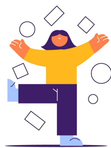
Lec 31: Graphs

May 8, 2026

Admin



- Lab 10 released after class
- Practice final as soon as I can (hopefully today)
- Grading catchup soon
- TA hours end next Friday (a week from today)
- I cannot have office hours on Monday of reading period, but I believe I can on Tuesday as usual (and Friday during finals period).
  - I’ll keep all of this posted/announced on the calendar.
- Any questions?

Hashtable Performance

Hashtable Performance

- Given the hash code of an object o, how long does `get(o)` take?
- O(run length) for linear probing; O(chain length) for external chaining
- How long does calculating a hash code take?
  - Can be long for, say, a long string.
  - $O(1)$ in terms of the number of items in the hash table
  - Another example of being careful about how we're stating our running time. Usually: in terms of number of strings in the table. But do we care about the length of our strings?

Impact on Performance

- Let's say we have constant load factor
- Assume we have a good hash function
  - Spreads objects out "like random"
  - Probability of an element hashing to a given slot is $1/m$
- Then an *average* element is in a bucket with constant chain length
- An *average* element is in a run of constant length
- (With overwhelming probability, never gets worse than $O(\log n)$ for any bucket/chain)
- Usually we say we have $O(1)$ performance. True on average; the actual worst case might be a bit worse

Summary of Map Performance

	`put`	`get`	`space`
Unsorted ArrayList	$O(n)$	$O(n)$	$O(n)$
Unsorted LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log n)$	$O(n)$
Balanced BST (AVL Tree)	$O(\log n)$	$O(\log n)$	$O(n)$
Hashtable (average)	$O(1)$	$O(1)$	$O(n)$

Graphs

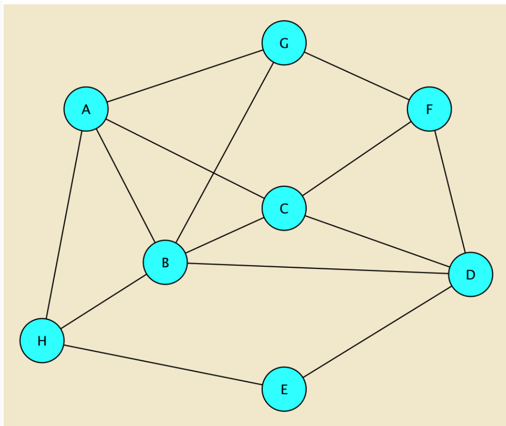
Purpose of Graphs

- Data structures up until this point have mostly been designed to *store data* for easy access
- Much of them were essentially implementations of a `Map` (or Dictionary)
- Trees were, in some cases, able to store relationships between pieces of data
  - Family tree: parent/child relationships
  - Lexicon trie: relationship between nodes in the trie represents stored words
- Graphs: a new data structure to store relationships between data. (Graphs are not particularly useful for Dictionary-like operations.)

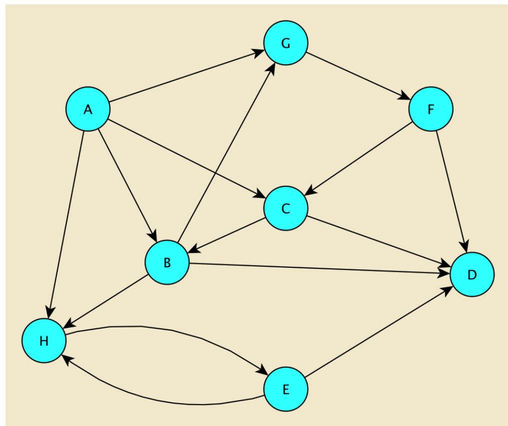
Graphs

- Graphs consist of *nodes* and *edges*
- Much like a tree! But no restrictions on how edges may connect nodes
- An edge may be *directed* or *undirected*
  - Directed edges represent a relationship from (say) node A to node B. Undirected represent a relationship between node A and node B (no direction on the relationship)

