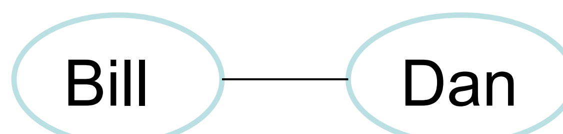
CSCI 136
Data Structures & Advanced Programming

Lecture 30

Spring 2020

Instructors:



Last Time

- Hashing applications
  - Cuckoo hashtables
  - Bloom filters
  - Data Verification
  - Data Deduplication
- Hashing is a powerful tool that can be applied in order to solve many problems.

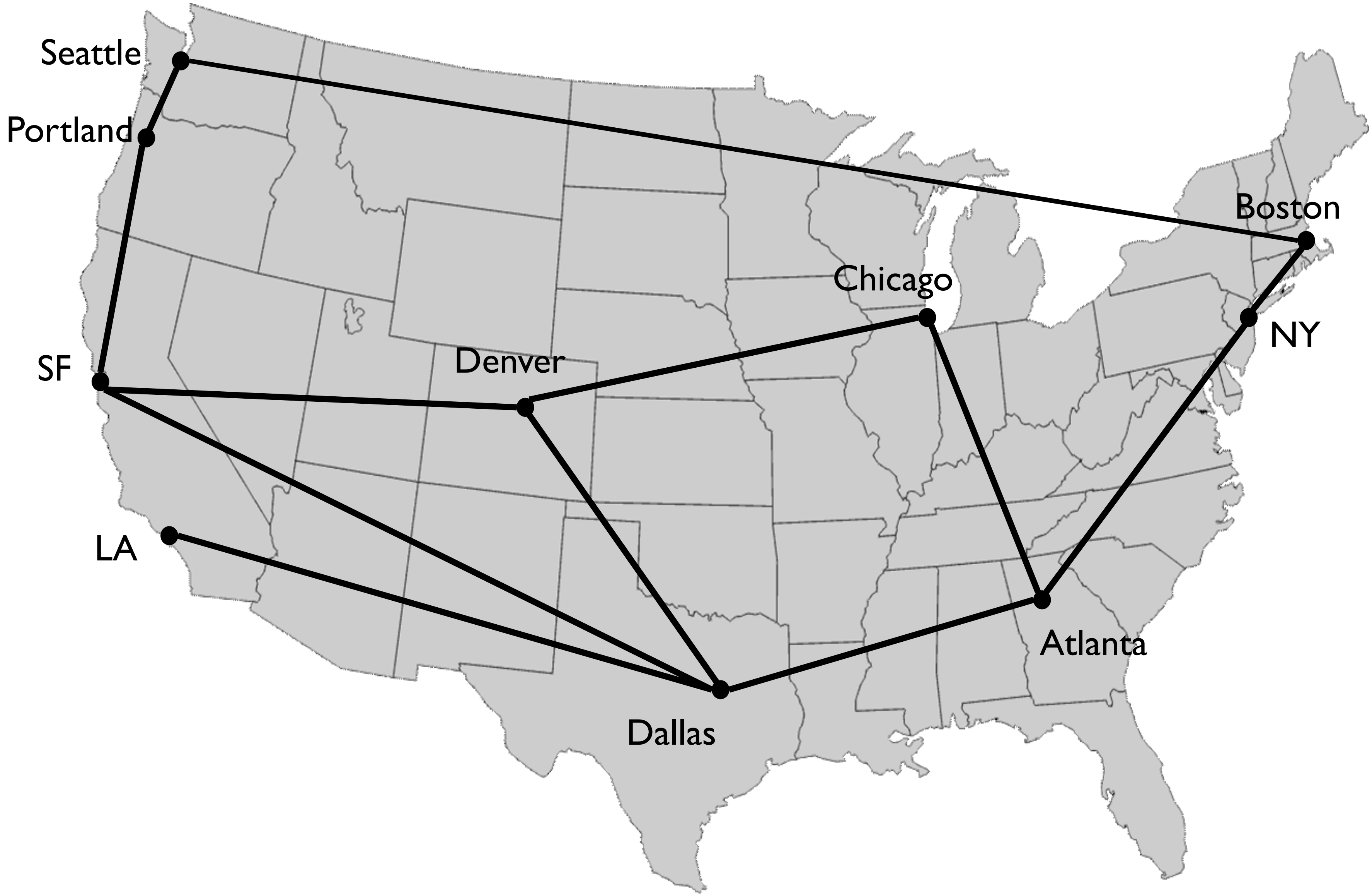
Today's Outline

- Introduction To Graphs
  - Definitions and Properties: Undirected Graphs
  - Small Proofs
  - Rechability
  - Graph Interface in Structure5

