Introduction to Graphs

Instructors: Sam McCauley and Dan Barowy

May 2, 2022

Admin

- Last week’s quiz back Wednesday
- Any questions?

Graphs

Purpose of Graphs

- Data structures up until this point have mostly been designed to *store data* for easy access

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)

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

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

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 represented stored words

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 represented 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*

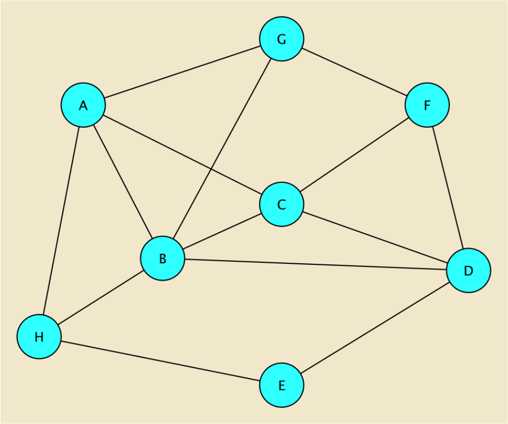
Graphs

- Graphs consist of *nodes* and *edges*
- Much like a tree! But no restrictions on how edges may connect nodes

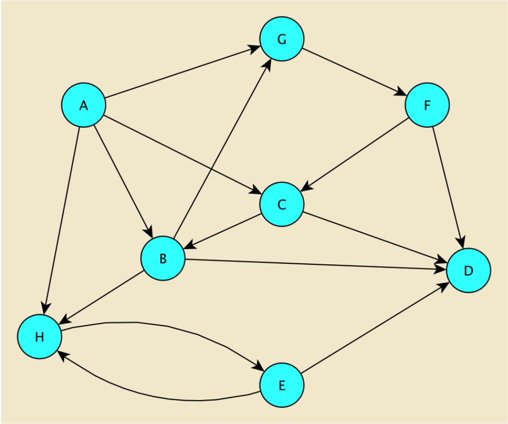
Graphs

- 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



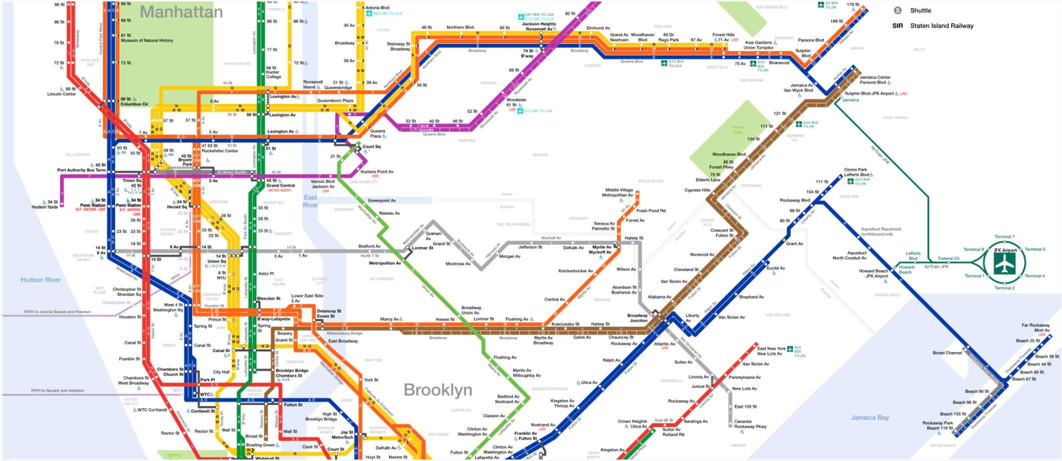
An undirected graph



A directed graph

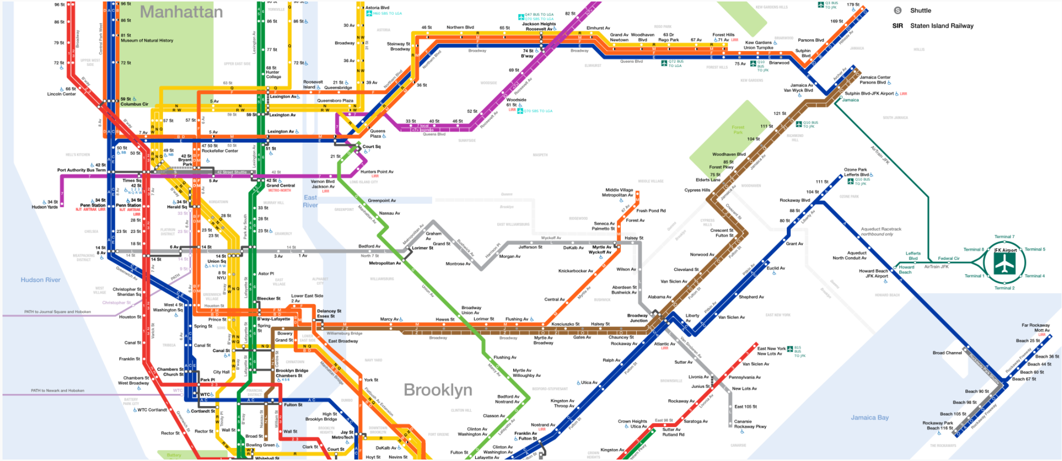
We usually draw graphs much like we drew trees. In directed graphs, we show the direction of an edge with an arrow.