Drawing Graphs

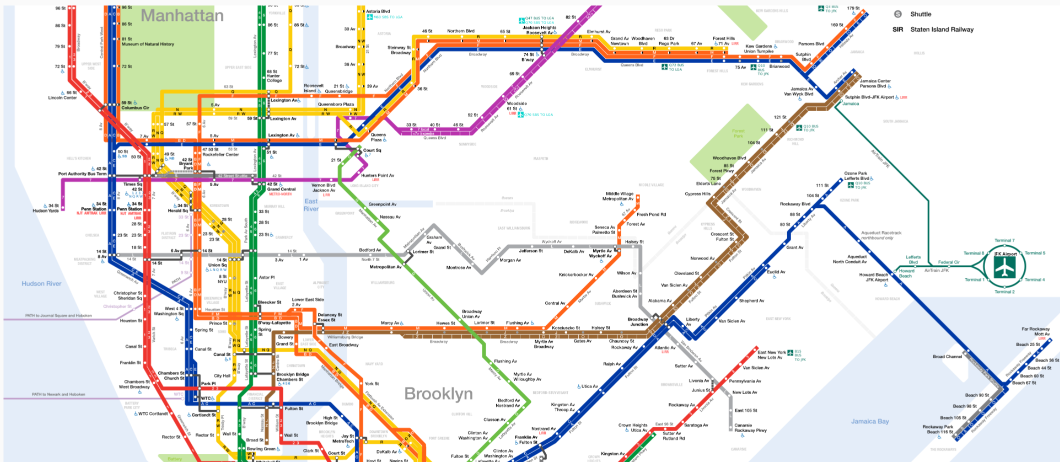


An undirected graph



A directed graph

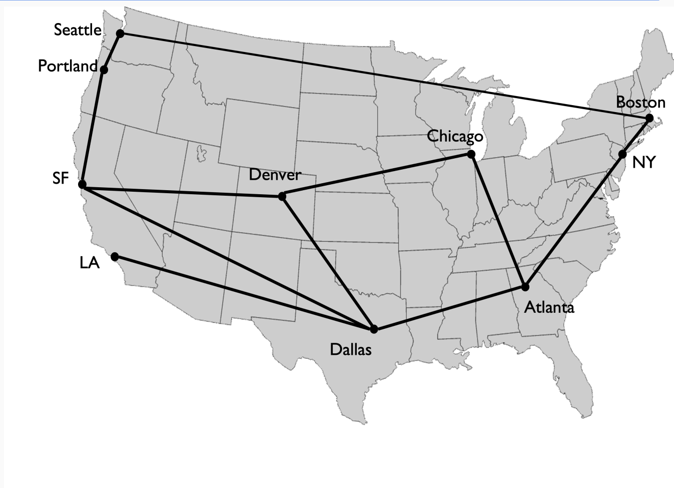
We usually draw graphs much like we drew trees. In directed graphs, we show the direction of an edge with an arrow.



What are the nodes here? Edges?

Nodes: subway stops. An edge between two stops if there is a train between them.

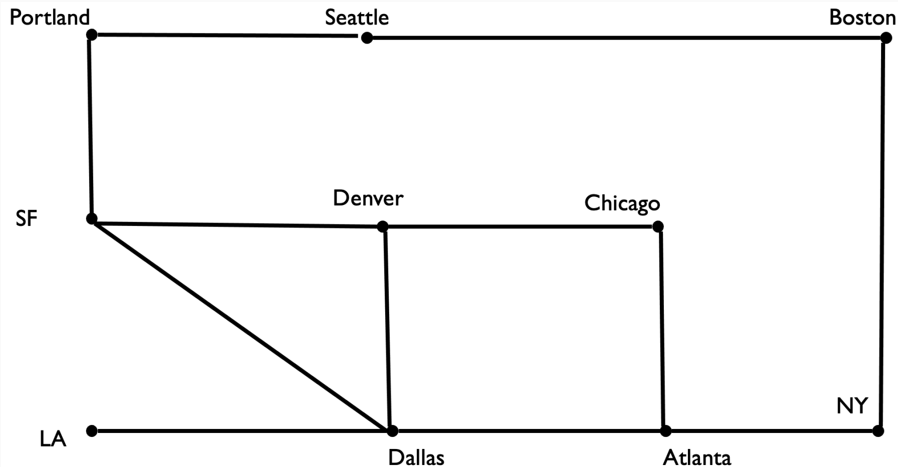
(Simplified) US Train Map



What are the nodes here? Edges?

