Graphs

Data Structures and Algorithms

Warmup

Discuss with your neighbors:

Come up with as many kinds of relational data as you can (data that can be represented with a graph).

Scientific paper
V: paper
e: refs.

maps:
V: dest.
e: roads

familial relationships
v: people
e: is parent of

food web:
V: animals
e: (u,v) mean u eats v

Announcements

Project 2 Due on Friday

Next individual HW assignment assigned tonight, due next Wednesday

- Two problems – merge sort and graph practice

Sign up for Project 3 partners by Thursday night!

I am gone tonight through Tuesday. Robbie will be lecturing again. No instructor office hours tomorrow or Tuesday next week.

Graph: Formal Definition

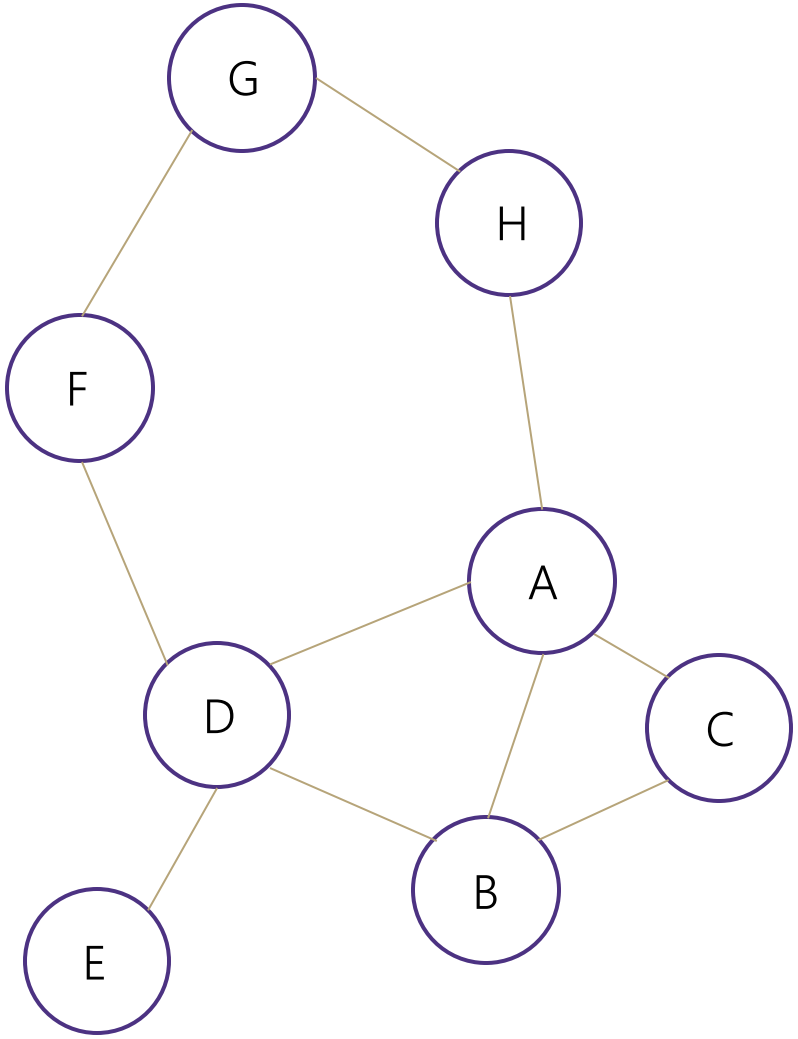
A **graph** is defined by a pair of sets G = (V, E) where...

- V is a set of **vertices**
  - A vertex or "node" is a data entity

  V = { A, B, C, D, E, F, G, H }

- E is a set of **edges**
  - An edge is a connection between two vertices

  E = { (A, B), (A, C), (A, D), (A, H),
  (C, B), (B, D), (D, E), (D, F),
  (F, G), (G, H)}



Graph Vocabulary

Graph Direction

- **Undirected graph** – edges have no direction and are two-way

  V = { A, B, C }

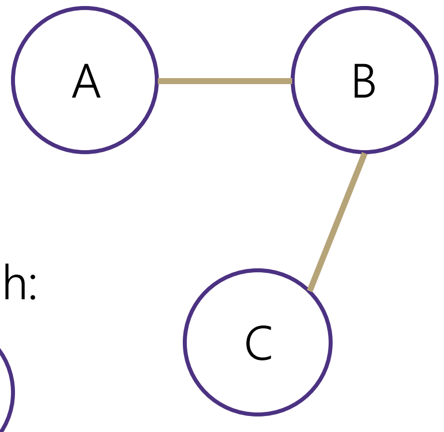
  E = { (A, B), (B, C) } *inferred (B, A) and (C,B)*