height



What are the nodes here? Edges?

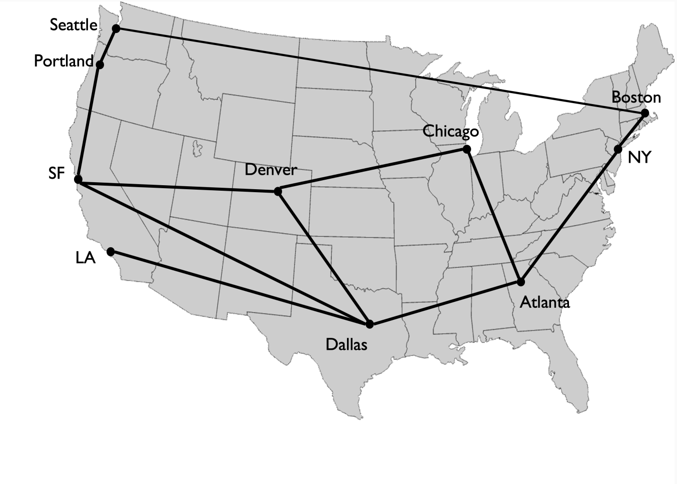
height



What are the nodes here? Edges?

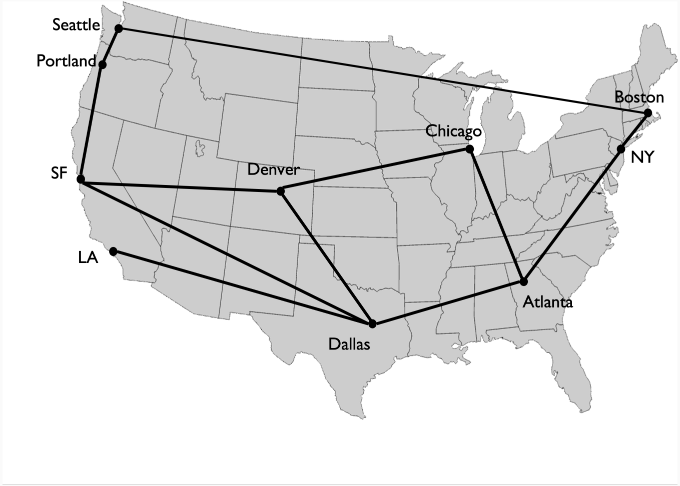
Nodes: subway stops. An edge between two stops if there is a train between them.

(Simplified) US Train Map



What are the nodes here? Edges?

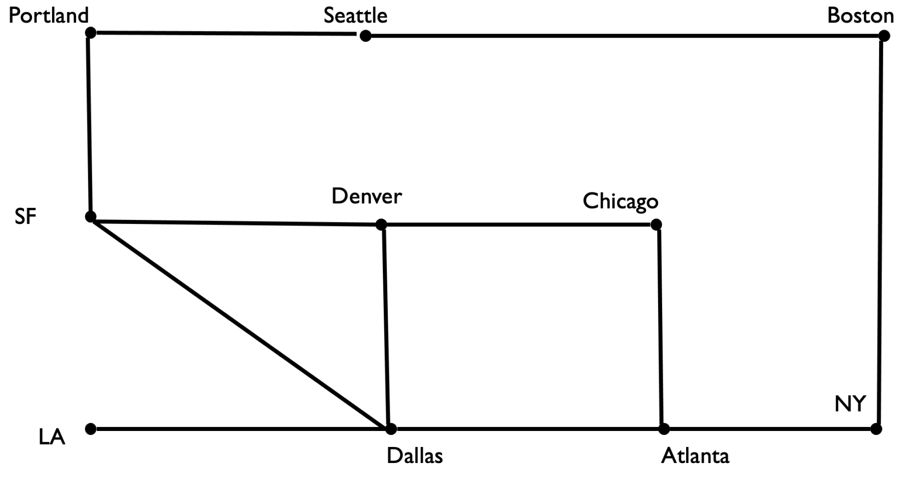
(Simplified) US Train Map



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.