Nodes: cities. An edge between two cities if there is a train between them. Note that it's not important how we draw the edges.

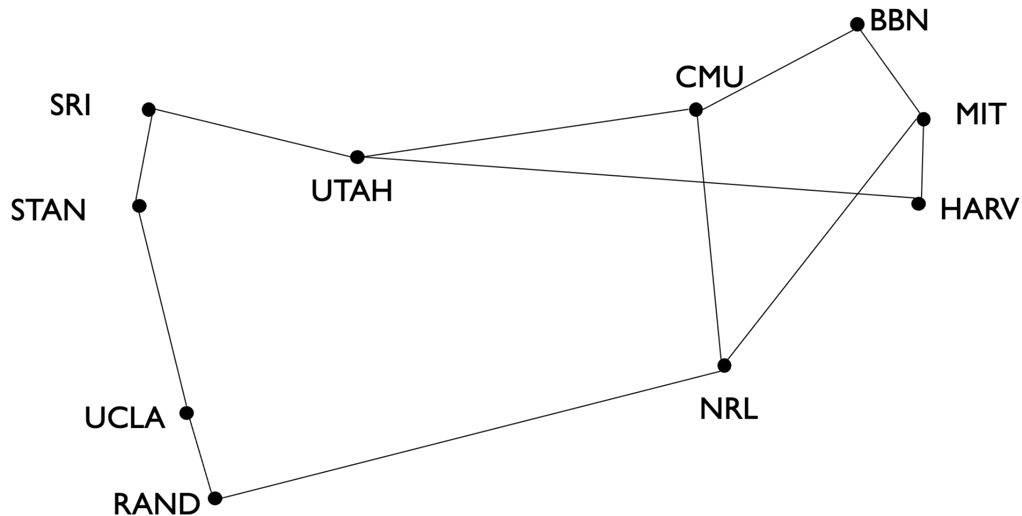
(Simplified) US Train Map



What are the nodes here? Edges?

Nodes: cities. An edge between two cities if there is a train between them. Note that it's not important how we draw the edges.

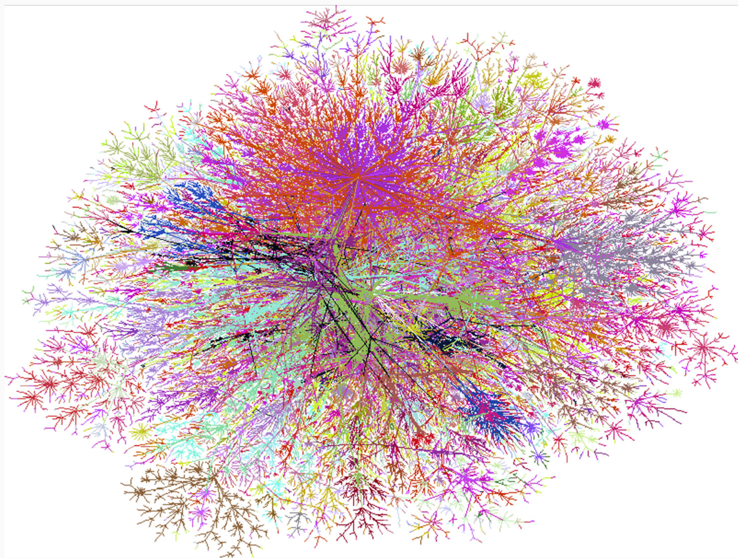
Internet Circa 1972



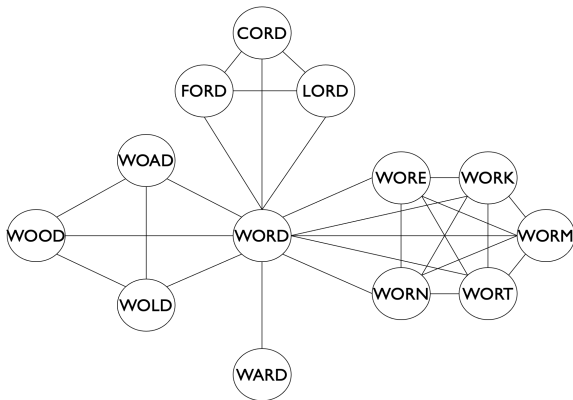
What are the nodes here? Edges?