- **Directed graphs** – edges have direction and are thus one-way

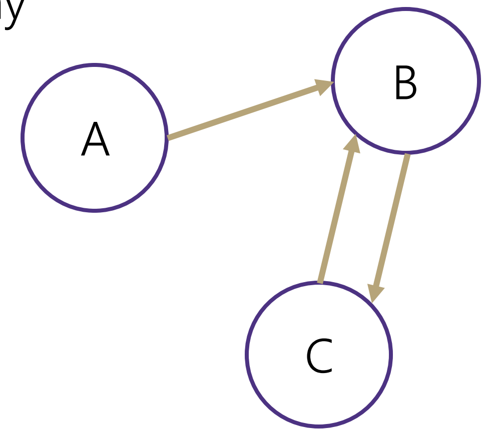
  V = { A, B, C }

  E = { (A, B), (B, C), (C, B) }

Undirected Graph:



Undirected Graph:



Degree of a Vertex

- **Degree** – the number of edges containing that vertex

  A : 1, B : 1, C : 1

- **In-degree** – the number of directed edges that point to a vertex

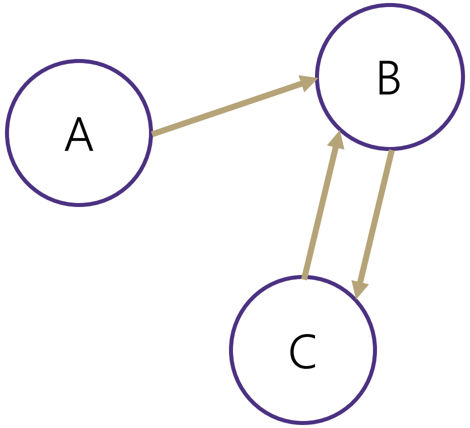
  A : 0, B : 2, C : 1

- **Out-degree** – the number of directed edges that start at a vertex

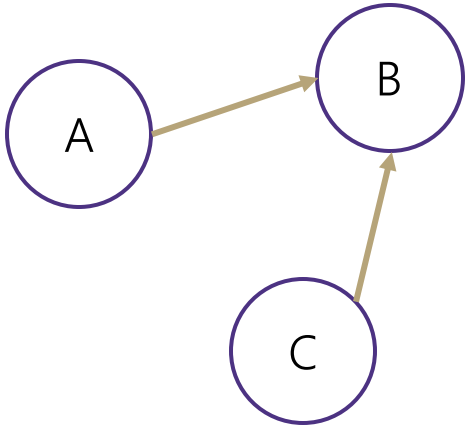
  A : 1, B : 1, C : 1

Food for thought

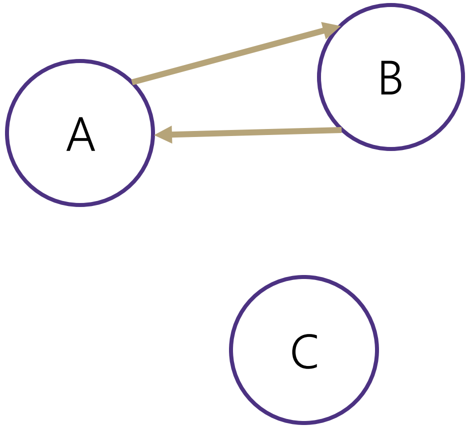
Is a graph valid if there exists a vertex with a degree of 0? Yes



A has an "in degree" of 0

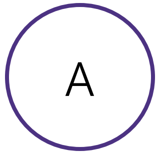


B has an "out degree" of 0



C has both an "in degree" and an "out degree" of 0

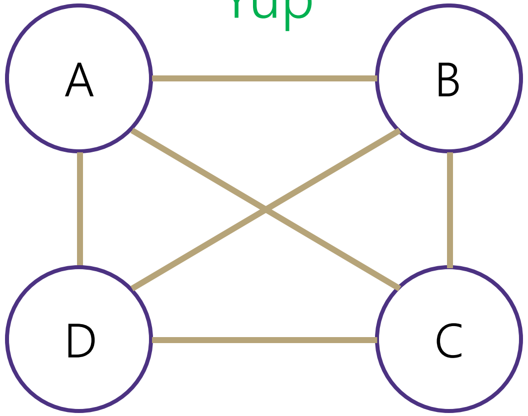
Is this a valid graph?

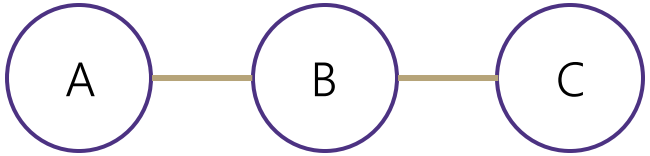


Yes!

Are these valid?