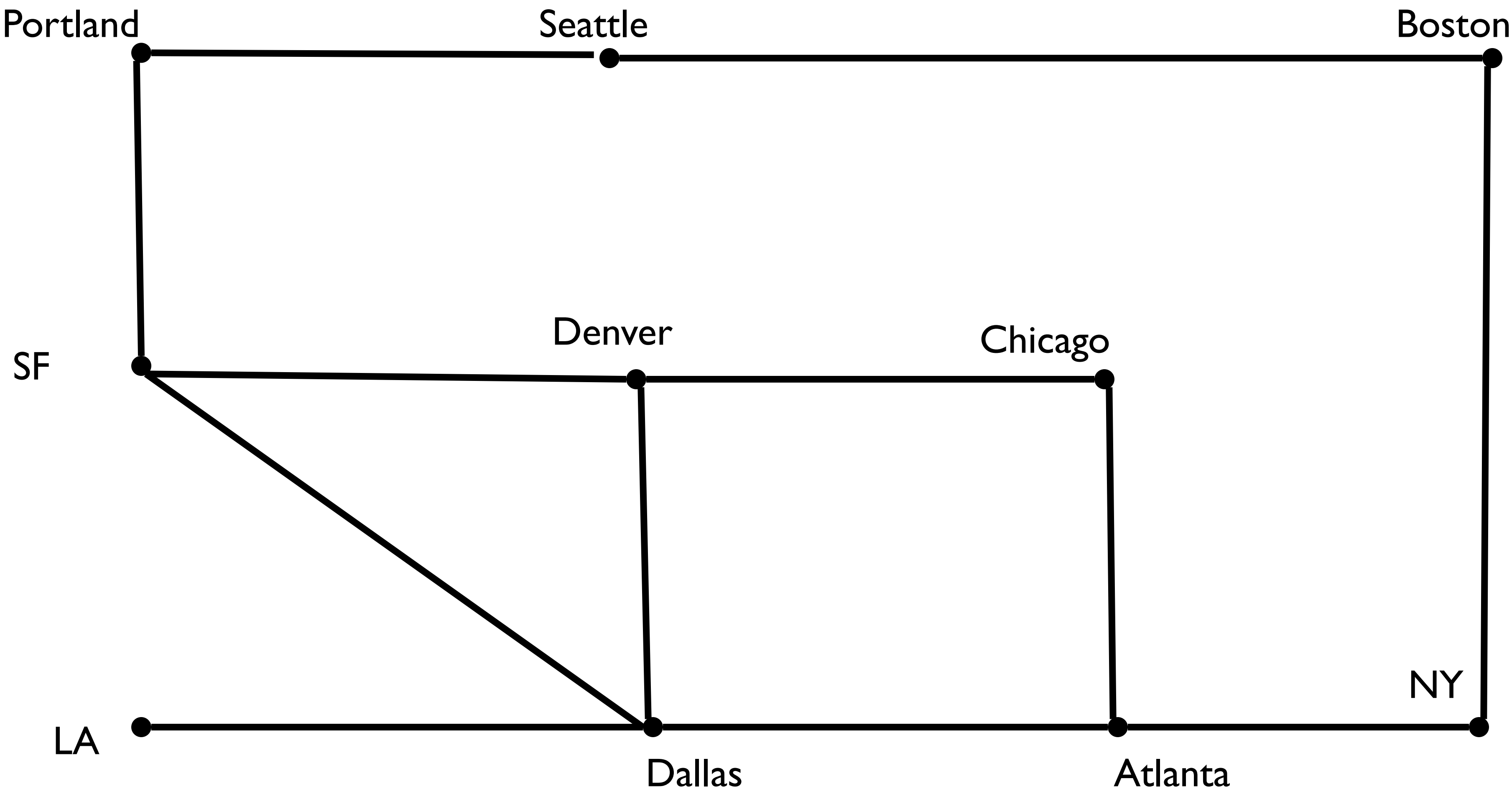
Graphs Describe the World[1]

