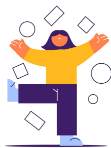
Lec 32: Graphs Contd.

May 11, 2026

Admin



- Practice final as soon as I can
- Lab Wednesday
- Review Friday; come with questions!
- Any questions?

Graphs

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 incident 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 vertices u that are adjacent to v

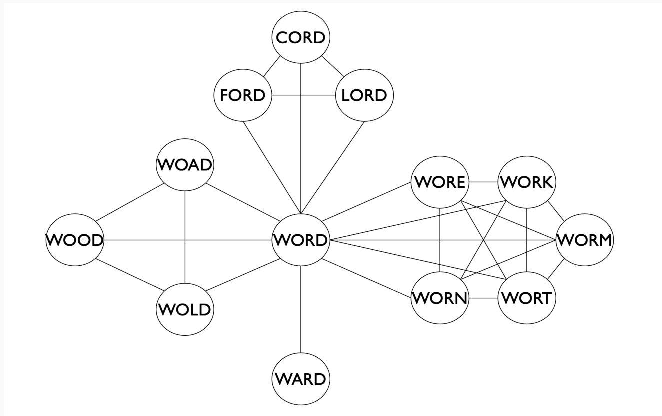
Paths in a graph

- Often we want to traverse a graph from one vertex 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

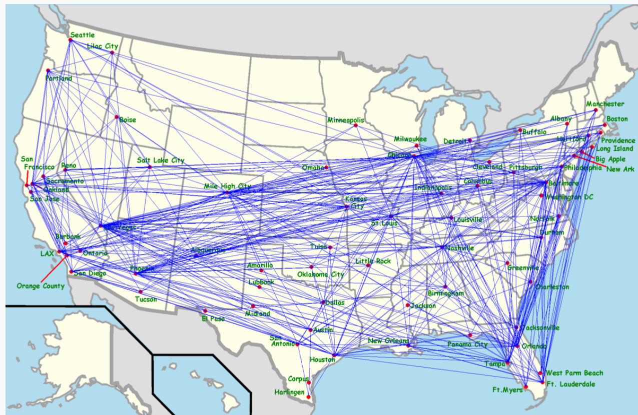
Word Game



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



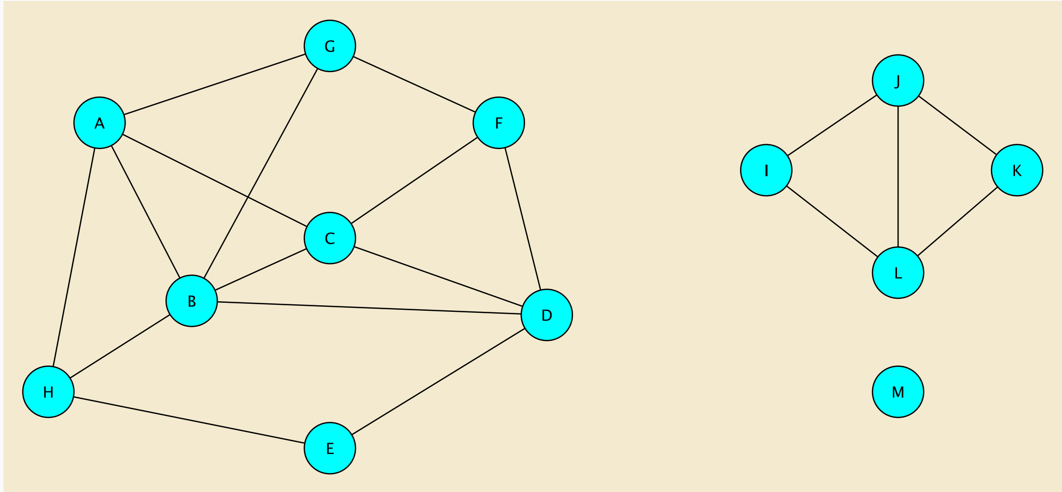
What is a path? What is the length of the path? What is the distance between two airports?

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 vertices 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?

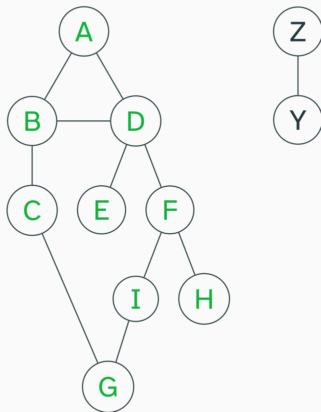
What operations do we need on graphs?

- Given a vertex v, need to be able to find *all adjacent vertices* of v (all neighbors)
- Might also want (not for this algorithm, but in the future):
  - 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

Implementing our idea

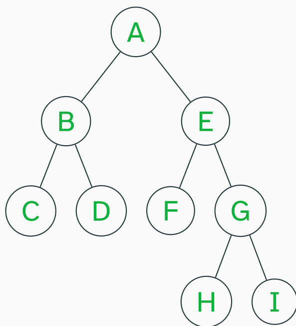
- Basic premise: start with v. Check its neighbors, then their neighbors, and so on.
- This algorithm is called *Breadth-First Search* (usually abbreviated BFS)
- What does BFS look like?

Breadth-First Search Example 1



Vertices that we have already traversed are marked in green. The vertex we are currently traversing is marked in orange.

Breadth-First Search Example 2



Vertices that we have already traversed are marked in green. The vertex we are currently traversing is marked in orange.