Yup

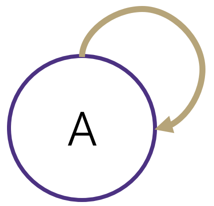




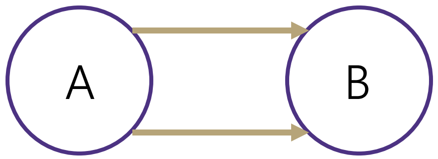
Sure

Graph Vocabulary

Self loop – an edge that starts and ends at the same vertex



Parallel edges – two edges with the same start and end vertices



Simple graph – a graph with no self-loops and no parallel edges

Complete graph – a graph with edges between every pair of vertices

For simple, undirected graphs…

What is the fewest number of edges a graph with n vertices can have?

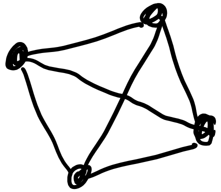
0

What is the sum of the degrees of all vertices in a graph? (in terms of |V| and |E|)

2|E|

What is the maximum number of edges a graph with n vertices can have?

n(n-1)/2 = $O(n^2)$ - (the complete graph on n vertices K_n)



← K_4 has $\frac{4 \cdot 3}{2} = 6$ edges

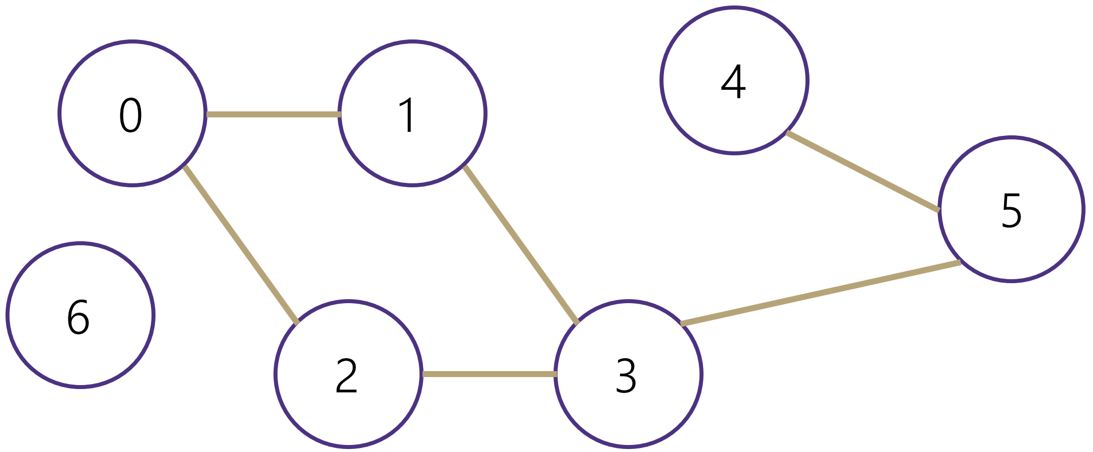
Representing Graphs

simple

Discuss with your neighbor: How would you implement a non-weighted, undirected graph?

Assume there are **n** vertices, and those vertices are numbered **0, 1, 2, ..., n-1**.

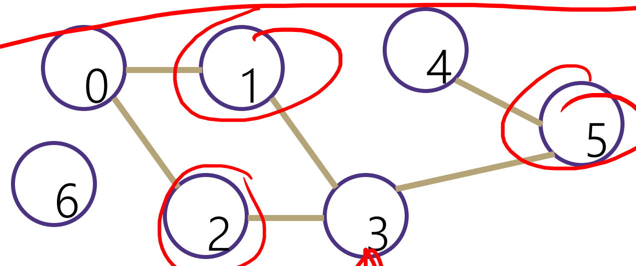
As an example:



G = (V, E)

V = {0,1,2,3,4,5,6}, E = { (0,1), (0,2), (1,3), (3,5), (4,5) }

Representing Graphs



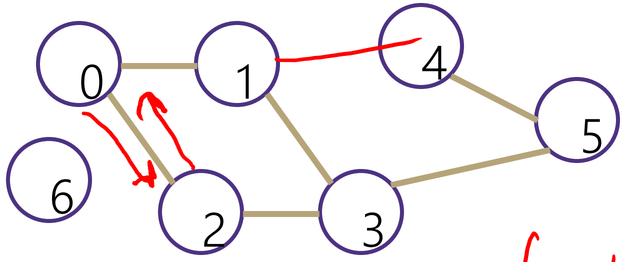
Map<vertex, set<vertex>>

class vertex:
DLL<vertices>

+ Map, array of vertizes

adjacency list

Adjacency Matrix



(u,v) from u to v
u → v

A **matrix** is a table of numbers, a[u][v].

