

CSCI 136
Data Structures &
Advanced Programming

Lecture 26

Fall 2019

Instructors: B&S

Administrative Details

- Lab 9: Super Lexicon is online
 - Partners are permitted this week!

Last Time

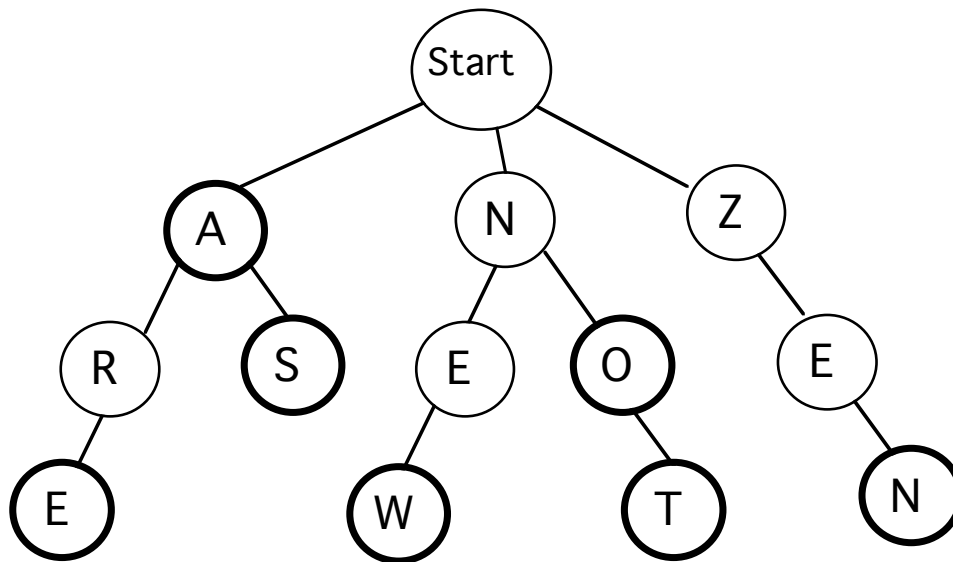
- *Efficient* Binary search trees (Ch 14)
 - AVL Trees
 - Height is $O(\log n)$, so all operations are $O(\log n)$
 - Red-Black Trees
 - Different height-balancing idea: height is $O(\log n)$
 - All operations are $O(\log n)$

Today's Outline

- Lab 9: Super Lexicon
- Introduction To Graphs
 - Basic Definitions and Properties
 - Applications and Problems

Lab 9 : Lexicon

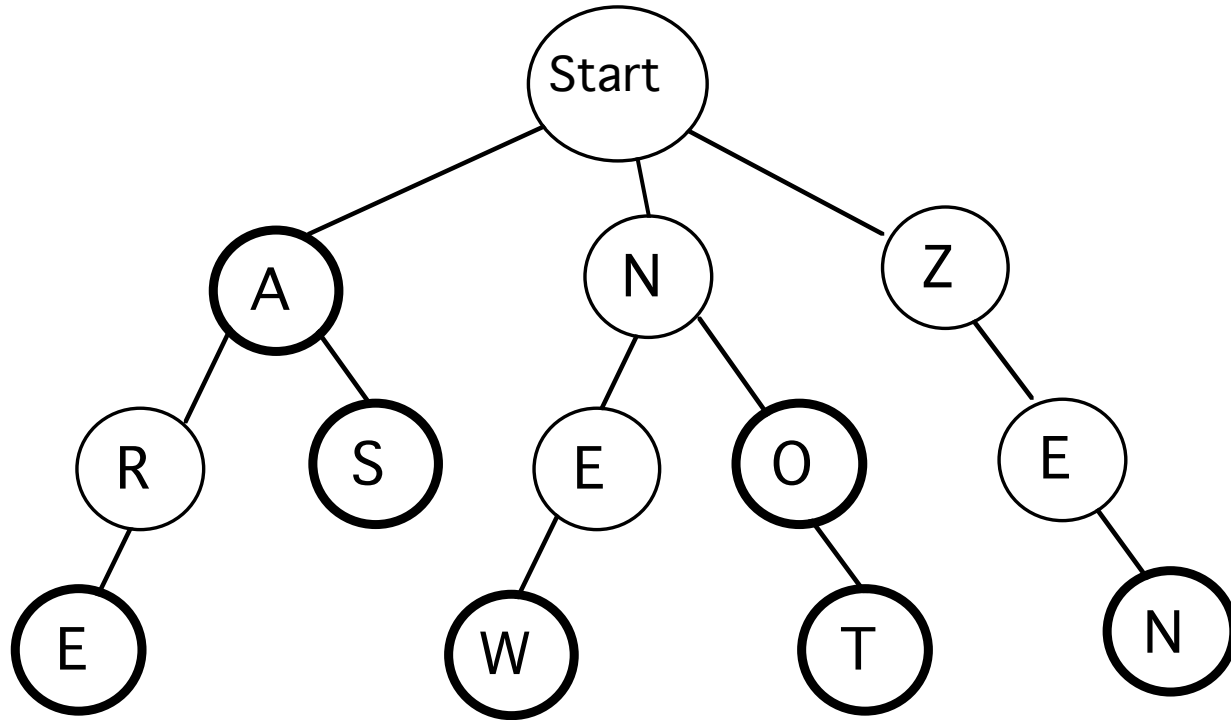
- Goal: Build a data structure that can efficiently store and search a large set of words
- A special kind of tree called a *trie*