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

Nodes = subway stops; Edges = subway lines

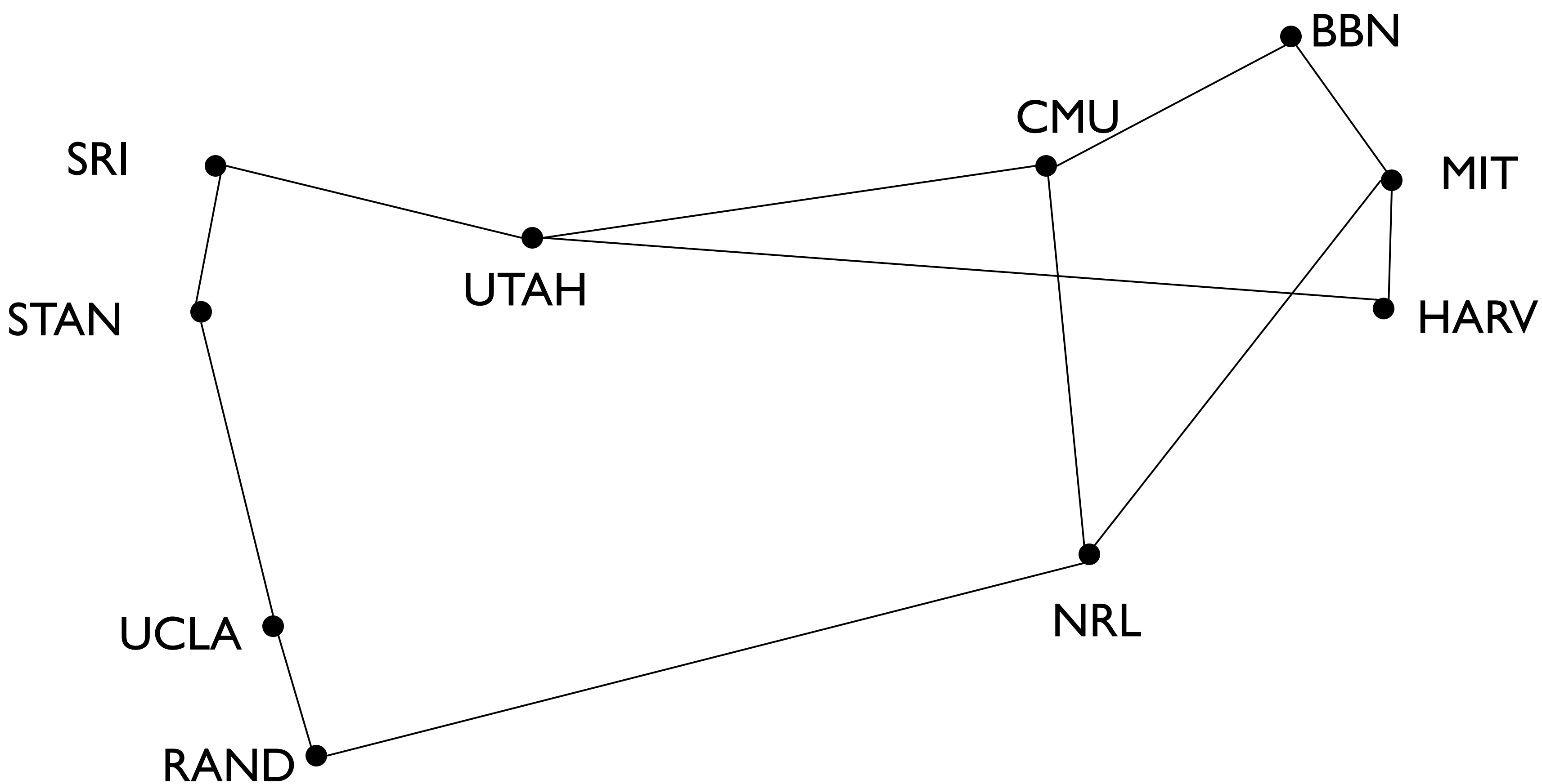


Nodes = cities; Edges = rail lines connecting cities