In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

Can represent both undirected and directed graphs

Can represent self-loops and parallel edges (interpret as the # of edges between two vertices

Time Complexity (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(1)
- Check edge exists from (u,v): O(1)
- Get neighbors of u (out): O(n)
- Get neighbors of u (in): O(n)

Space Complexity: $O(n^2)$

v →

u ↓

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	0	1	~~0~~ 1	0	0
2	1	0	0	1	0	0	0
3	0	1	1	0	0	1	0
4	0	~~0~~ 1	0	0	0	1	0
5	0	0	0	1	1	0	0
6	0	0	0	0	0	0	0

Sparsity

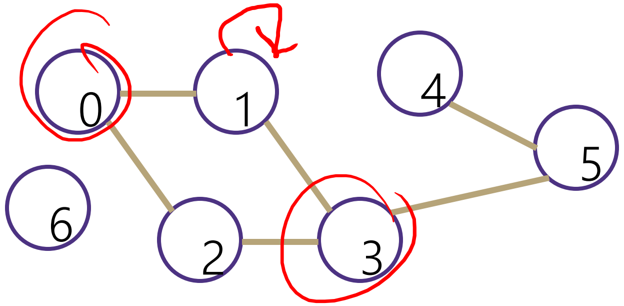
A **Dense Graph** is a graph with many edges $|E| \approx |V|^2$

A **Sparse Graph** is a graph with few edges $|E| \approx |V|$

Adjacency Matrices are very wasteful of space for spare graphs – they are almost all 0s!

How could we save space?

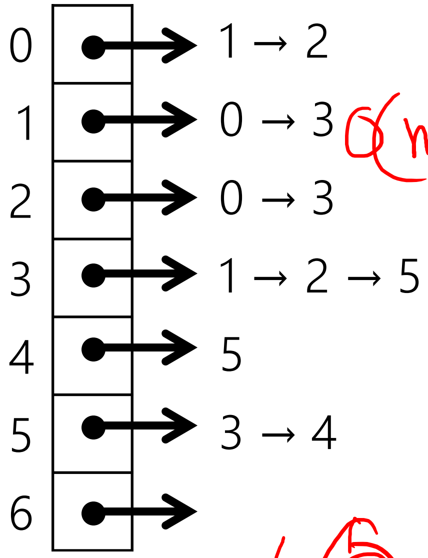
Adjacency List



An array where the u'th element contains a list of neighbors of u.

Can represent both undirected and directed graphs
In the directed case, put the out neighbors (a[u] has v for all (u,v) in E)

Can represent self-loops and parallel edges (repeat neighbor)



O(m+n)

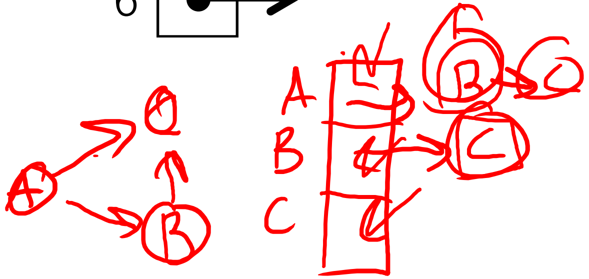
A, C

Time Complexity (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(min(n, m))
- Check edge exists from (u,v): O(min(n, m))
- Get neighbors of u (out): O(n) O(min(n, m))
- Get neighbors of u (in): O(n + m)

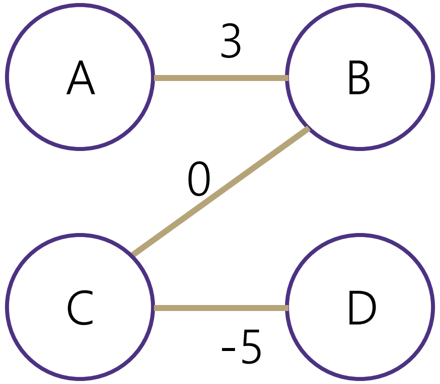
Space Complexity: O(n + m) ←

m possible edges
still need to check all
n array positions



Graph Vocabulary

Weighted Graph – a graph with numeric weights associated with each edge



- can be directed or undirected
  - weights can be negative, positive or 0
  - common to specify "only non-negative edge weights", or "only positive edge weights"
  - denoted e = (u, v, w) or sometimes e = ((u,v), w)
- Weights often carry meaning such as "distance", (e.g. driving time between cities)

Representing Weighted Graphs

Adjacency Matrix – You can make the value of at each element the **weight** of the edge

- In an int array (int[]) you can't distinguish between 0 weight edge and no edge
    - Solution 1 – You know ahead of time that 0 weight (or negative weight) edges do not exist and use that to represent no edge
    - Solution 2 (Java) – Use an Integer array (Integer[]) – have null represent no edge