Lab 9 : Tries

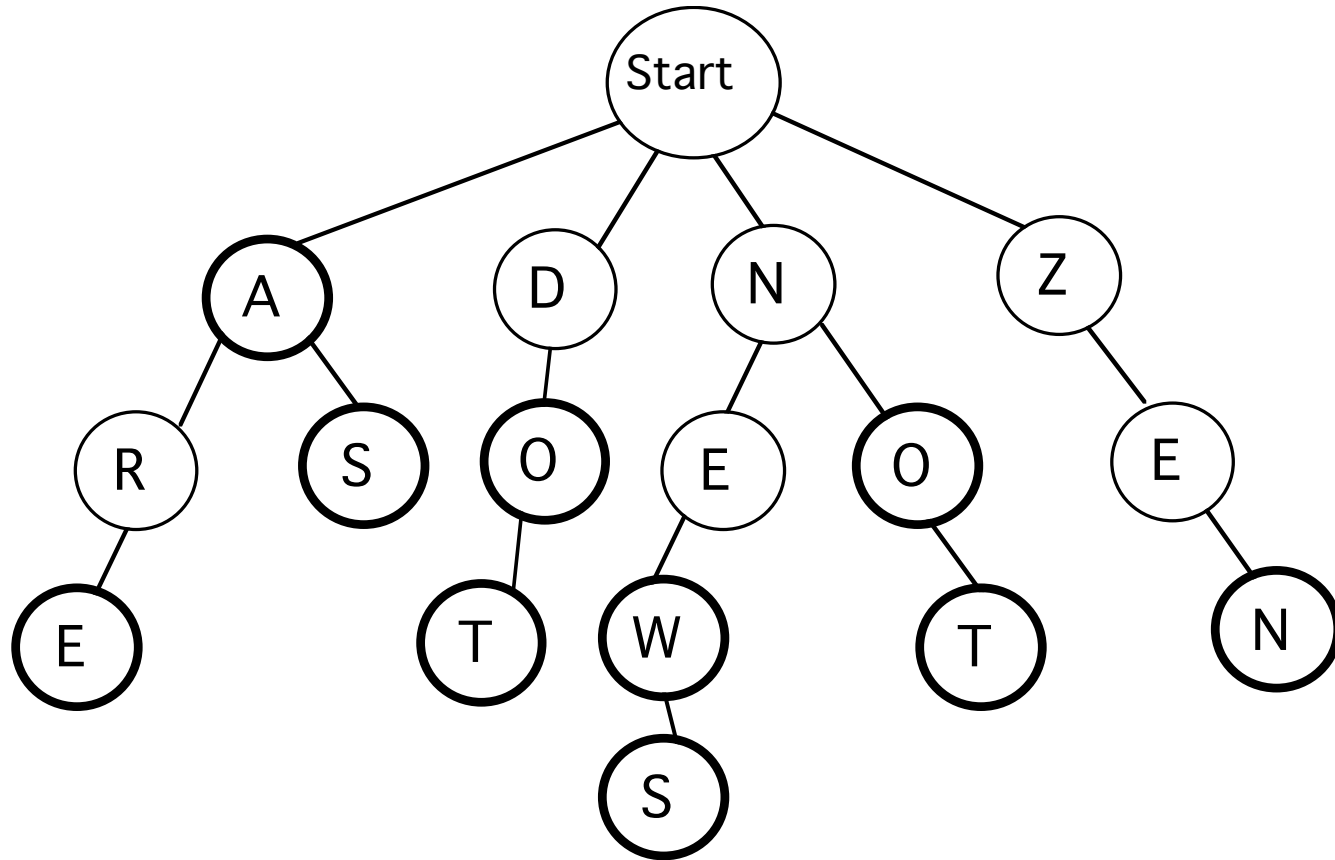
- A trie is a tree that stores words where
 - Each node holds a letter
 - Some nodes are “word” nodes (dark circles)
 - Any path from the root to a word node describes one of the stored words
 - All paths from the root form prefixes of stored words (a word is considered a prefix of itself)

Tries



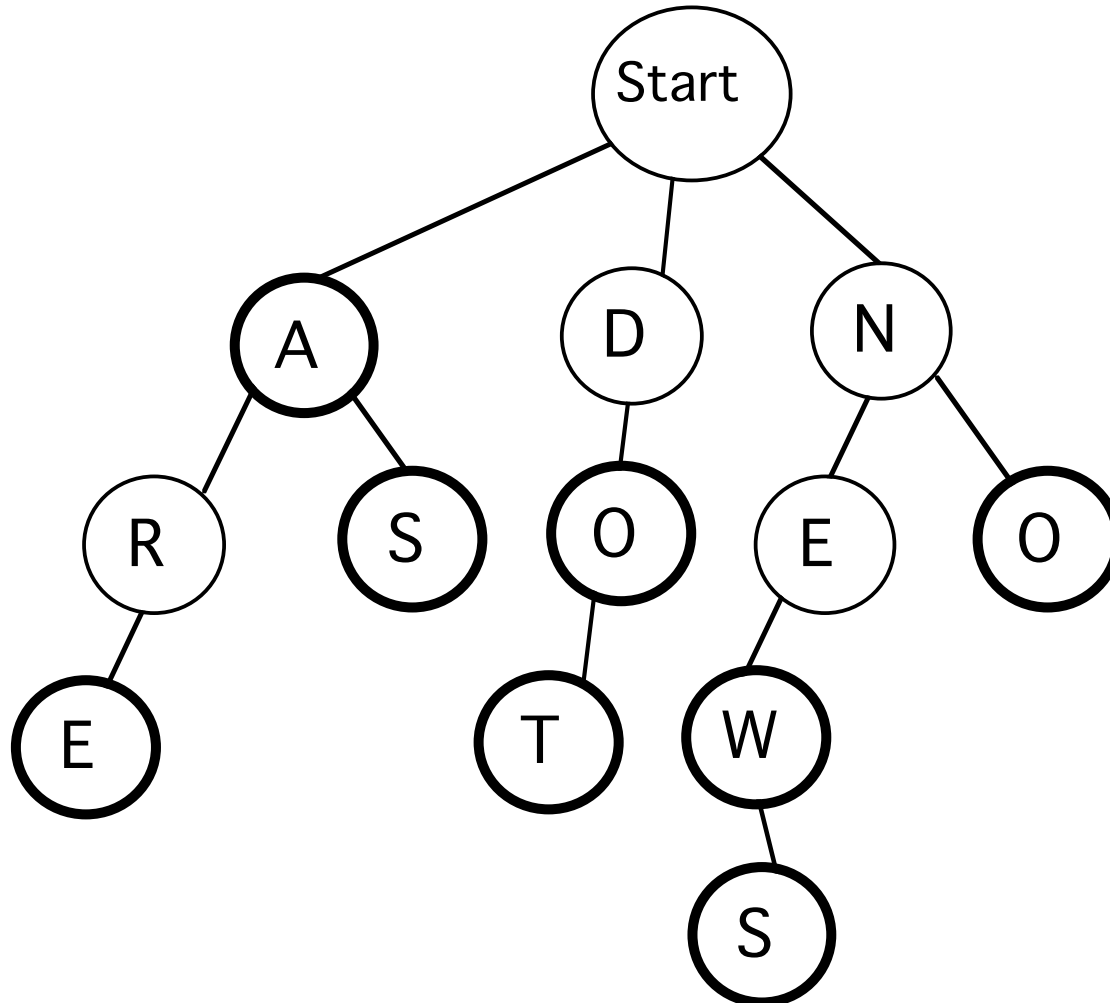
Now add “dot” and “news”