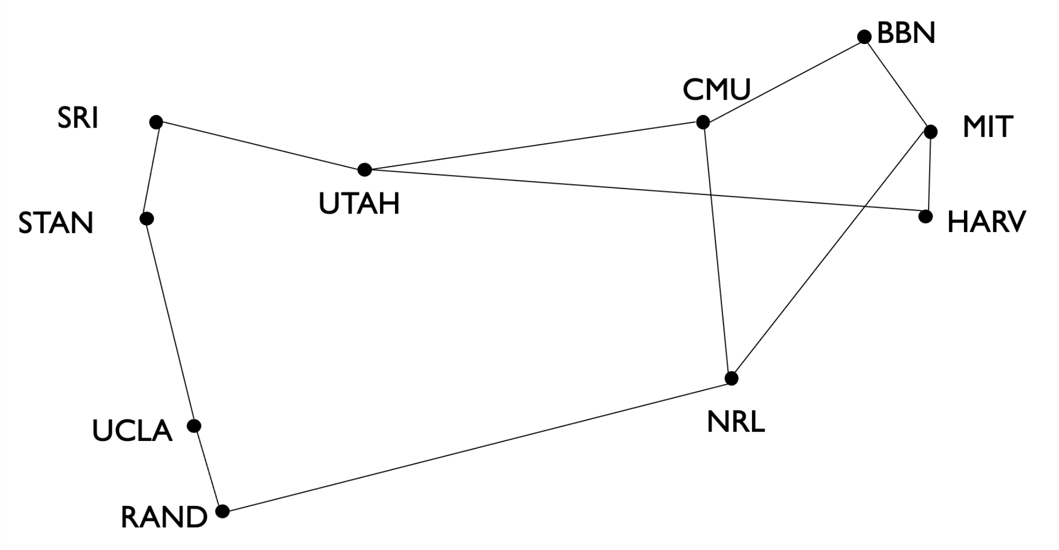
(Simplified) US Train Map



What are the nodes here? Edges?

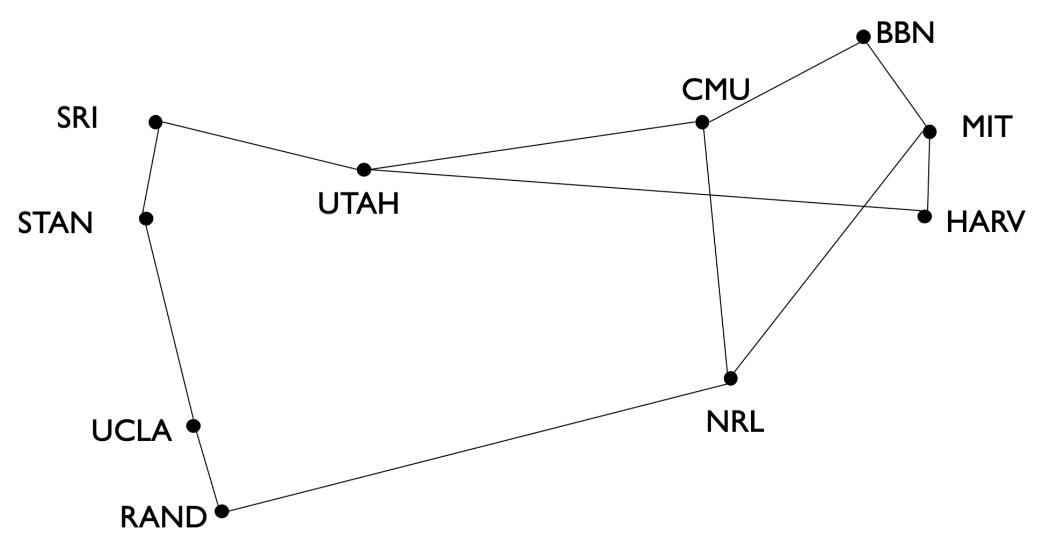
Nodes: cities. An edge between two cities if there is a train between them. Note that it's not important how we draw the edges.

Internet Circa 1972



What are the nodes here? Edges?