Adjacency list – Store pairs (neighbor, edge weight)

Storing Data In Graphs

We often have data associated with vertices and edges

Vertex Data Examples:

- Facebook: Name, Age, Birthday, Hometown, Likes
- Google Maps: City name, elevation, hours of operation
- Internet: Page title, page contents, date last modified

Edge Data Examples:

- Facebook: Date friendship was made, friend vs. acquaintance, etc.
- Google Maps: Length of road, speed limit

Storing Data in Graphs

We could have a Graph<V, E> where V is a data type for Vertices and E is one for Edges

Adjacency Matrix: E[][] – now each entry has a pointer to edge data, or null if that edge is not in the graph

Adjacency List: E[] – the neighbor lists are now a list of edges

Both: Maintain a list V[] of vertices

Alternative Adjacency List: V[] just a list of vertices, where each vertex contains within it a list of (outgoing) edges.

Arrays → HashTables

When we analyze and describe graph algorithms, for simplicity we assume that each vertex has a unique identity 0, 1, 2, … n-1.

Accesses in Adjacency Matrices and getting an adjacency list are both **worst case** O(1) for arrays.

In reality, we often don't have unique sequential integers for each vertex. In a real graph implementation, we often use **HashMaps** or **HashSets**.

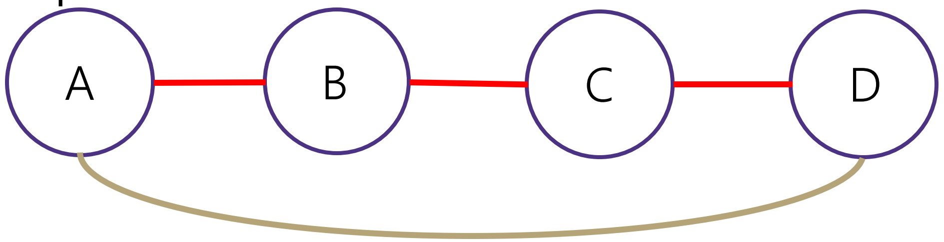
This would get us **average case** O(1), but **worst case** O(n). This is bad in analysis, but fine in practice. An adjacency list that stores references to the actual vertex objects can avoid repeated table lookups.

HashTable

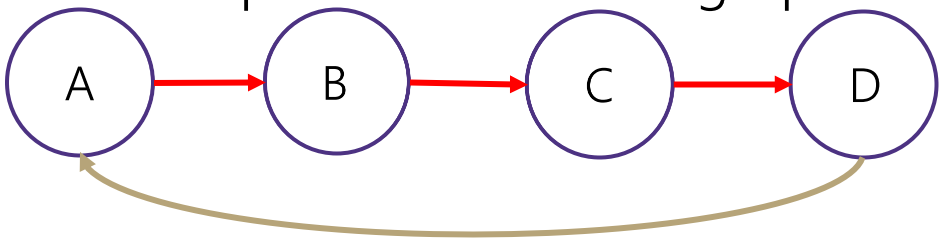
We *could* always use a hash table to give unique integer IDs to every element, incurring an O(n) **worst case**, but O(1) **average case** overhead before each call (which can save our worst case analysis for any O(n) or slower algorithm), but we don't usually bother to.

Graph Vocabulary

Path – A sequence of connected vertices



(Directed) Path – A path in a directed graph must follow the direction of the edges



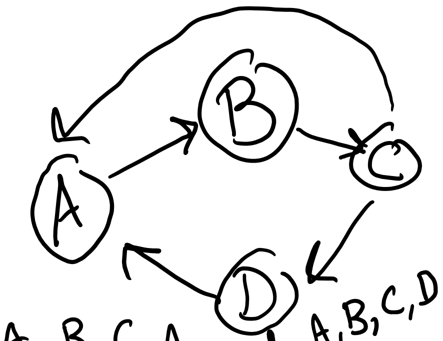
Path Length (unweighted) – The number of edges in a path

- (A,B,C,D) has length 3.

weighted $\Rightarrow$ Σ edgeweights

Simple Path – A path that doesn't repeat vertices (except maybe first=last)

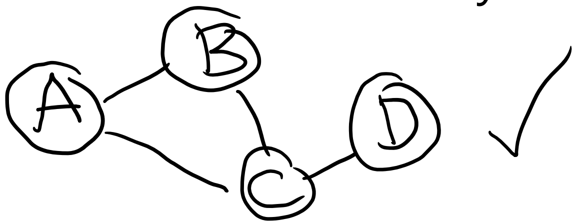
Cycle – A path that starts and ends at the same vertex (of length at least 1)