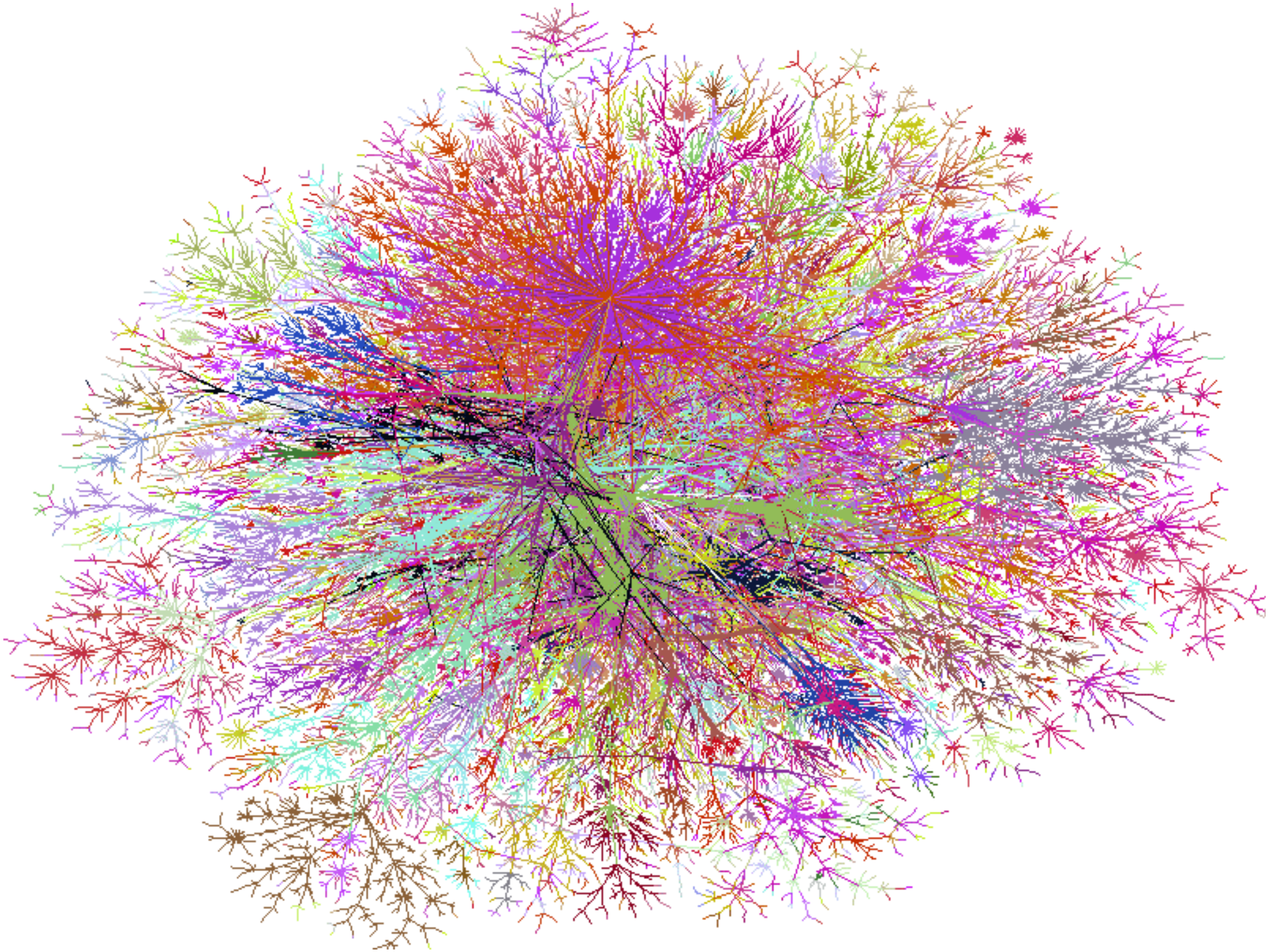


Note: Connections in graph matter, not precise locations of nodes

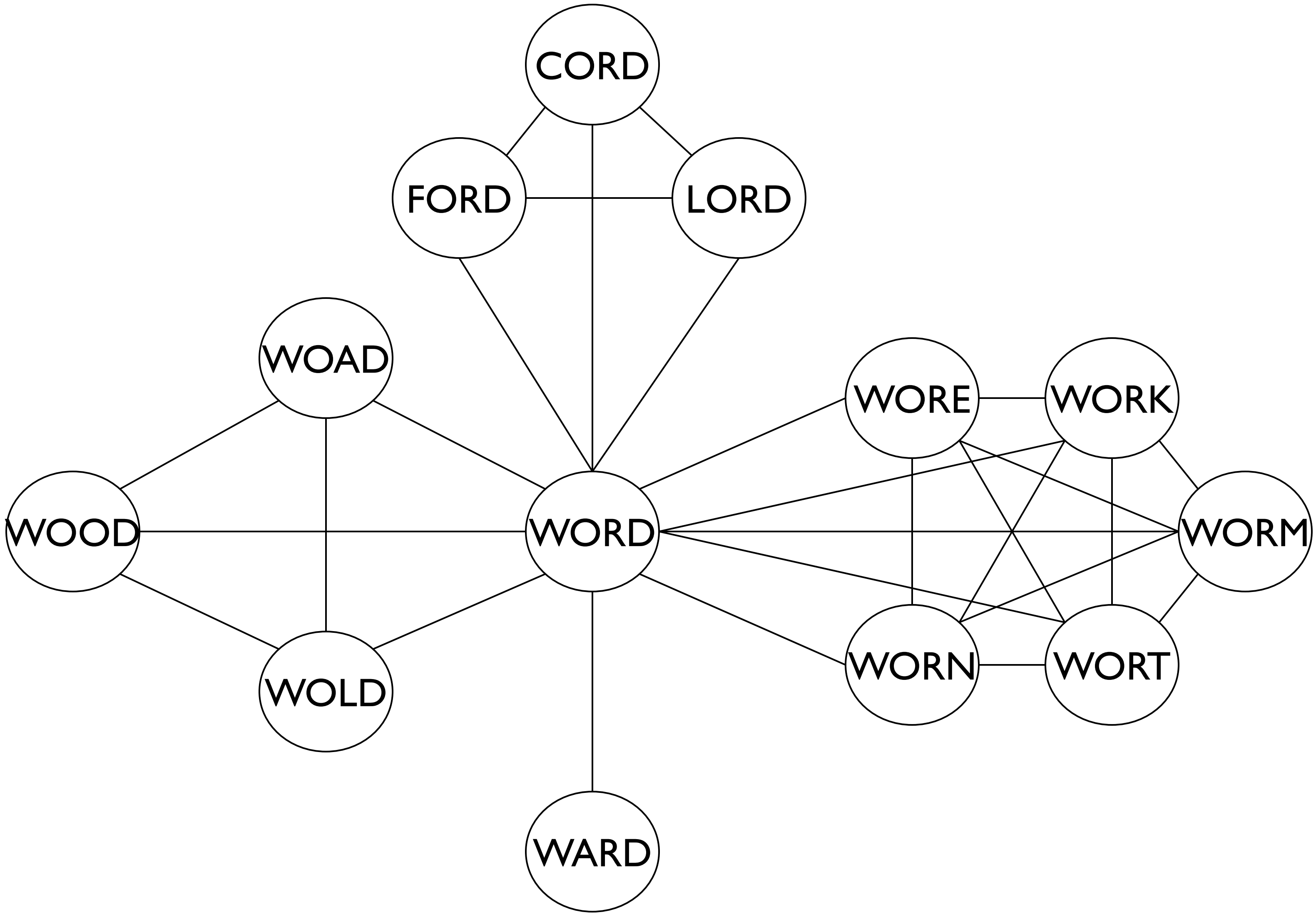
Internet (~1972)