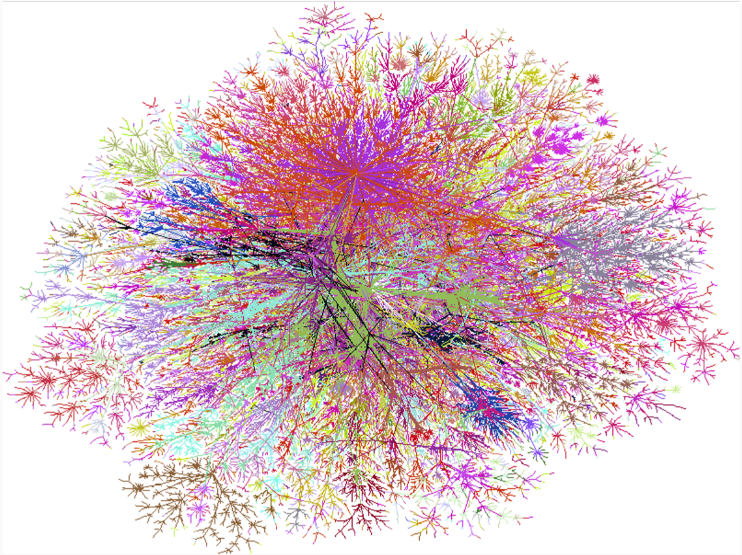
Internet Circa 1972



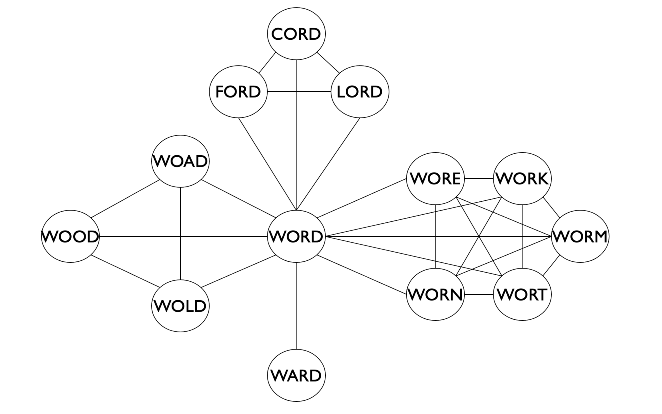
What are the nodes here? Edges?

Nodes: network access points. Edges represent a connection.

Internet Circa 1998



Word Game

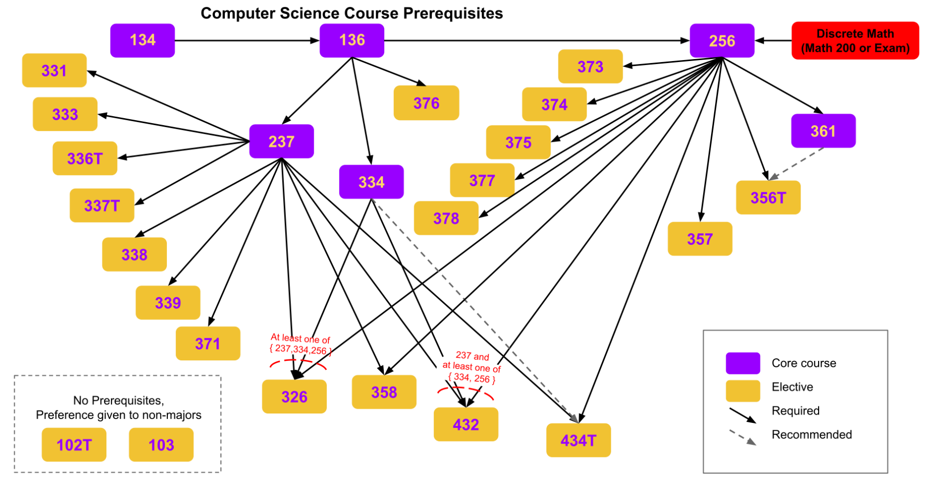


Goal of the game: given two words, transform one into the other by changing one letter at a time, always maintaining a legal word.

CORD $\rightarrow$ WORD $\rightarrow$ WORM

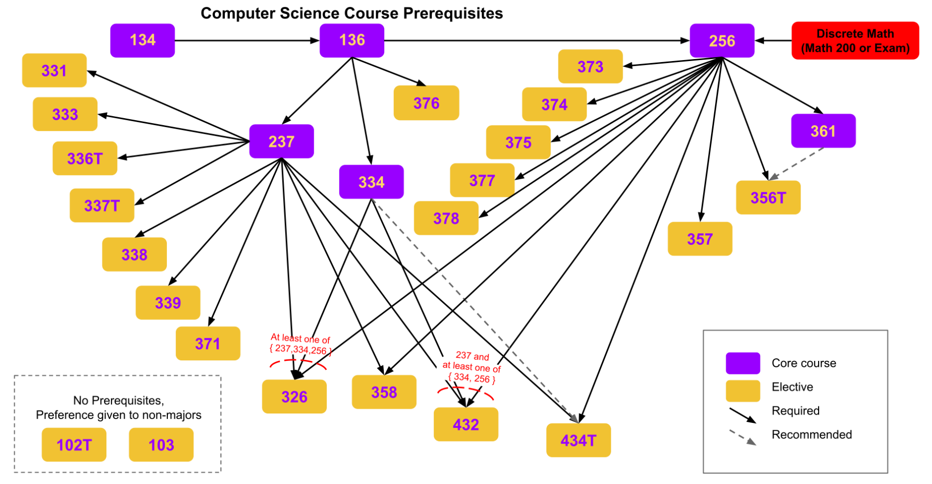
Two words are connected if they differ by one letter.

