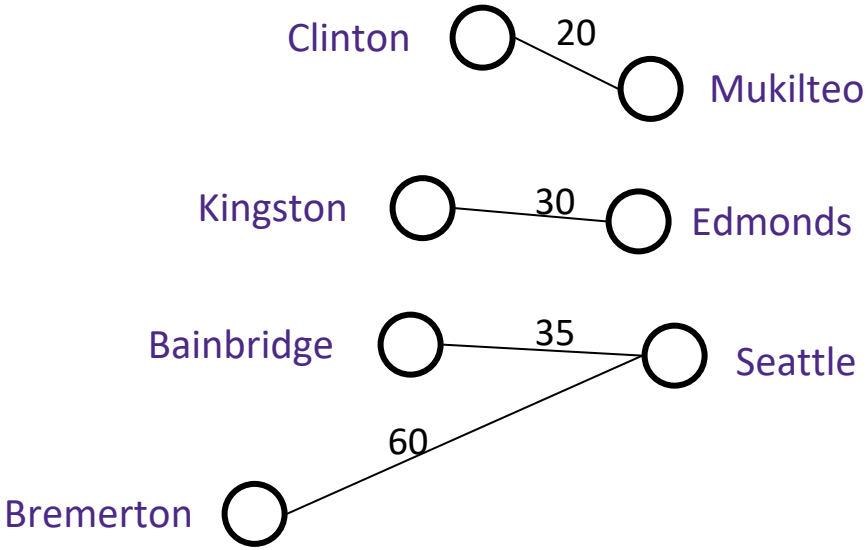
gradescope.com/courses/275833

Graph Examples, Again

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



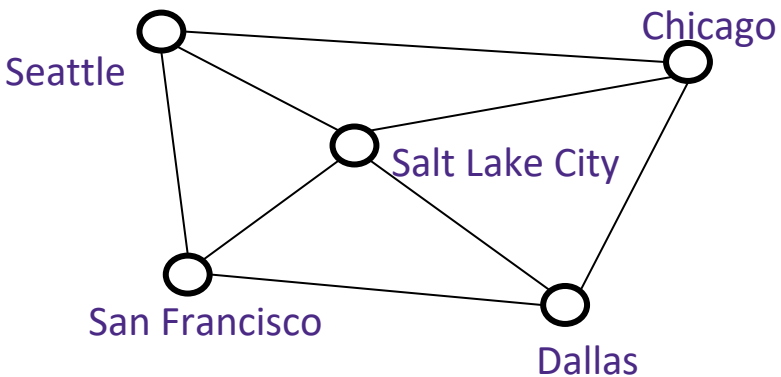
Graph Examples, Once More Unto the Breach

- 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

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

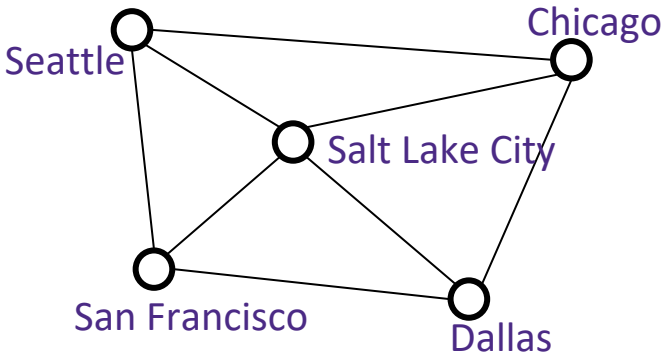
- A ***cycle*** is a path that begins and ends at the same node
  - i.e., $v_0 == v_n$

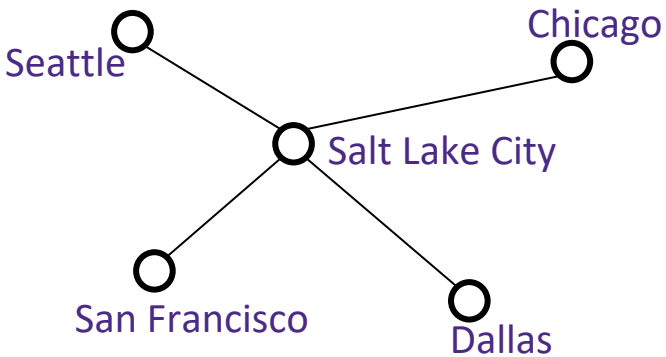


- Example path:
  - [Seattle, SLC, Chicago, Dallas, SF, Seattle]
  - Also happens to be a cycle!