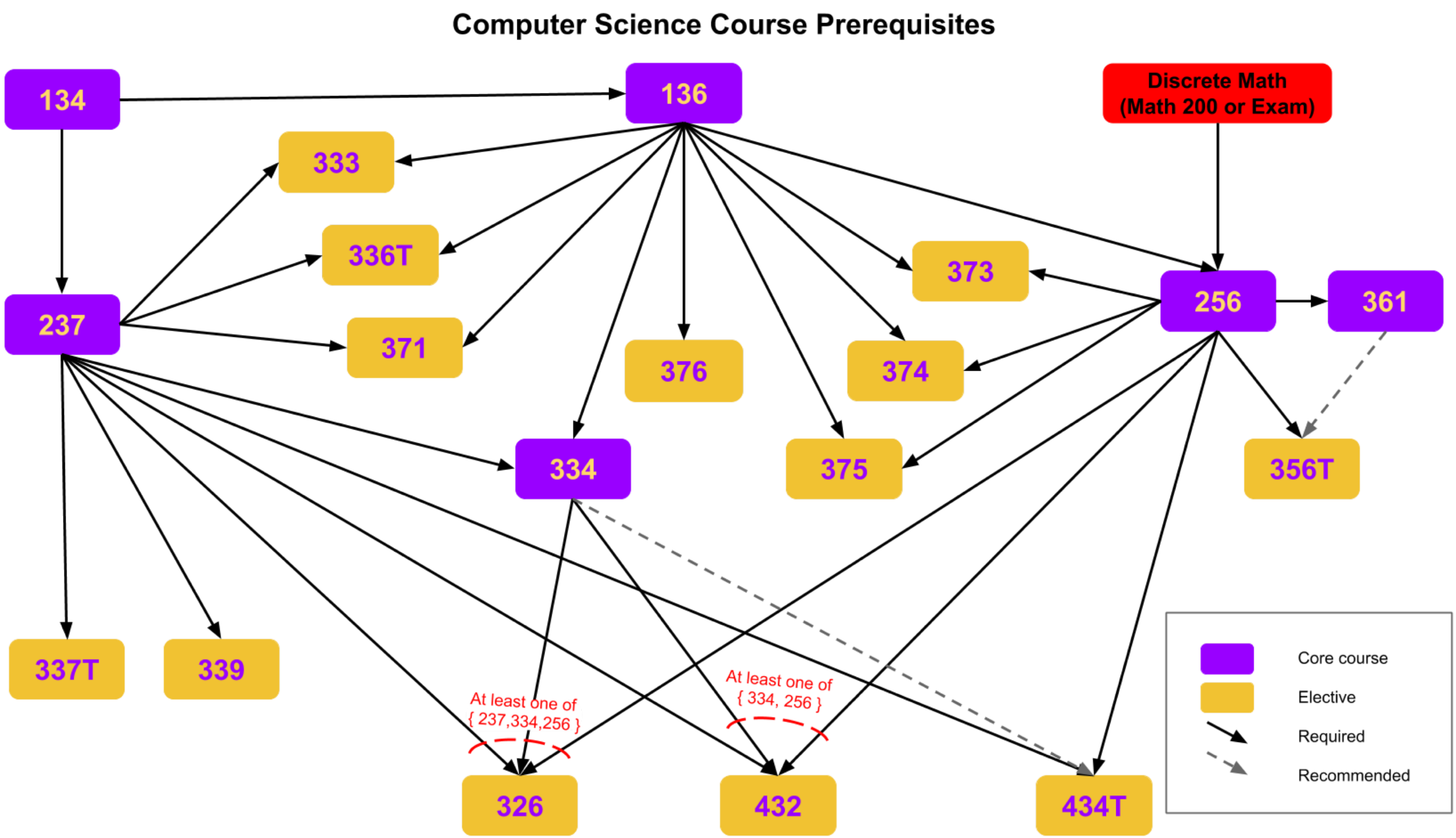
Internet (~1998)