Plan

- Breadth-First Search is a lot like level order traversal!
- Start with a vertex. Explore its children in order. Then, explore their children (in the same order)
- Plan: use a queue to store vertices 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  BFS(G, v) // Do a breadth-first search of G starting at v
2      Ensure all vertices in G are unvisited
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
```

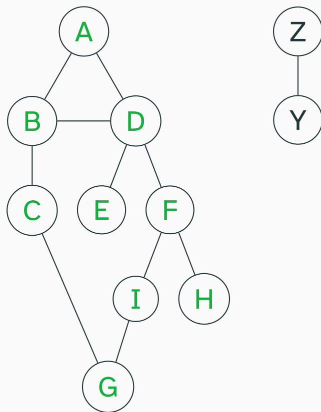
Breadth-First Search for Distance

- Key property of BFS: closer vertices are visited first
- Let's gain some intuition about this
- Claim: If a vertex is at distance 1 from the start node, it is visited before all vertices at distance > 1 from the start vertex
  - 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 vertex is at distance 2 from the start vertex, it is visited before all vertices at distance > 2 from the start vertex
  - 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 vertices at distance d before any vertex at distance $> d$
- The first time we explore a vertex, we have found the *shortest path* to the vertex.
- BFS effectively calculates shortest paths!
- Question (in pairs): let's say we're running BFS, and we explore a vertex for the first time. How can we get the shortest path to that vertex?
    - Let's revisit the BFS example first

Breadth-First Search Example

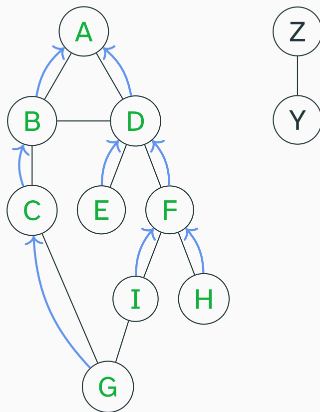


Vertices that we have already traversed are marked in green. The vertex we are currently traversing is marked in orange.

Finding the shortest path to a vertex

- For neighbors of the start vertex, the path is easy (just go back to the start)
- For *their* neighbors—vertices with distance 2—we want to store which neighbor of the start vertex they are adjacent to
- In general, for each vertex, store “who it was discovered by.”
- Put another way: this always stores a link to a *closer* vertex
- Going backwards gets you the shortest path!

Breadth-First Search Example



Vertices that we have already traversed are marked in green. The vertex we are currently traversing is marked in orange.

Implementing Graphs

What do we want out of a Graph implementation?

- Each vertex has a *label*
    - Could be just a letter or number (like in our examples so far)
    - Name of a city
    - A word in the `WordLadder` lab
- Our graph operations should use *labels*
    - Example: get me the vertex with a given label
- Can we store a graph like a linked list/tree?
    - Data structure stores some metadata
    - Nodes store the structure itself

Graph Implementation

- Two methods for storing a graph:
  - Adjacency List (we’ll use this one in the lab; similar to linked list/tree)
  - Adjacency Matrix (different approach; closer to an array/arraylist)
- Let’s discuss both, and the tradeoffs between them

Adjacency List Representation

- Store a class `Vertex<E>`
  - Generic type `E` stores the type of the vertex label
- Each `Vertex<E>` stores a list of its neighbors
  - We'll use an `ArrayList` in the lab, though a `LinkedList` is more common
- What is the type of the `ArrayList`?
  - It stores a list of `Vertex<E>`. So it is an `ArrayList<Vertex<E> >`

Adjacency List Representation

`Vertex<E>`:

- Methods to get the name of the vertex
- Method to get all neighbors of a vertex
- Keep a `boolean visited` for BFS
- Some getter/setter methods, including a method to add a new neighbor

Adjacency List Representation: Graph class

- How can we add a new edge in our graph, given the *labels* of two vertices in our graph?
- Need a way to get from `E label` to the `Vertex<E>` that has the matching label.
- What data structure should we use?
  - A hash table! Keys are the labels; value of a key is the `Vertex<E>` with that label
  - Keys of the hash table have type `E`, values have type `Vertex<E>`
- To add an edge, look up each `Vertex<E>` using the hash table. Add each to the others' neighbor list

Adjacency List Representation: Graph Class

- How do we add a new vertex?
  - Create the `Vertex<E>`; add an entry to the hash table
- How do we determine if a vertex of a given label exists in the graph?
  - Check if a key with the label exists in the hash table
- Let’s do an example on the board

When Does an Adjacency List Fall Short?

- How do we check if a given edge (between a vertex with label `label1` and another with label `label2`) is contained in the graph?
- Use the hash table to look up the vertex with label `label1`. Check through all of its neighbors to see if `label2` is one of them.
- How much time does this take?
  - $O(\#\text{neighbors})$
  - This *can* be as bad as $O(n)$
- Takeaway: adjacency list is very slow if you want to check if a given edge exists, and there ar many edges in the graph (vertices have many neighbors)

Adjacency Matrix Representation

- Let's store a graph all in one place!
- An $n \times n$ boolean matrix
- Entry $[i][j]$ has a 1 if the ith vertex shares an edge with the jth verted
- Let's do an example on the board

Adjacency Matrix Representation

- How do we keep track of vertex labels?
- How do we check if two vertices share an edge?
- How do we get all neighbors of a vertex? How long does it take?
- How much space does this representation take?
- How do we add/remove vertices? What do we need to store to ensure that this is efficient?