Paths and Cycles (2 of 2)

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

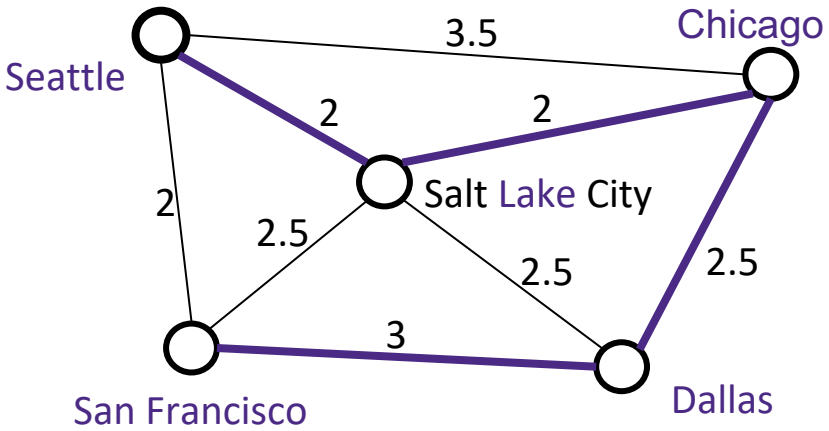




Path Length and Cost

- ***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]



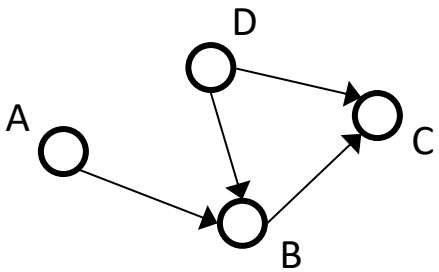
length(P) = 4
cost(P) = 9.5

gradescope

gradescope.com/courses/275833

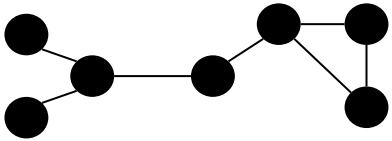
- Is there a path from A to D? Does the graph contain any cycles?

A. Yes / Yes
B. Yes / No
C. No / Yes
D. No / No
E. I'm not sure …

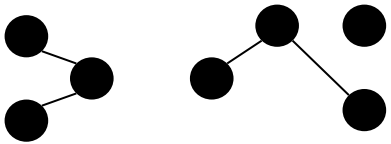


Connectivity: Undirected Graphs

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

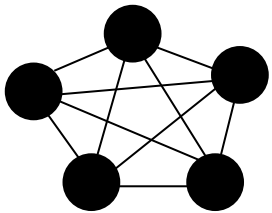


Connected graph



Disconnected graph

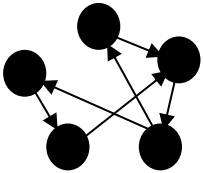
- An undirected graph is ***complete*** (aka ***fully connected***) if for all pairs of vertices `u`, `v`, there exists an *edge* from `u` to `v`



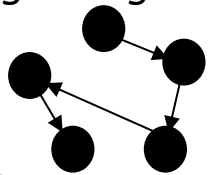
(not pictured: self edges)

Connectivity: Directed Graphs

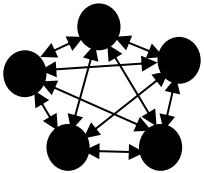
- A directed graph is ***strongly connected*** if for all pairs of vertices `u`, `v`, there exists a *path* from `u` to `v`



- 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)

Much Example. Very Graph. Wow.