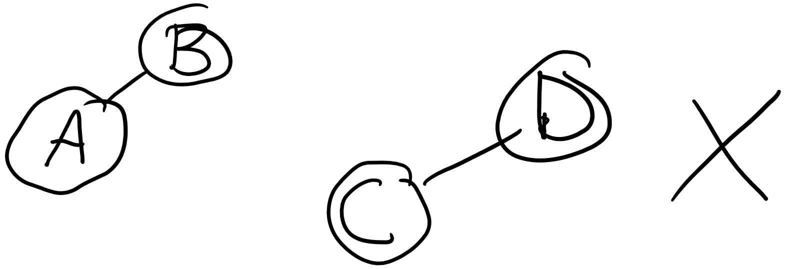


A, B, C, A and A, B, C, D, A

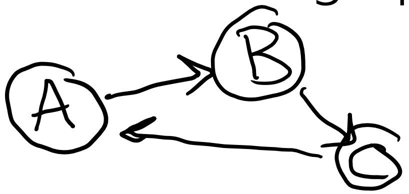
Graph Vocabulary

Connected Graph – A graph that has a path from every vertex to every other vertex (i.e. every vertex is **reachable** from every other vertex.

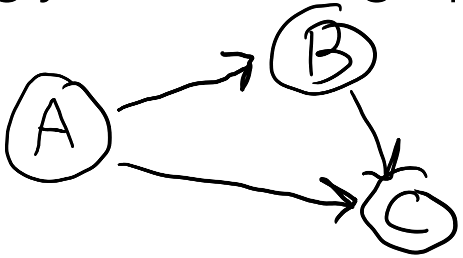




Strongly Connected Graph – A directed graph that is connected (note the direction of the edges!)



Weakly Connected Graph – A directed graph that is connected when interpreted as undirected. (Note all strongly connected graphs are also weakly connected)



Paths and Reachability

Very common questions:

- Is there a path between two vertices? (Can I drive from Seattle to LA?)
- What is the length of the shortest path between two vertices? (How long will it take?)

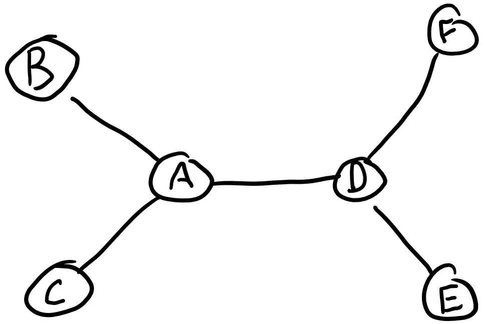
Less common, but still important:

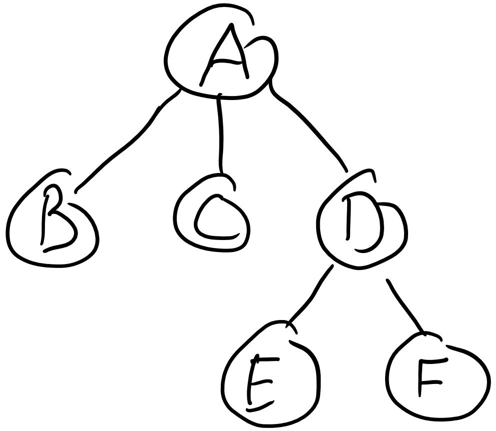
What is the longest path in a graph?

- In general a “hard”™ problem
- 7 degrees of Kevin Bacon
- Length of this path is called the “diameter” of a graph

Trees

A **tree** is a connected, acyclic graph.





An a tree there exist **exactly one** path between every pair of vertices.

A graph consisting of several disconnected trees is called a **forest**.

The trees we have seen so far have been **rooted trees** – interpret one vertex as the **root**, and its neighbors are now **children**, and the root of their own **subtrees.**

How many edges does a tree with **n** vertices have?

n - 1

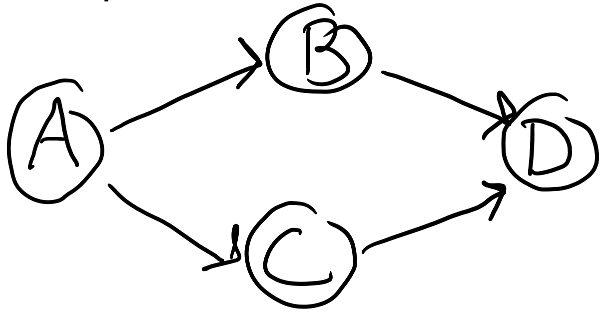
DAGs

DAG stands for **Directed, Acyclic, Graph**

This is the directed graph analog of a forest.