Tries



Now remove “not” and “zen”

Tries



Lab 9 : Lexicon

An interface that provides the methods

```
public interface Lexicon {  
    public boolean addWord(String word);  
    public int addWordsFromFile(String filename);  
    public boolean removeWord(String word);  
    public int numWords();  
    public boolean containsWord(String word);  
    public boolean containsPrefix(String prefix);  
    public Iterator<String> iterator();  
    public Set<String> suggestCorrections(String  
        target, int maxDistance);  
    public Set<String> matchRegex(String pattern);  
}
```

Lab 9

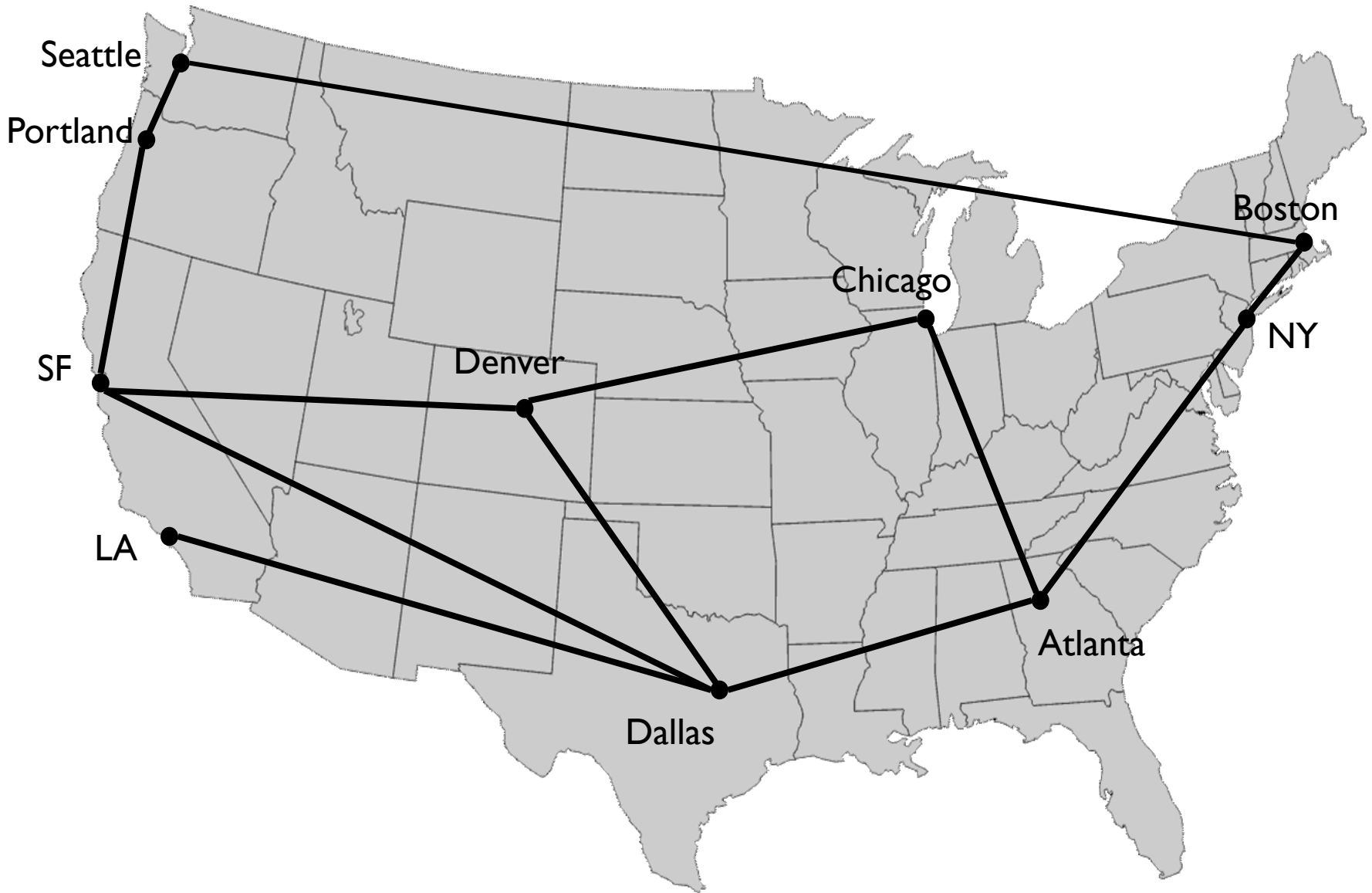
- Implement a program that creates, updates, and searches a Lexicon
 - Based on a LexiconTrie class
 - Each node of the Trie is a LexiconNode
 - Analogous to a SLL consisting of SLLNodes
 - LexiconTrie implements the Lexicon Interface
 - Supports
 - adding/removing words
 - searching for words and prefixes
 - reading words from files
 - Iterating over all words