Word Game

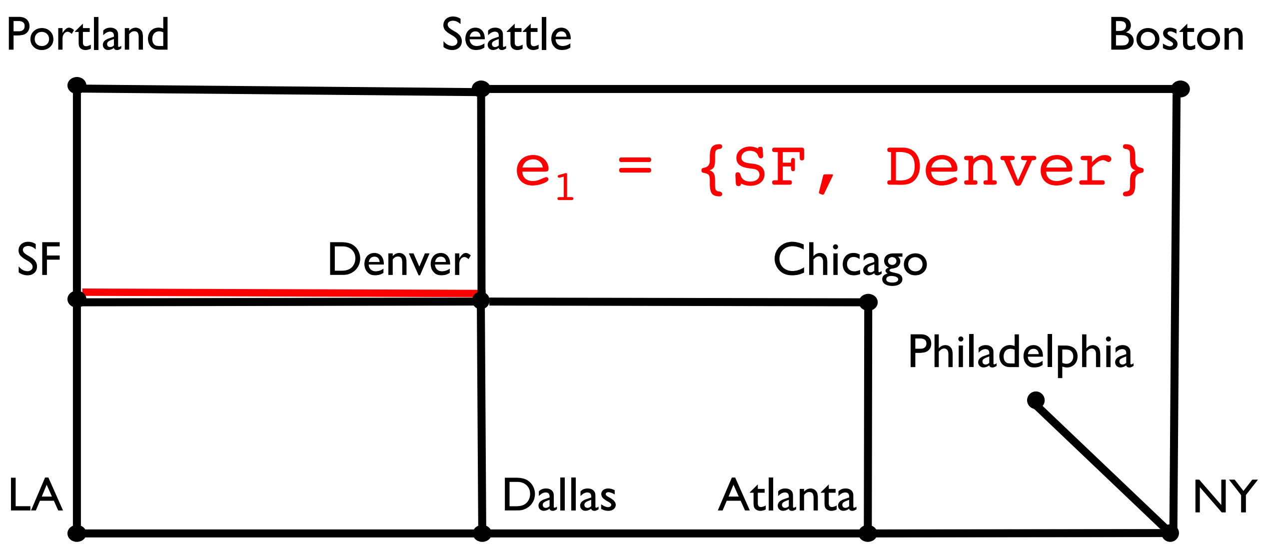


Nodes = words; Edges = words that differ by exactly one letter



Nodes = courses; Edges = prerequisites ***

Basic Definitions & Concepts



Definition: 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.*

Basic Definitions & Concepts

- **Definition:** An *undirected graph* `G = (V,E)` consists of two sets:
  - `V` : the *vertices* of `G`
  - `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*
- The *degree* of a vertex is the number of *incident edges* (loops counted twice)

Walking Along a Graph

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

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

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

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

  such that $v_0 = v_k$ (it ends at the starting v)
- A *circuit* is a *path* where $v_0 = v_k$
  - Circuit vs. closed walk? Circuit has no repeat edges
- A *cycle* is a *simple* path where $v_0 = v_k$
  - Circuit vs. cycle? Cycle has no repeated vertices.
- 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!]

See Handout

- We give example graph of rail network from earlier in slides
  - Task: Define each term, then give examples from the graph
- Also provided sample solutions to check against for practice

Graphs in Structure5

- Implementation involves a number of design decisions, depending on intended uses
  - What kinds of graphs will be available?
    - Undirected, directed, mixed
  - What underlying data structures will be used?
  - What functionality will be provided?
  - What aspects will be public/protected/private
- We'll focus on popular implementations for undirected and directed graphs (separately)

Graphs in structure5

- Please refer to the graph interface handout as you follow along with the rest of this recording
- If you can, make annotations on the PDF or print out a copy to take notes

Graphs in structure5

- We want to store information at vertices and at edges, but we will favor vertices
  - Let V and E represent the types of information held by vertices and edges respectively
  - Interface Graph<V,E> extends Structure<V>
    - Vertices are the building blocks; edges depend on them
- Type V holds a *label* for a (hidden) vertex
- Type E holds a *label* for an (available) edge
  - Label?: Application-specific data for a vertex/edge

Graphs in structure5

- So, the methods described in the Structure interface are about vertices (but also impact edges: e.g., `clear()`)
- We'll want to add a number of similar methods to provide information about edges, and the graph itself
  - Ultimately the `Structure` interface is a subset of the total functionality in the graph classes