- For undirected graphs: ***connected***?
- For directed graphs: ***strongly connected***? ***weakly connected***?
  - (A) Web pages with links
  - (B) Facebook friends
  - (C) Methods in a program that call each other
  - (D) Road maps (e.g., Google maps)
  - (E) Airline routes
  - (F) Family trees
  - (G) Course pre-requisites

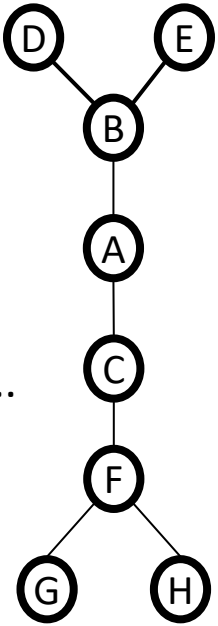
Trees as Graphs

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

- So all trees are graphs, but not all graphs are trees

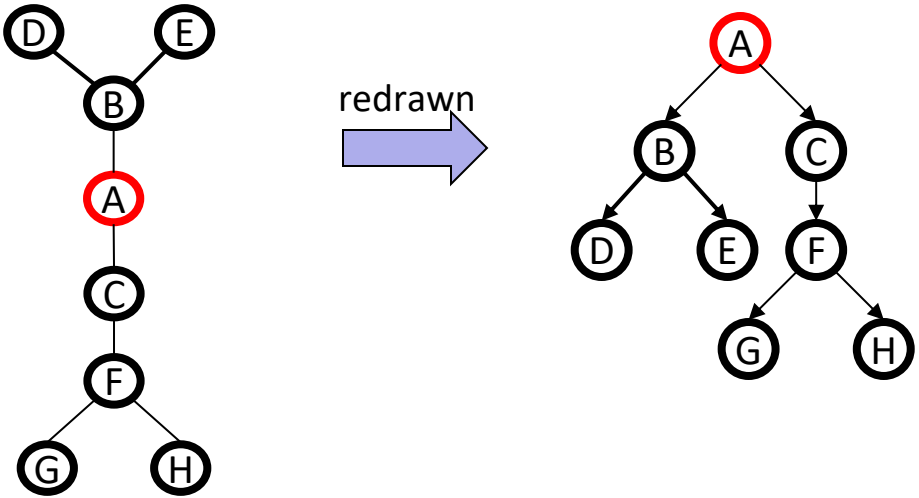
- How does this relate to the trees we know and love?...

Example:



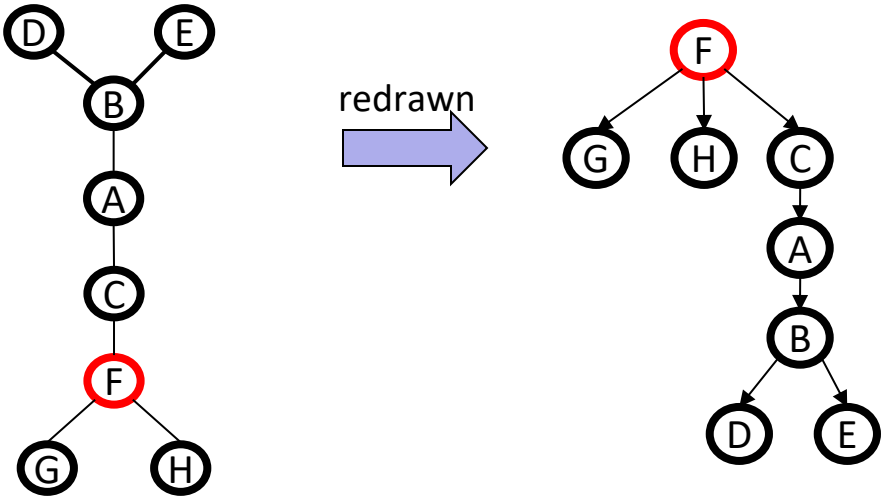
Rooted Trees (1 of 2)

- 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)

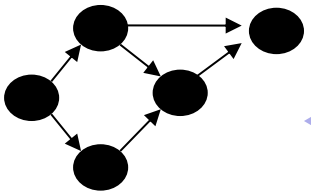
- We've previously worked with ***rooted trees***, where:
  - We identify a unique ("special") vertex: the root
  - We think of edges as **directed**: parent to children
- The same tree can be redrawn as multiple rooted trees depending on which *node you pick as the root*



Directed Acyclic Graphs (aka DAGs)

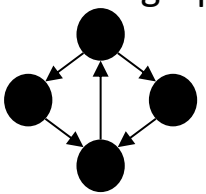
- A ***DAG*** is a directed graph with no directed cycles

- 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 (**D**AG: It's in the name)
  - But not every directed graph is a DAG:



Not a DAG; has a directed cycle

Graph Examples: One more time

- Which of our ***directed***-graph examples do you expect to be a ***DAG***?
  - 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

Density / Sparsity (1 of 2)

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

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

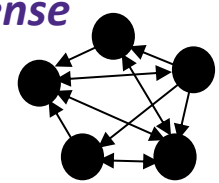
- One more fact:
  - In a *connected* undirected graph, $|E| \geq |V|-1$
  - In a *weakly connected* directed graph, $|E| \geq |V|-1$
  - In a *strongly connected* directed graph, $|E| \geq |V|$

So for any *connected* graph, $|E| \in \Omega(|V|^2)$

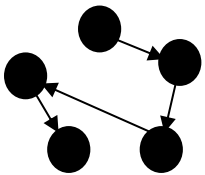
Density / Sparsity (2 of 2)

- 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"



What is the Data Structure?

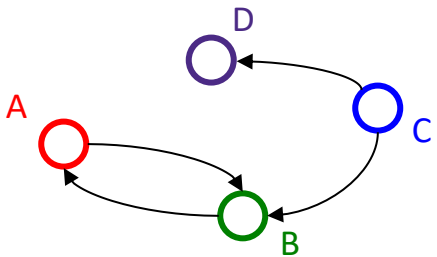
- Is a Graph an ADT? Maybe!
  - "Develop an algorithm over the graph, then use whatever data structure is efficient"

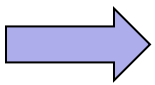
- The "best" data structure can depend on:
  - Properties of the graph (e.g., dense versus sparse)
  - Common queries
    - e.g., "is `(u,v)` an edge?" vs "what are the neighbors of node `u`?"

- 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`



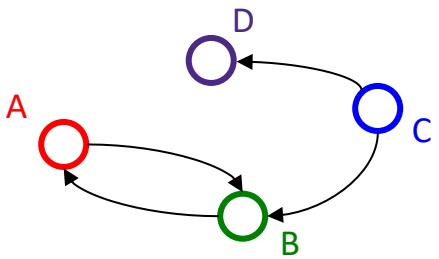


	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix: Properties (1 of 3)

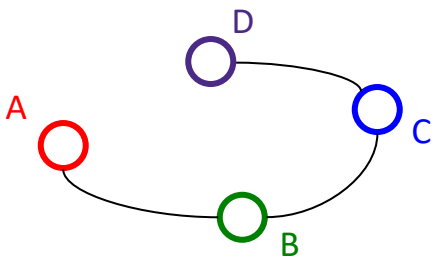
- Running time to:
  - Get a vertex's out-edges:
    - O(|V|)
  - Get a vertex's in-edges:
    - O(|V|)
  - Decide if some edge exists:
    - O(1)
  - Insert an edge:
    - O(1)
  - Delete an edge:
    - O(1)
- Space requirements:
  - $|V|^2$ bits
- Best for sparse or dense graphs?
  - Best for dense 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)

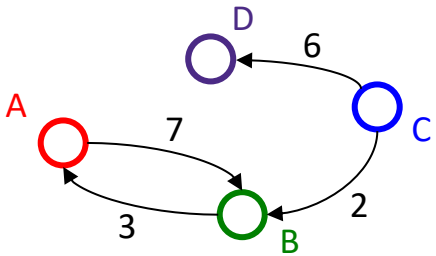
- How does the adjacency matrix vary for an *undirected graph*?
  - *Undirected graphs are symmetric about diagonal axis*
  - *Languages with array-of-array matrix representations can save ½ the space by omitting the symmetric half*
    - *Languages with "proper" 2D matrix representations (eg, C/C++) 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)