Nodes: network access points. Edges represent a connection.

Internet Circa 1998



Word Game

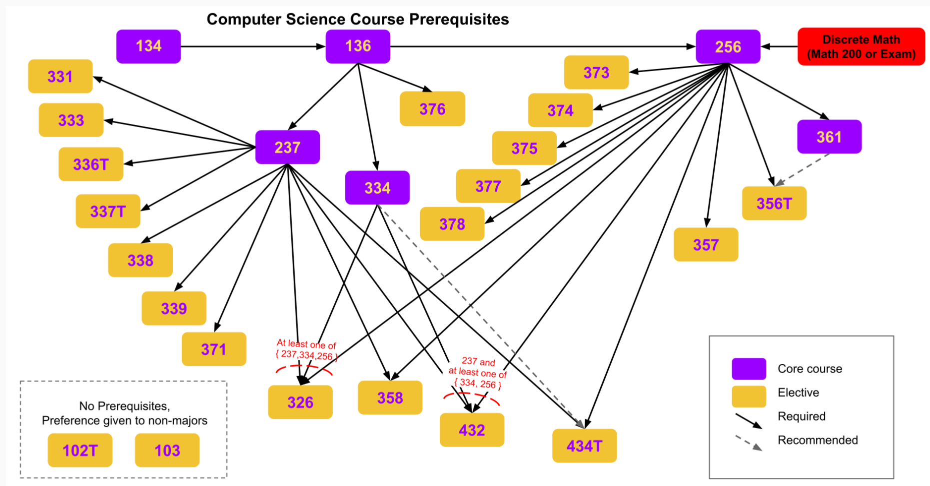


Goal of the game: given two words, transform one into the other by changing one letter at a time, always maintaining a legal word.

CORD $\to$ WORD $\to$ WORM

Two words are connected if they differ by one letter.

CS Prerequisite Graph



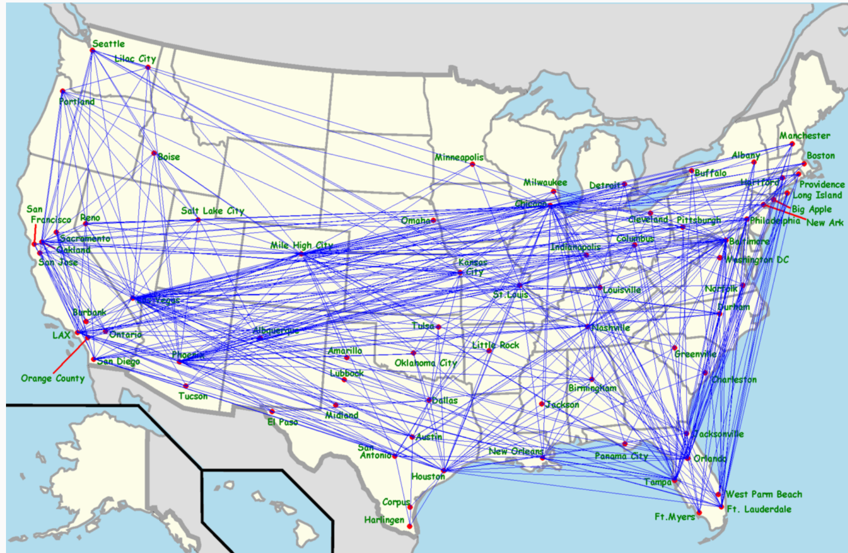
What are the nodes? Edges?

Is this graph a tree?