Graphs Describe the World

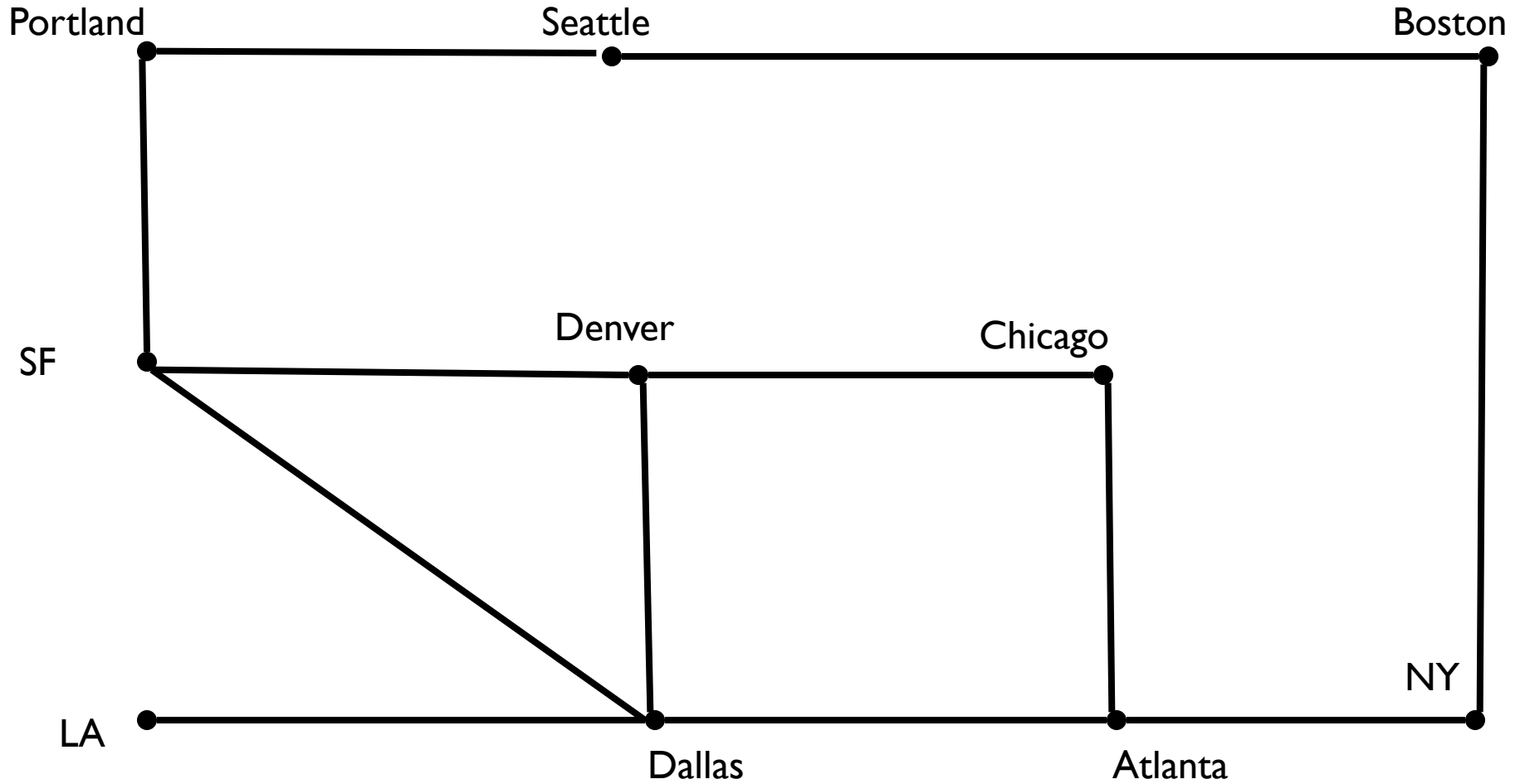
- Transportation Networks
- Communication Networks
- Molecular structures
- Dependency structures
- Scheduling
- Matching
- Graphics Modeling
-