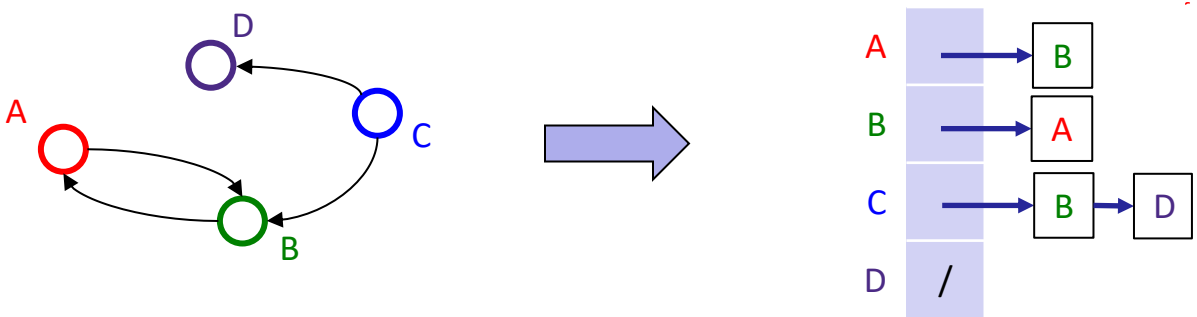
- How can we adapt the representation for *weighted graphs*?
  - *Store the weight in each cell*
  - *Need some value to represent "not an edge"*
    - *In some situations, 0 or -1 works*



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

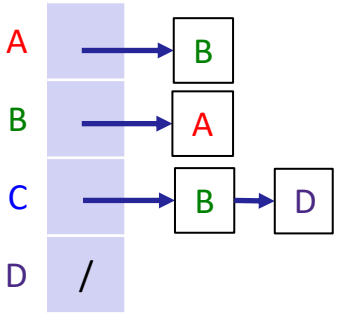
Adjacency List: Representation

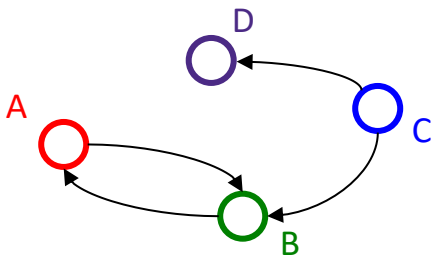
- Assign each node a number from `0` to `|V|-1`
- Graph is an array of length `|V|`; each entry stores a list of all adjacent vertices
  - E.g. linked list



Adjacency List: Properties (1 of 3)

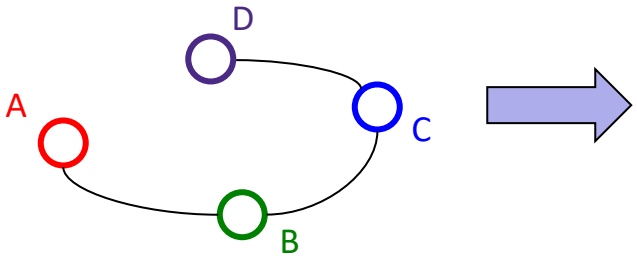
- Running time to:
  - Get a vertex's out-edges:
    - O(*d*) where *d* is out-degree of vertex
  - Get a vertex's in-edges:
    - O(|V| + |E|)
    - (but could keep a second adjacency list for this!)
  - Decide if some edge exists:
    - O(*d*) where *d* is out-degree of source vertex
  - Insert an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

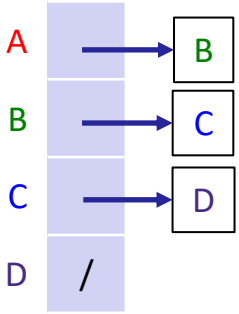




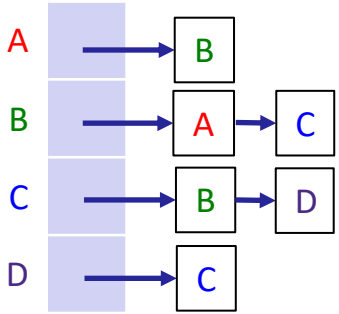
Adjacency List: Properties (2 of 3)

- How does the adjacency list vary for an *undirected graph*?
  - *Optionally, can double the entries to increase edge lookup speed*