Relationships to Trees

- Every tree is a graph!
- But not every graph is a tree.

Flight Routes



Basic Definitions

- A graph $G = (V, E)$ consists of two sets: V, the vertices (also known as nodes) of G; and E, the edges of G
- Each edge e is defined by a pair of vertices: its indicent vertices.
- We write $e = \{u, v\}$. (If the edge is directed, it's from u to v; otherwise it's between u and v.)
- We say that u and v are *adjacent* (they are connected by an edge)
- The *neighbors* of v are all nodes u that are adjacent to v

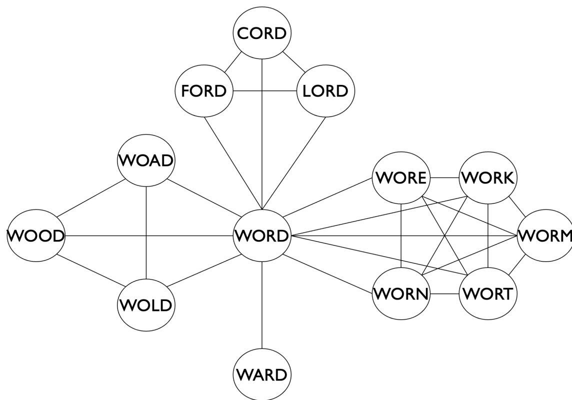
Paths in a graph

- Often we want to traverse a graph from one node to another
- A *path* from a vertex u to a vertex v in G is an alternating sequence of vertices and edges:

$$u = v_0, e_1, v_1, e_2, \ldots, v_{k-1}, e_k, v_k = v$$

- Each edge must be connected to the previous vertex and vice-versa.
- The *length* of a path or cycle is the number of edges in the sequence
- The *distance* from a start vertex s to a vertex v is the length of the shortest path from s to v

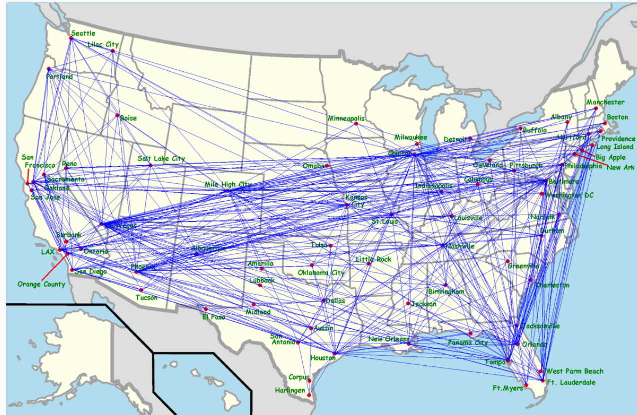
Word Game



Goal of the game: given two words, transform one into the other by changing one letter at a time, always maintaining a legal word.

What does a *path* mean in this graph? What is the meaning of the length of the path? What is the *distance* from one word to another?

Flight Routes



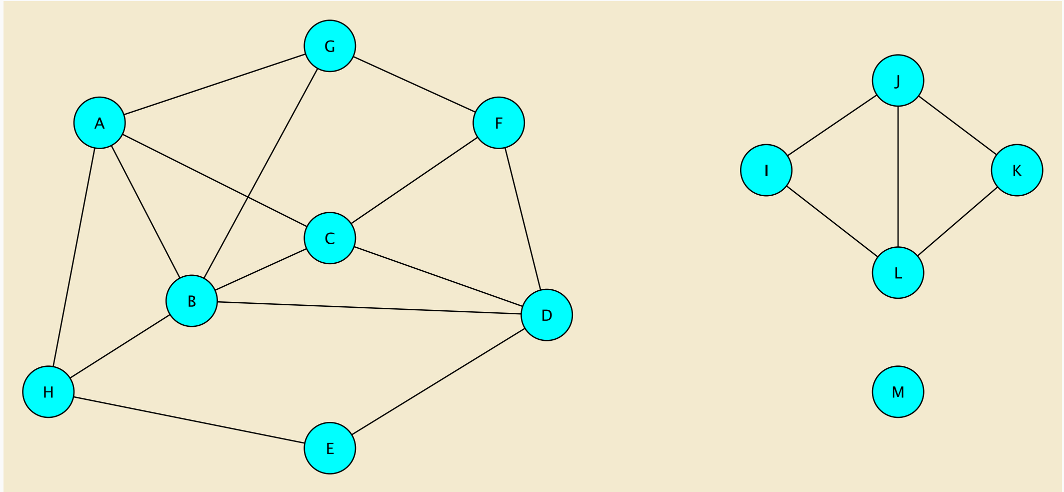
What is a path? What is the length of the path? What is the distance between two airports?