Nodes = subway stops; Edges = track between stops

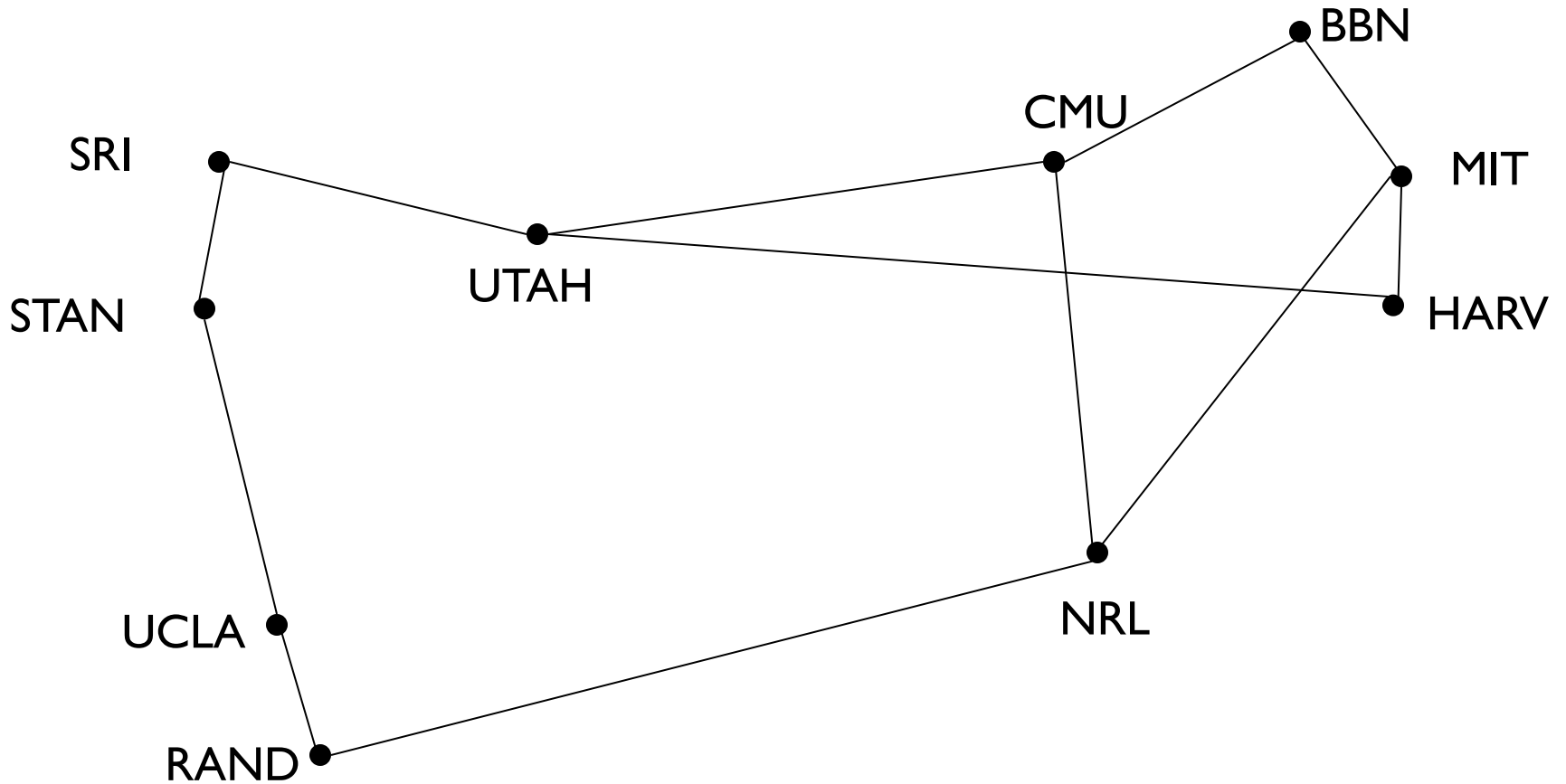


Nodes = cities; Edges = rail lines connecting cities

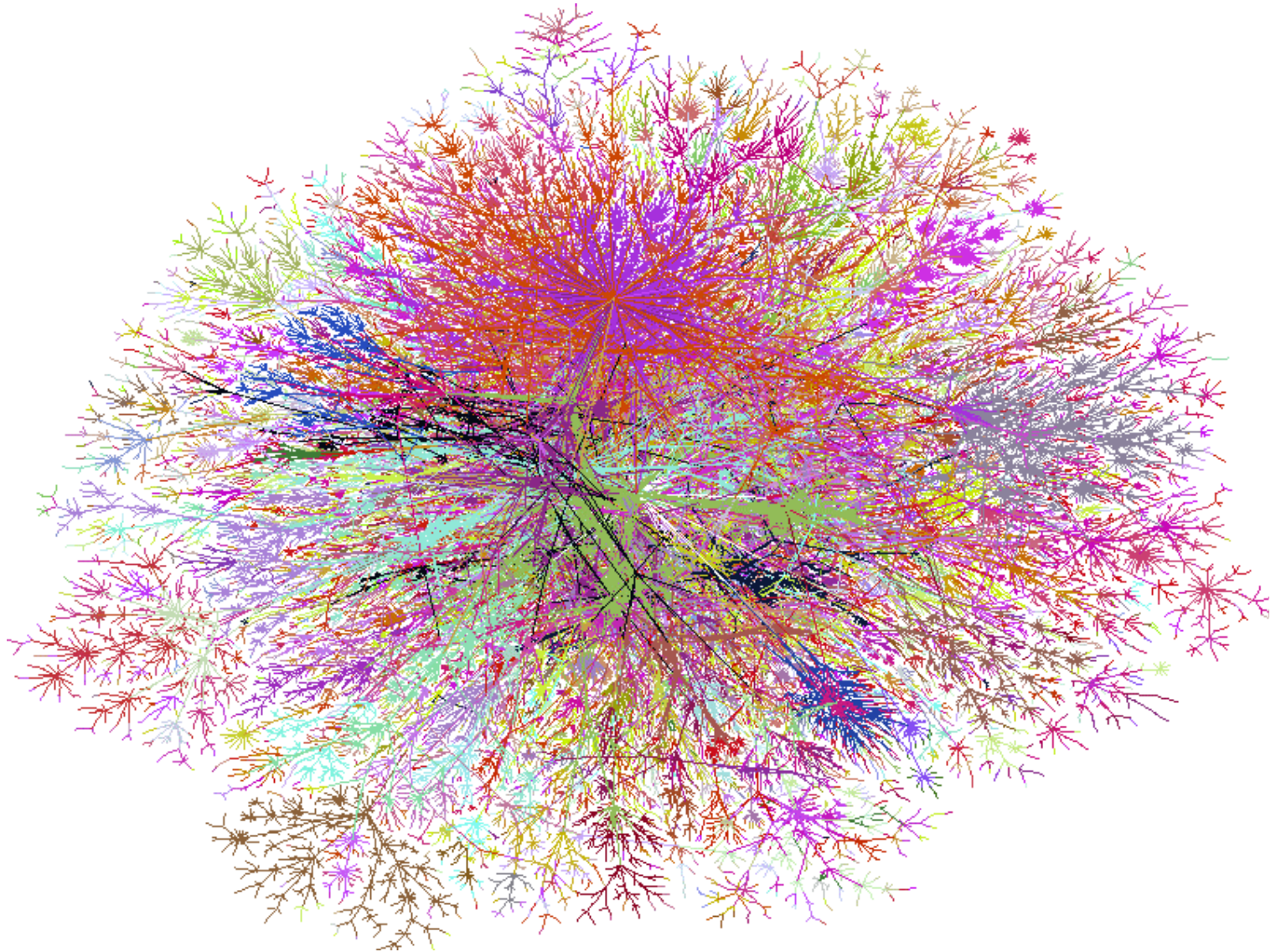