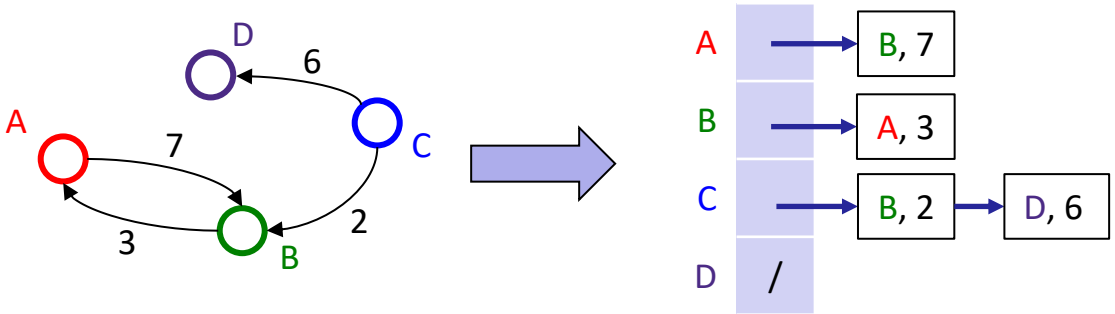


… or …



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"!*



Summary: Which is Better?

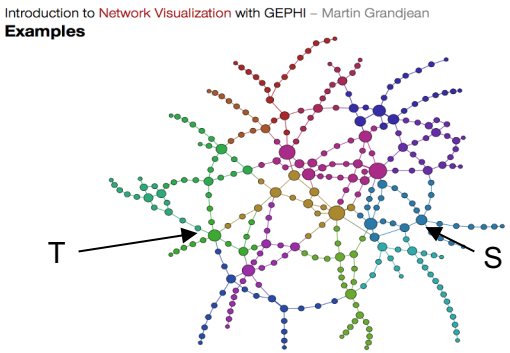
- Graphs are often sparse:
  - Road networks are often grids
    - Every corner isn't connected to every other corner
  - Airlines rarely fly to all possible cities
    - Or if they do it is to/from a hub

- 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()
Adjacency Matrix	$\Theta(V)$	$\Theta(1)$	$\Theta(V^2)$
Adjacency List	$\Theta(\text{degree}(v))$	$\Theta(\text{degree}(v))$	$\Theta(E + V)$

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?
  - Is there a cycle that uses each *vertex* exactly once?
  - Is there a cycle that uses each *edge* exactly once?



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?
  - **Euler Tour.** Is there a cycle that uses every edge exactly once?
  - **Hamilton Tour.** Is there a cycle that uses every vertex exactly once?
  - **Planarity**. Can you draw the graph on paper with no crossing edges?
  - **Isomorphism**. Are two graphs the same graph (in disguise)?

- 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
  - An efficient Euler tour algorithm O(# edges) 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.

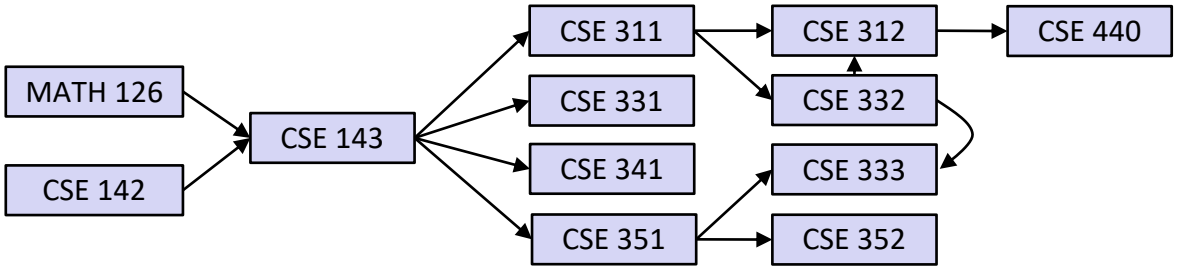
- Graph problems are among the most mathematically rich areas of CS theory

Topological Sort

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

- Given a DAG, output all the vertices in an order such that no vertex appears before any other vertex that has a path to it

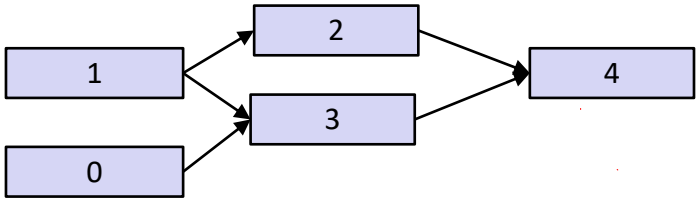
- Example input:



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

Activity: Valid Topological Sorts

- List 3 valid sorts:

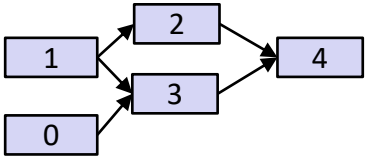


- Why do we perform topological sorts only on DAGs?
  - *A cycle means there is no correct answer*
- Does a DAG always have a unique answer?
  - *No; there can be 1 or more answers, depending on the graph*
- What DAGs have exactly 1 answer?
  - *A list*
- *Terminology*: A DAG represents a ***partial order***, and a topological sort produces a ***total order*** that is consistent with it

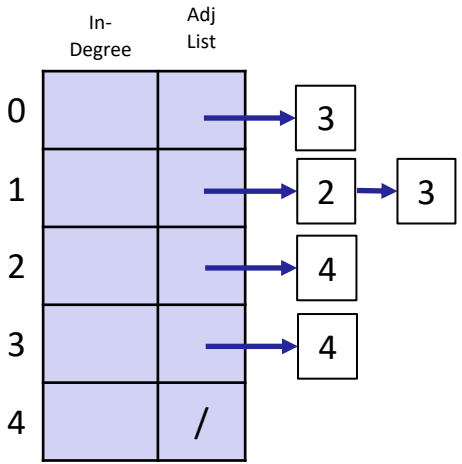
Topological Sort: Applications

- 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
- *Often: finding an order of execution for a dependency graph*

TopoSort: A Naïve Algorithm



1. Label ("mark") each vertex with its in-degree
   - Could write directly into a vertex's field or a parallel data structure (e.g., array)
2. While there are vertices not yet output:
   - Choose a vertex v with labeled with in-degree of 0
   - Output v and conceptually remove it from the graph
   - Foreach vertex w adjacent to v:
     - Decrement the in-degree of w

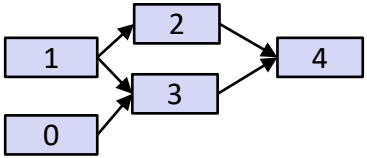


TopoSort: Notes

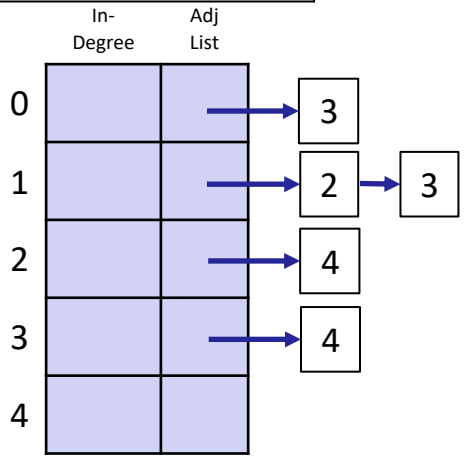
- Needed a vertex with in-degree of 0 to start
  - Remember: graph must be acyclic!

- If >1 vertex with in-degree=0, can break ties arbitrarily
  - Potentially many different correct orders!

Naïve TopoSort: Running Time?




```
labelEachVertexWithItsInDegree();
for (i=0; i < numVertices; i++){
  v = findNewVertexOfDegreeZero();
  put v next in output
  for each w adjacent to v
    w.indegree--;
}
```

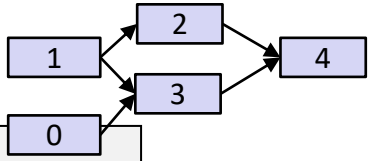


TopoSort's Runtime: Doing Better

- Avoid searching for a zero-degree node every time!
  - Keep the "pending" 0-degree nodes in a list, stack, queue, table, etc
  - The irder we process them affects output, but not correctness or efficiency (*as long as add/remove are both O(1)*)

- Using a queue:
  - Label each vertex with its in-degree, enqueuing 0-degree nodes
  - While "pending" queue is not empty:
    - v = dequeue()
    - Output v and remove it from the graph
    - For each vertex w adjacent to v (i.e. w such that (v,w) in E):
      - decrement the in-degree of w
      - if new degree is 0, enqueue it

Better TopoSort: Running Time?




```
labelAllAndEnqueueZeros();
for (i=0; i < numVertices; i++){
    v = dequeue();
    put v next in output
    for each w adjacent to v
      w.indegree--;
      if (w.indegree == 0)
        enqueue(w);
}
```