CS Prerequisite Graph



What are the nodes? Edges?

CS Prerequisite Graph



What are the nodes? Edges?

Is this graph a tree?

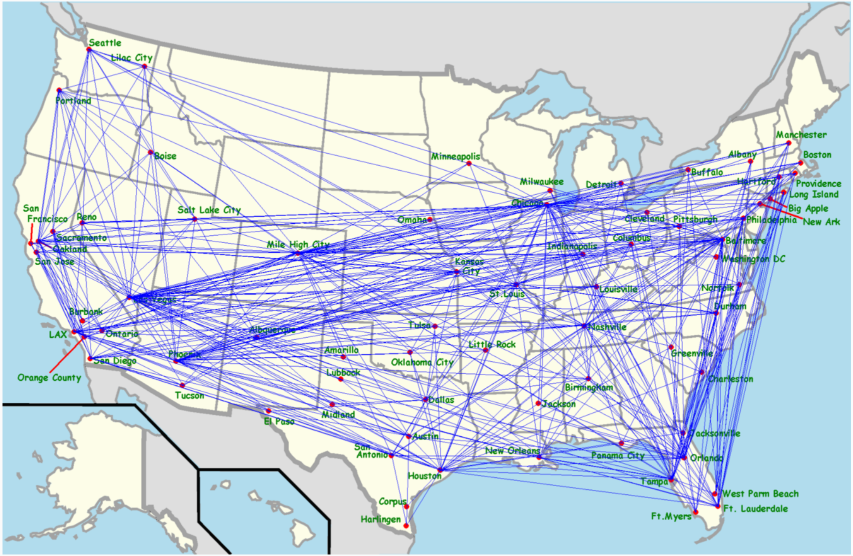
Relationships to Trees

- Every tree is a graph!

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

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.

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.)

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)

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

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:

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

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 each $e_i = \{v_{i-1}, v_i\}$ for all $i = 1, \ldots, k$

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 each $e_i = \{v_{i-1}, v_i\}$ for all $i = 1, \ldots, k$
    - Note that this needs to follow the direction of the edge in a directed graph; for an undirected graph the can go in either direction

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 each $e_i = \{v_{i-1}, v_i\}$ for all $i = 1, \ldots, k$
    - Note that this needs to follow the direction of the edge in a directed graph; for an undirected graph the can go in either direction
- No edge can appear more than once: $e_i \neq e_j$ if $i \neq j$

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 each $e_i = \{v_{i-1}, v_i\}$ for all $i = 1, \ldots, k$
  - Note that this needs to follow the direction of the edge in a directed graph; for an undirected graph the can go in either direction
- No edge can appear more than once: $e_i \neq e_j$ if $i \neq j$
- In a *simple path*, no vertex appears more than once

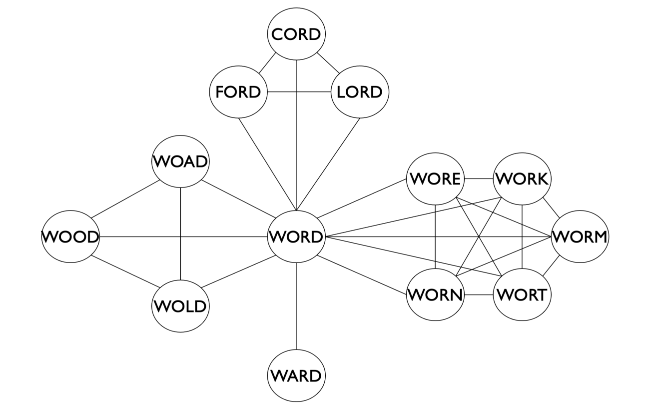
Path Continued

- A *cycle* is a simple path that begins and ends at the same vertex

Path Continued

- A *cycle* is a simple path that begins and ends at the same vertex
- The *length* of a path or cycle is the number of edges in the sequence