Note: Connections in graph matter, not precise locations of nodes

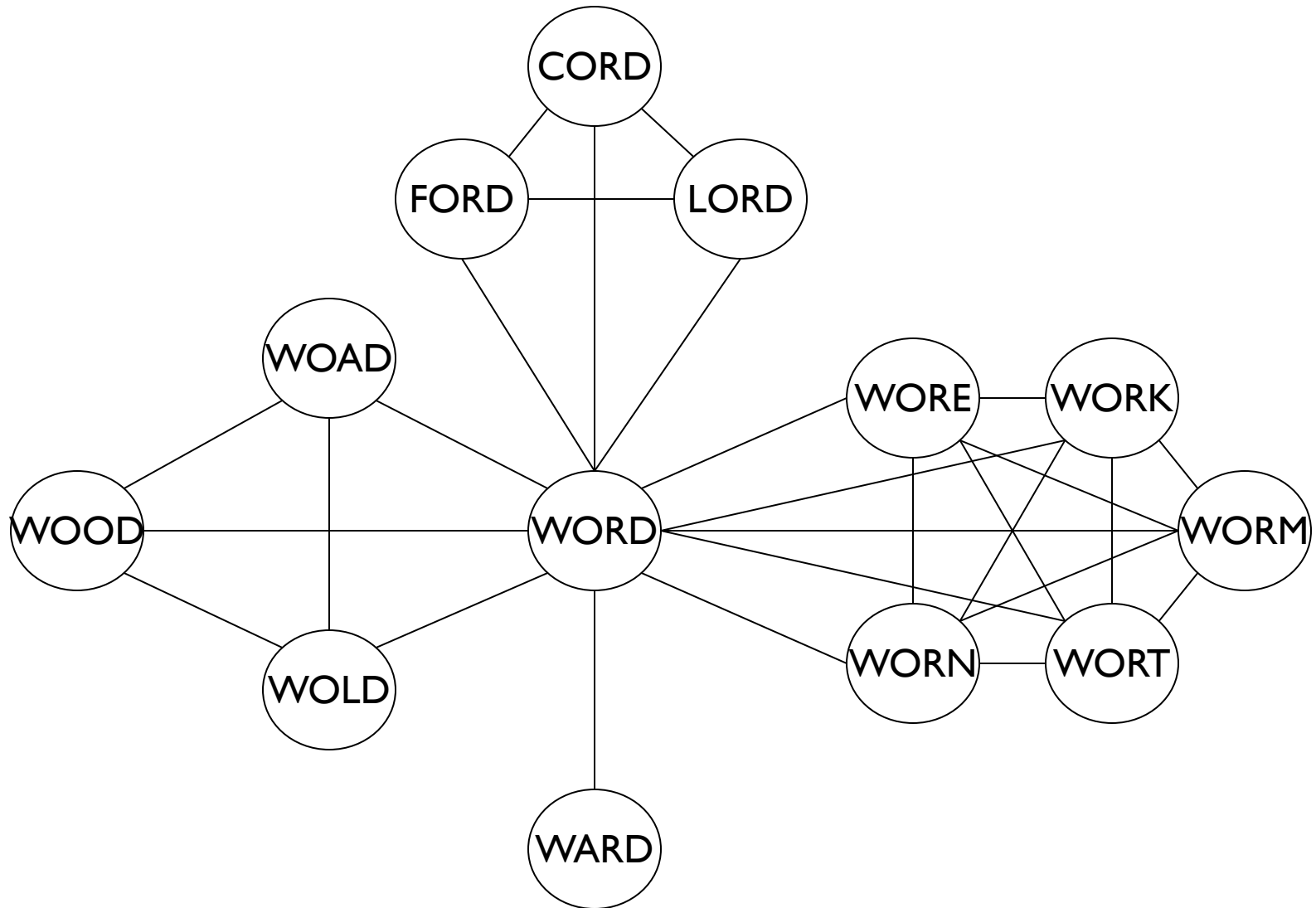
Internet (~1972)



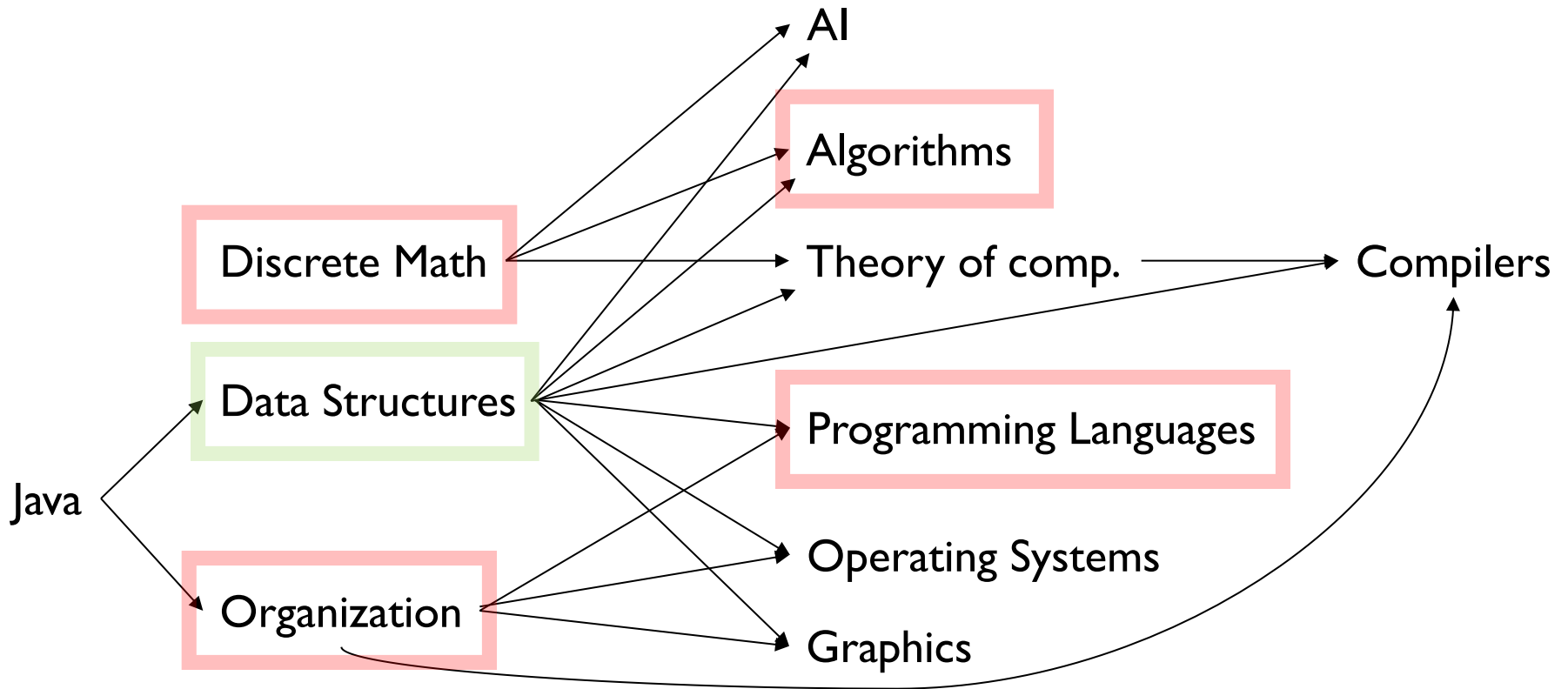
Internet (~1998)