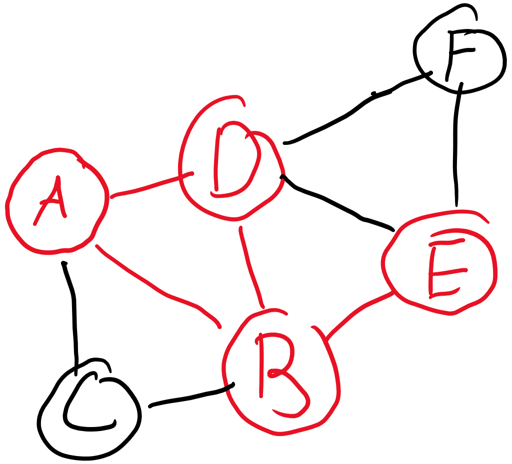
The trees we have made so far in this class have been implemented as weakly connected DAGs.

Can be used to represent **dependencies**: i.e. A must be completed before either B or C, and both B and C must be completed before D. Scheduling these tasks is called **topological sort.**



Subgraphs

Take a graph, and delete vertices and edges so you still have a graph.



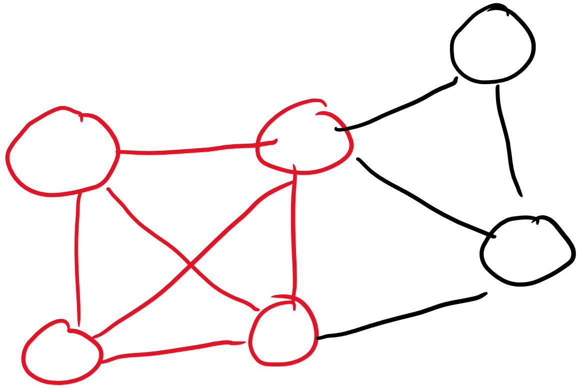
The remaining red vertices and edges form a subgraph.

Formally: $G' = (V', E') \subseteq G = (V, E) \Leftrightarrow V' \subseteq V$ and $E' \subseteq E$ *and* G' *is a graph*

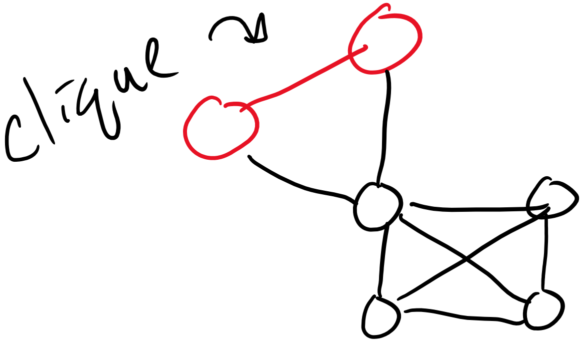
Interesting Subgraphs

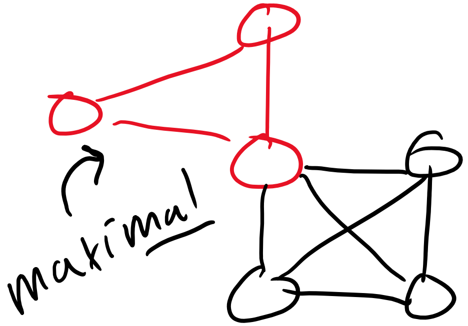
Cliques

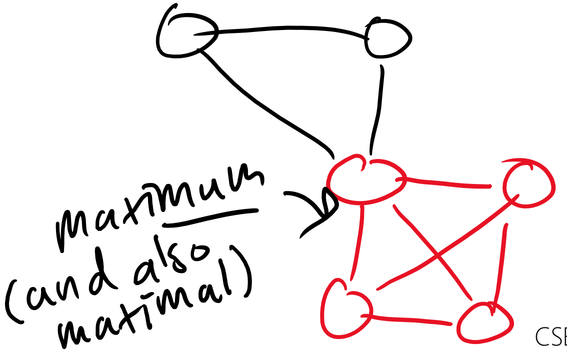
- A **clique** is a complete subgraph



- A **maximal** clique is a clique that you could not add any more vertices to and still have a clique. This is distinct from the **maximum** clique, which is the largest clique in a graph.