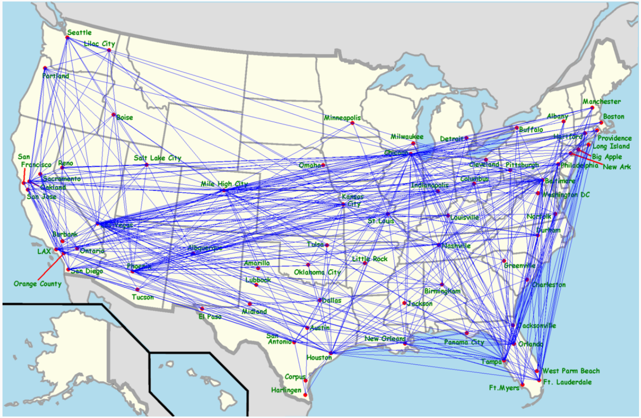
Word Game



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

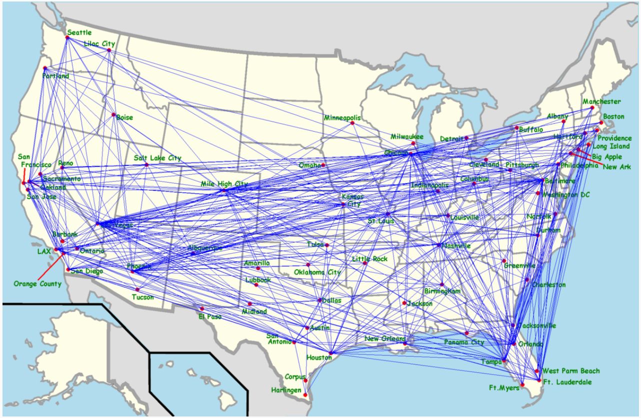
What does a *path* mean in this graph? What is the meaning of the length of the path?

Flight Routes



What is a path? What is a cycle? What is the length of the path?

Flight Routes



What is a path? What is a cycle? What is the length of the path?

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

Reachability and Connectedness

- A vertex v is *reachable* from a vertex u if there is a path from u to v in G
- A graph is *connected* if for every path of vertices u and v, v is reachable from u.

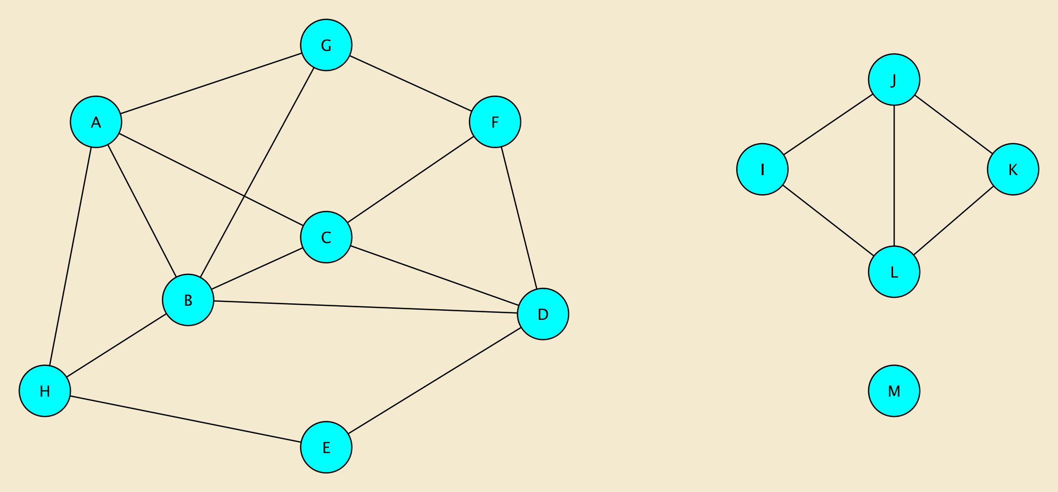
Reachability and Connectedness

- A vertex v is *reachable* from a vertex u if there is a path from u to v in G
- A graph is *connected* if for every path of vertices u and v, v is reachable from u.
- What does it mean if one vertex is reachable from another in the graph of flights? What does it mean if the flight graph is connected?

Connected Component

- All vertices reachable from v, along with all edges of G connecting two of them, constitute the *connected component* of v.

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

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?

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*?

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.

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 recurse! Check all of their neighbors, and so on.

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 recurse! 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

What operations do we need on graphs?

- Given a vertex v, need to be able to find *all adjacent vertices* of v

- Probably also want:

What operations do we need on graphs?

- Given a vertex v, need to be able to find *all adjacent vertices* of v
- Probably also want:
  - Given vertices u and v, determine if they are adjacent

What operations do we need on graphs?

- Given a vertex v, need to be able to find *all adjacent vertices* of v
- Probably 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

What operations do we need on graphs?