What is the Desired Functionality

- What are the basic operations we need in order 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`? (*`deg(v)`*)
    - The vertices adjacent to `v` are called its *neighbors*
  - Get a list of the neighbors of `v` (or the edges incident with `v`)

Graph Interface Methods

- void add(V vLabel), V remove(V vLabel)
  - Add/remove vertex to graph
- void addEdge(V vLabel1, V vLabel2, E edgeLabel),
  E removeEdge(V vLabel1, V vLabel2)
  - Add/remove edge between vLabel1 and vLabel2
- boolean containsEdge(V vLabel1, V vLabel2)
  - Returns true iff there is an edge between vLabel1 and vLabel2
- Edge<V,E> getEdge(V vLabel1, V vLabel2)
  - Returns edge between vLabel1 and vLabel2
- void clear()
  - Remove all nodes (and edges) from graph

Graph Interface Methods

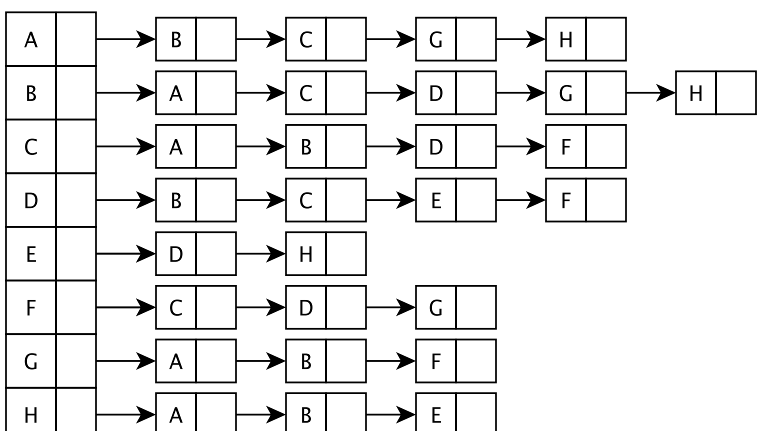
- boolean visit(V vLabel)
  - Mark vertex as "visited" and return *previous* value of visited flag
- boolean visitEdge(Edge<V,E> e)
  - Mark edge as "visited"
- boolean isVisited(V vLabel), boolean isVisitedEdge(Edge<V,E> e)
  - Returns true iff vertex/edge has been visited
- Iterator<V> neighbors(V vLabel)
  - Get iterator for all neighbors of vLabel
  - For directed graphs, out-edges only
- Iterator<V> iterator()
  - Get vertex iterator
- void reset()
  - Remove visited flags for all nodes/edges

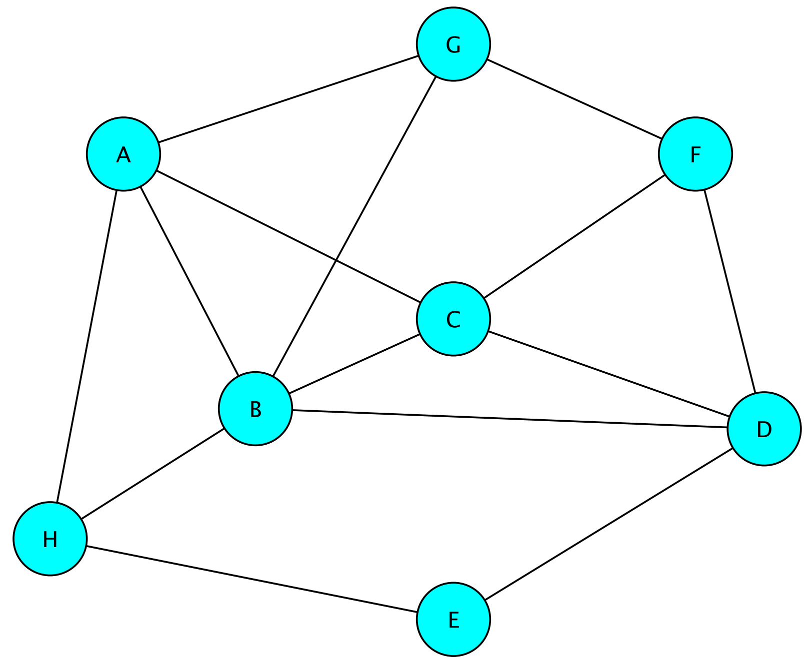
Representing Graphs

- Two standard approaches
  - Option 1: Array-based (directed and undirected)
  - Option 2: List-based (directed and undirected)
- We'll look at both
  - Array-based graphs store the edge information in a 2-dimensional array indexed by the vertices
  - List-based graphs store the edge information in a (1-dimensional) array of lists
    - The array is indexed by the vertices
    - Each array element is a list of edges incident with that vertex