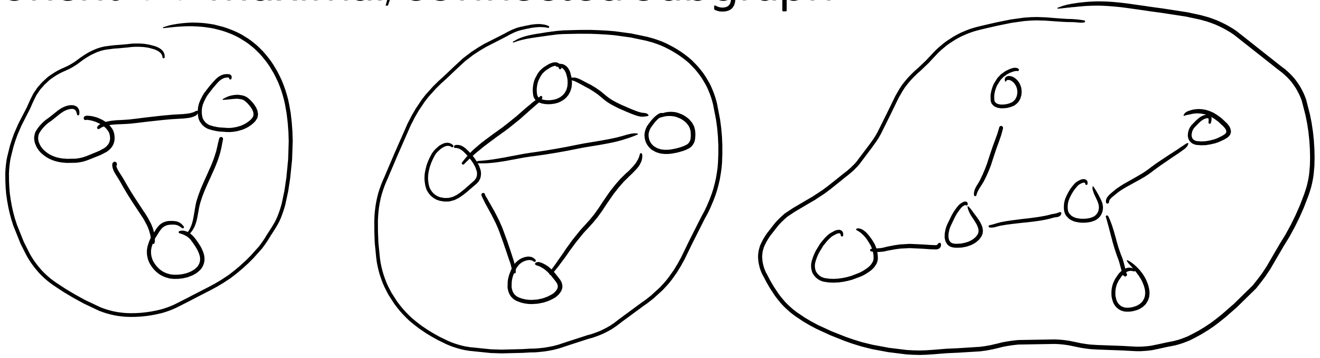




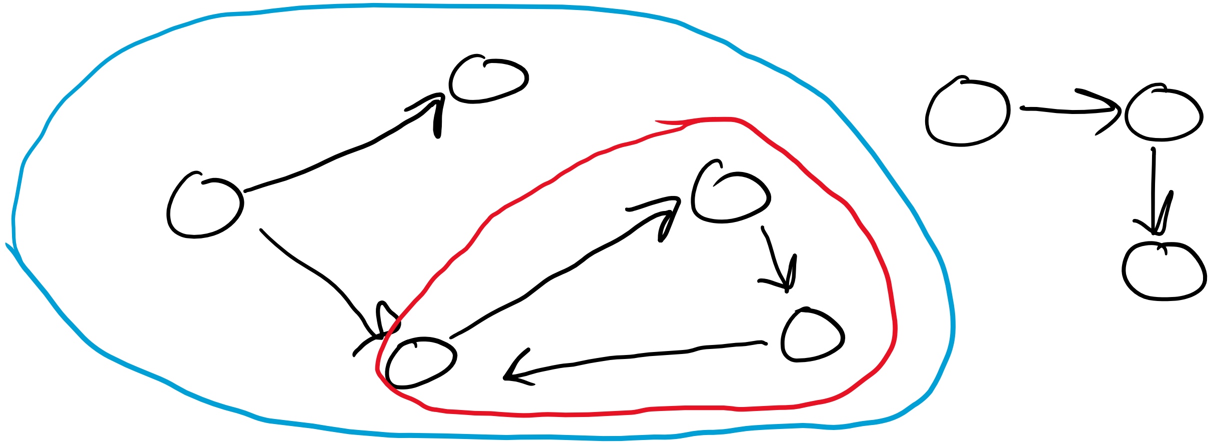
Interesting Subgraphs

Connected Components

- A **connected component** is a **maximal**, **connected subgraph**

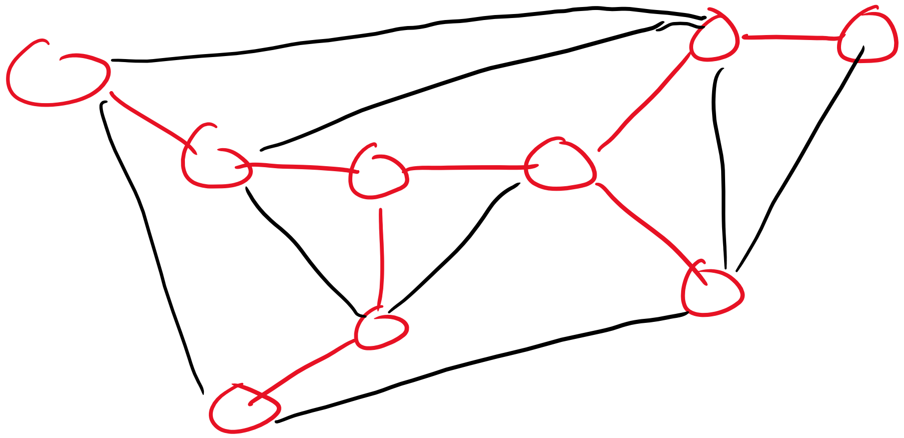


- In **directed** graphs, you have two kinds: **strongly connected**, and **weakly connected**:



Interesting Subgraphs

A **Spanning Tree** is a subgraph that is both a **tree** and includes **every vertex** (it **spans** the graph).



Every **connected** graph has at least one spanning tree. They are like skeletons of the graph.

An important problem is finding the **minimum spanning tree**. We will learn 2 algorithms for this. It is useful for optimization tasks (e.g. minimum cost to build a road network).

Other interesting Graph Problems

- Circuits – paths or cycles that touch every vertex

- Reductions – Everything we've seen so far in this class can be represented as a graph – a lot of other problems can too! Graphs can solve many problems.