- Given a vertex v, need to be able to find *all adjacent vertices* of v
- Probably 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.

Implementing our idea

- Basic premise: start with v. Check its neighbors, then their neighbors, and so on.
- This algorithm is called *Breadth-First Search*

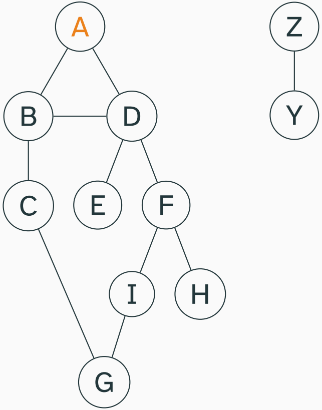
Implementing our idea

- Basic premise: start with *v*. Check its neighbors, then their neighbors, and so on.

- This algorithm is called *Breadth-First Search*

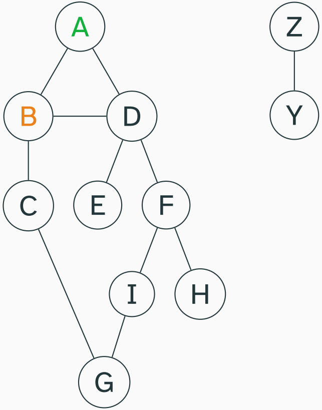
- What does this look like?

Breadth-First Search Example 1



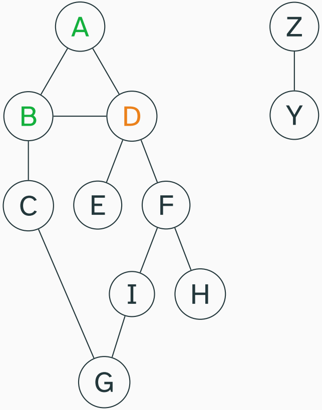
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 1



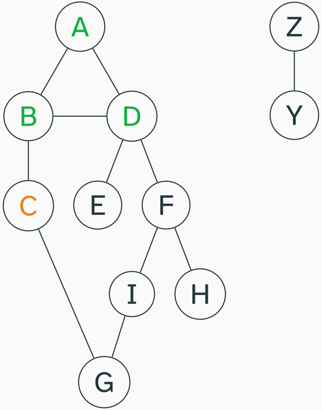
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 1



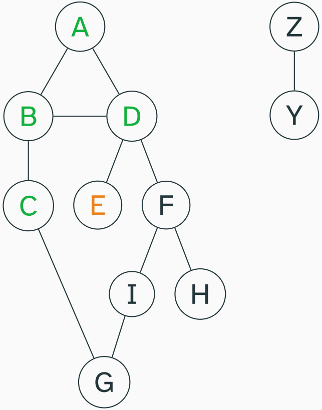
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 1



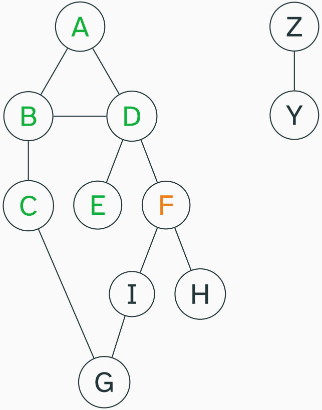
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 1



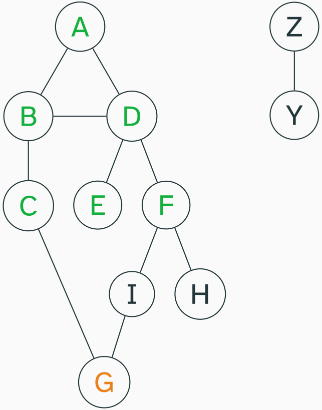
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 1



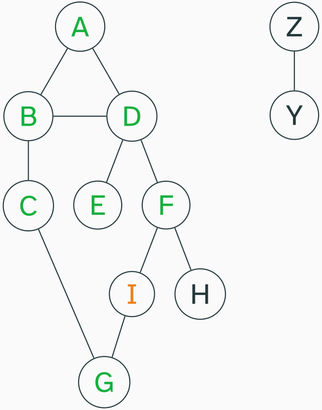
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 1



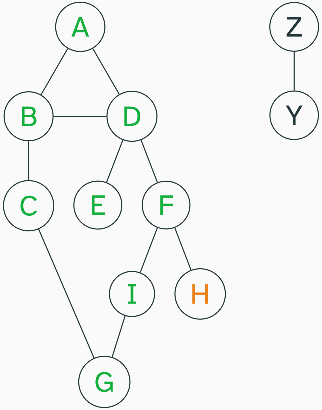
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 1