Takeaway: graphs really can represent a very broad variety of real-world problems.

Reachability and Connectedness

- A vertex v is *reachable* from a vertex u if there is a path from u to v in G
- What does it mean if one vertex is reachable from another in the graph of flights?

Reachability Example



Determining Reachability

First Graph Algorithm Example

- Let's say we're given a graph G, and two vertices u and v in G
- How can we tell if u is reachable from v?
- Are there any nodes for which this question is *easy*?
- Start: check all neighbors of v. See if any of them are u.
- Then repeat! Check all of their neighbors, and so on.
- How can we implement this?

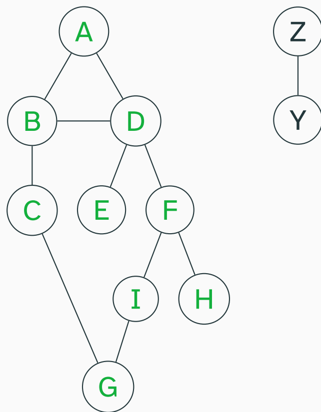
What operations do we need on graphs?

- Given a vertex v, need to be able to find *all adjacent vertices* of v (all neighbors)
- Might also want:
  - Given vertices u and v, determine if they are adjacent
  - Given a vertex v and an edge e, determine if v is incident to e
  - Get *all adjacent edges* of v

Implementing our idea

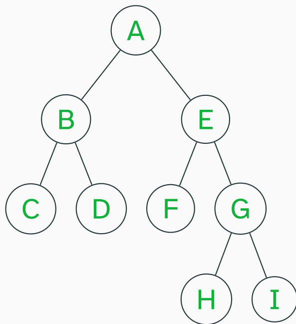
- Basic premise: start with v. Check its neighbors, then their neighbors, and so on.
- This algorithm is called *Breadth-First Search* (usually abbreviated BFS)
- What does BFS look like?

Breadth-First Search Example 1



Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 2



Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Plan

- Breadth-First Search is a lot like level order traversal!
- Start with a node. Explore its children in order. Then, explore their children (in the same order)
- Plan: use a queue to store nodes that are waiting
- Let’s plan this out in more detail
- Use *pseudocode*: a description of an algorithm in code-like notation (without worrying about language-specific details)

Breadth-First Search

```
1  // pre: all vertices are marked as unvisited
2  BFS(G, v) // Do a breadth-first search of G starting at v
3      Create empty queue Q
4      enqueue v
5      mark v as visited
6      while Q isn't empty:
7          current ← Q.dequeue()
8          for each unvisited neighbor u of current:
9              add u to Q
10             mark u as visited
```

Breadth-First Search for Distance

- Key property of BFS: closer nodes are visited first
- Let's gain some intuition about this
- Claim: If a node is at distance 1 from the start node, it is visited before all nodes at distance > 1 from the start node
  - Why?
  - We add the neighbors of the start vertex first; then they're marked as visited so they're not explored again
- Claim: If a node is at distance 2 from the start node, it is visited before all nodes at distance > 2 from the start node
  - Why?
  - We add the neighbors of the neighbors of the start vertex next; then they're marked as visited so they're not explored again

Breadth-First Search for Distance

- We explore all nodes at distance d before any node at distance $> d$
- The first time we explore a node, we have found the *shortest path* to the node.
- BFS effectively calculates shortest paths!
- Question (in pairs): let's say we're running BFS, and we explore a node for the first time. How can we get the shortest path to that node?