Word Game

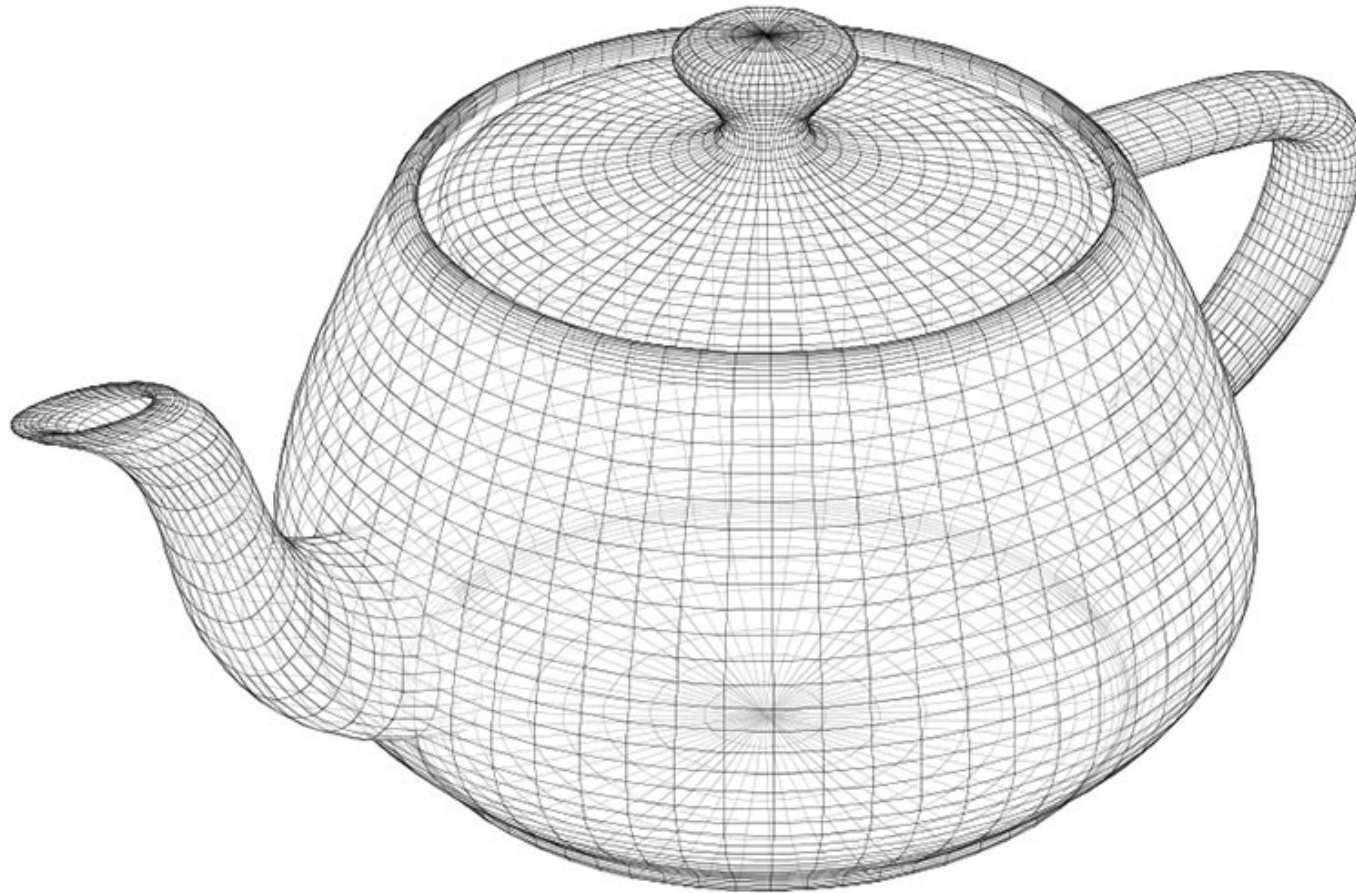


CS Pre-requisite Structure (subset)

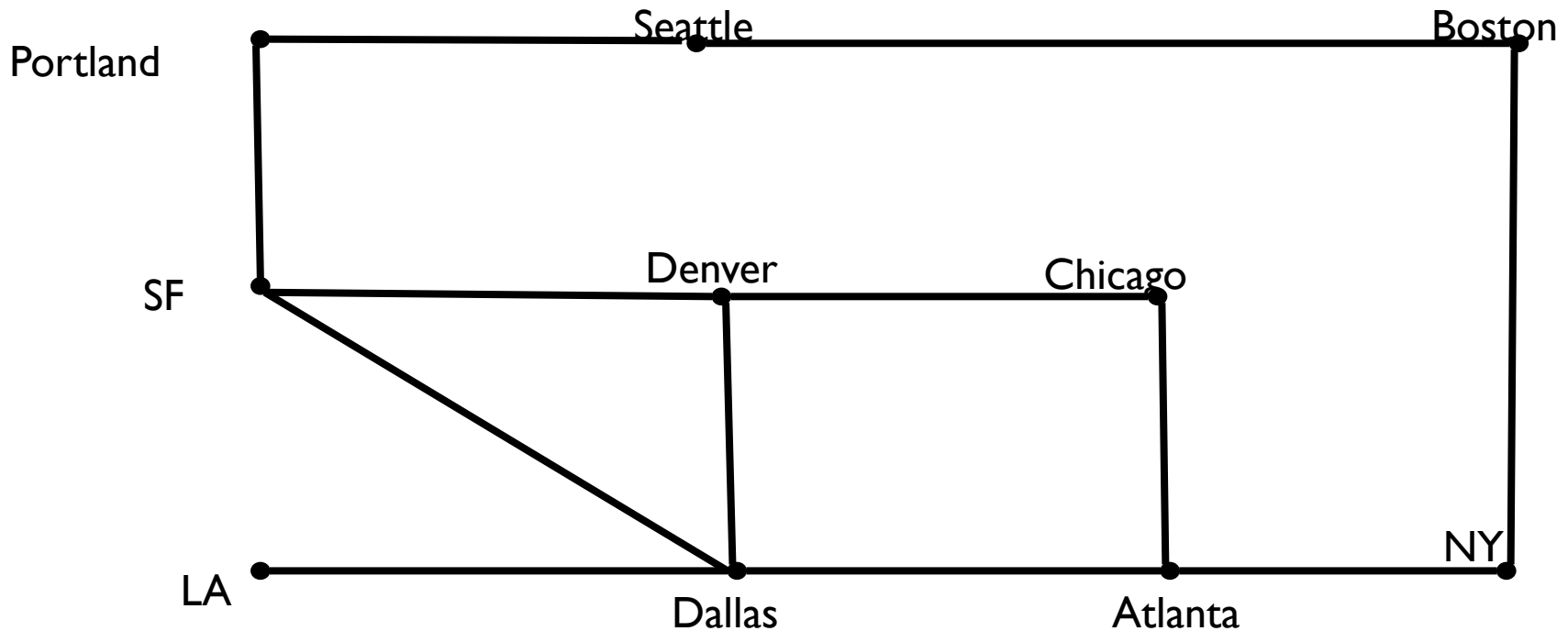


Nodes = courses; Edges = prerequisites ***

Wire-Frame Models



Basic Definitions & Concepts



Def'n: An *undirected graph* $G = (V, E)$ consists of two sets

- V : the *vertices* of G , and E : the *edges* of G
- Each edge e in E is defined by a set of two vertices: its *incident vertices*. We write $e = \{u, v\}$ and say that u and v are *adjacent*.