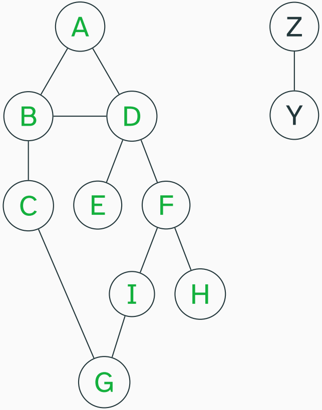
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 1



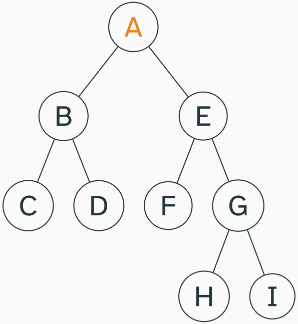
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 1



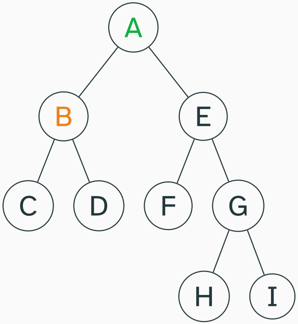
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 2



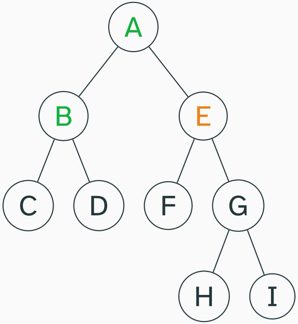
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 2



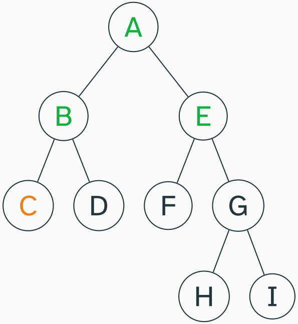
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 2



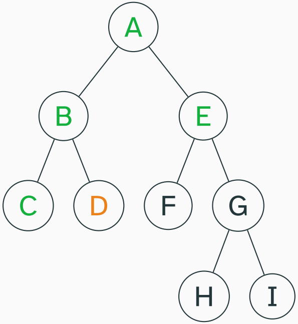
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 2



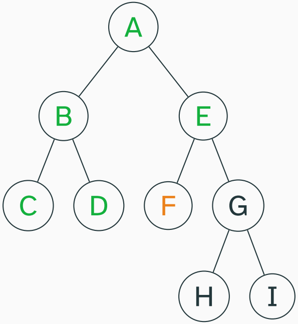
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 2



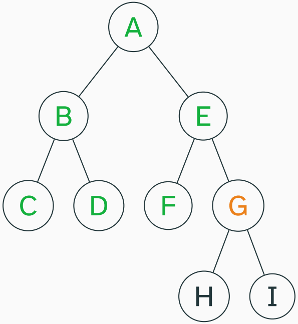
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 2



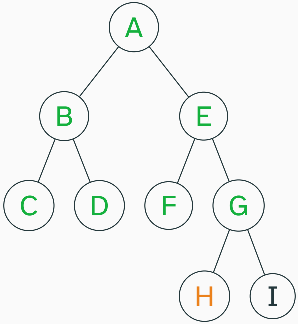
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 2



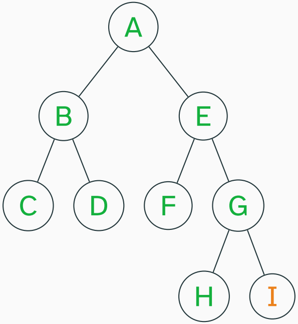
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

Breadth-First Search Example 2



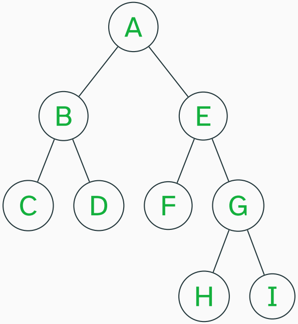
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

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.

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!

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

- 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

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

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

```
// pre: all vertices are marked as unvisited
BFS(G, v) // Do a breadth-first search of G starting at v
   count ← 0
   Create empty queue Q
   enqueue v
   mark v as visited
   count++
   while Q isn't empty:
      current ← Q.dequeue()
      for each unvisited neighbor u of current:
         add u to Q
         mark u as visited
         count++
   return count;
```

Implementing Graphs in Java

- What instance variables would we want our class to have? What methods?

Implementing Graphs in Java

- What instance variables would we want our class to have? What methods?
- How can we store the vertices?

Implementing Graphs in Java

- What instance variables would we want our class to have? What methods?
- How can we store the vertices?
- How can we store the edges?