Walking Along a Graph

- A *walk* from u to v in a graph $G = (V, E)$ is an *alternating* sequence of vertices and edges

$$u = v_0, e_1, v_1, e_2, v_2, \dots, v_{k-1}, e_k, v_k = v$$

such that each $e_i = \{v_i, v_{i+1}\}$ for $i = 1, \dots, k$

- Note a walk starts and ends on a vertex
- If no *edge* appears more than once then the walk is called a *path*
- If no *vertex* appears more than once then the walk is a *simple path*

Walking In Circles

- A *closed walk* in a graph $G = (V, E)$ is a walk

$$v_0, e_1, v_1, e_2, v_2, \dots, v_{k-1}, e_k, v_k$$

such that each $v_0 = v_k$

- A *circuit* is a path where $v_0 = v_k$
 - No repeated edges
- A *cycle* is a *simple path* where $v_0 = v_k$
 - No repeated vertices (uhm, except for v_0 !)
- The length of any of these is the number of *edges* in the sequence

Little Tiny Theorems

- If there is a walk from u to v , then there is a walk from v to u .
- If there is a walk from u to v , then there is a path from u to v (and from v to u)
- If there is a path from u to v , then there is a simple path from u to v (and v to u)
- Every circuit through v contains a cycle through v
- Not every closed walk through v contains a cycle through v ! [Try to find an example!]

Another Useful Graph Fact

- Degree of a vertex v
 - Number of edges incident to v
 - Denoted by $\deg(v)$
- Thm: For any graph $G = (V, E)$

$$\sum_{v \in V} \deg(v) = 2 |E|$$

where $|E|$ is the number of edges in G

- Proof Hint: Induction on $|E|$: How does removing an edge change the equation?
 - Or: Count pairs (v, e) where v is incident with e

Reachability and Connectedness

- Def'n: A vertex v in G is *reachable* from a vertex u in G if there is a path from u to v
- v is reachable from u *iff* u is reachable from v
- Def'n: An undirected graph G is *connected* if for every pair of vertices u, v in G , v is reachable from u (and, of course, u from v)
- The set of all vertices reachable from v , along with all edges of G connecting any two of them, is called the *connected component of v*

Distance in Undirected Graphs

Def: The *distance* between two vertices u and v in an undirected graph $G=(V,E)$ is the minimum of the path lengths over all u - v paths.

- We write it as $d(u,v)$. It satisfies the properties
 - $d(u,u) = 0$, for all $u \in V$
 - $d(u,v) = d(v,u)$, for all $u,v \in V$
 - $d(u,v) \leq d(u,w) + d(w,v)$, for all $u,v,w \in V$
- This last property is called the *triangle inequality*

Algorithms on Graphs

- What are the basic operations we need to describe algorithms on graphs?
 - Given vertices u and v : are they *adjacent*?
 - Given vertex v and edge e , are they *incident*?
 - Given an edge e , get its incident vertices (*ends*)
 - How many vertices are adjacent to v ? (*degree* of v)
 - The vertices adjacent to v are called its *neighbors*
 - Get a list of the vertices *adjacent* to v
 - From which we can get the edges *incident* with v

Basic Graph Algorithms

- We'll look at a number of graph algorithms
 - Connectedness: Is G connected?
 - If not, how many connected components does G have?
 - Cycle testing: Does G contain a cycle?
 - Does G contain a cycle through a given vertex?
 - If the edges of G have costs:
 - What is the cheapest connected subgraph of G that contains every vertex?
 - What is a cheapest path from u to v ?
 - And more....

Testing Connectedness

- How can we determine whether G is connected?
 - Pick a vertex v ; see if every vertex u is reachable from v
- How could we do this?
 - Visit the neighbors of v , then visit their neighbors, etc. See if you reach all vertices
 - Assume we can mark a vertex as “visited”
- How do we *efficiently* manage all this visiting?

Reachability: Breadth-First Search

```
BFS(G, v) // Do a breadth-first search of G starting at v  
// pre: all vertices are marked as unvisited  
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;
```

Now compare value returned from $\text{BFS}(G,v)$ to size of V

BFS Theorem

Thm. $\text{BFS}(G,v)$ visits exactly those vertices u reachable from v .

Proof: We'll show that if u is reachable from v then $\text{BFS}(G,v)$ visits u by induction on $d = d(v,u)$

- Base Case: $d = 0$. Then $u = v$.
 - v is reachable from v and $\text{BFS}(G,v)$ visits v
- Induction Hypothesis: For some $d \geq 0$, if $d(u,v) = d$ then $\text{BFS}(G,v)$ visits u .

BFS Theorem

- Induction Step: Assume now that $d(u,v) = d+1$
 - Let $v = v_0, e_1, v_1, e_2, v_2, \dots, v_d, e_{d+1}, v_{d+1} = u$ be a path of length $d+1$ from v to u
 - Then $v = v_0, e_1, v_1, e_2, v_2, \dots, v_d$ is a path of length d from v to v_d
 - By I.H., v_d is visited by $\text{BFS}(G,v)$ and put in Q
 - So v_d will be dequeued and all of its unvisited neighbors, including u , will be marked as visited

A similar argument shows that if u is visited by $\text{BFS}(G,v)$ then u is reachable from v

BFS Reflections

- The BFS algorithm traced out a tree T_v : the edges connecting a visited vertex to (as yet) unvisited neighbors
- T_v is called a *BFS tree of G with root v (or from v)*
- The vertices of T_v are visited in *level-order*
- Every path in T_v from v to a vertex u is a *shortest possible path* from v to u
 - That is, the path has length $d(v,u)$