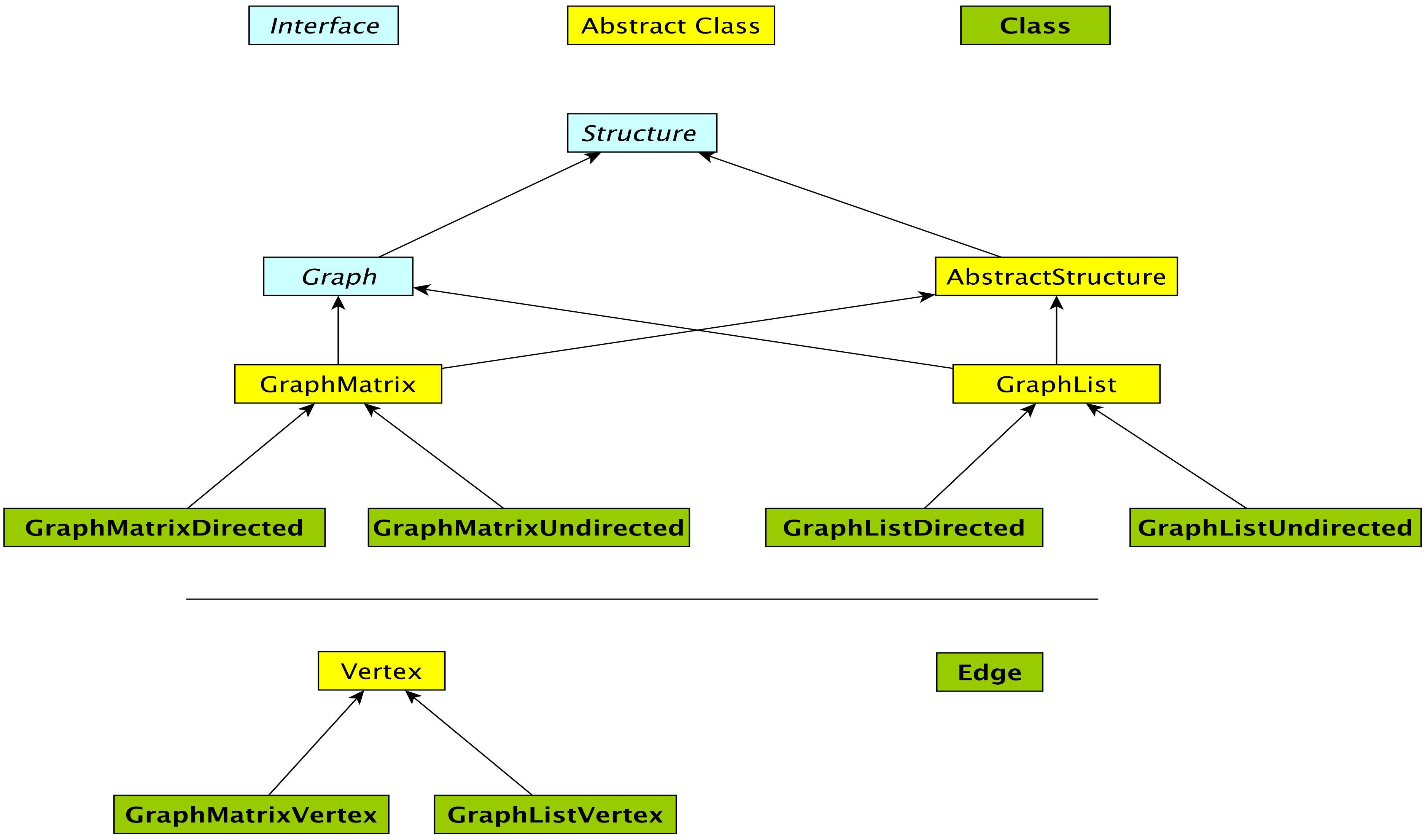
Example Graph Representations: Lists and Matrices

	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	1	0	1	1	0	0	1	1
C	1	1	0	1	0	1	0	0
D	0	1	1	0	1	1	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	1	0
G	1	1	0	0	0	1	0	0
H	1	1	0	0	1	0	0	0





Graph Classes in structure5



Edge Class

- Graph *edges* are defined in their own public class (*vertices* are hidden: referenced only by their label)
  - `Edge<V,E>(V vLabel1, V vLabel2,`
    `E label, boolean directed)`
  - Construct a (possibly directed) edge between two labeled vertices (`vLabel1` → `vLabel2`)
  - vLabel1 : here; vLabel2 : there
- Useful `Edge` methods (getters and setters):

```
label(), here(), there()
setLabel(), isVisited(), isDirected()
```

Reachability and Connectedness

- **Definition:** 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`
- **Definition:** An undirected graph G is *connected* if for every pair of vertices (`u`, `v`) in G, `v` is reachable from `u` (and vice versa)
- 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`

Reachability: Breadth-First Search

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

How does this translate to code?

Breadth-First Search

```
int BFS(Graph<V,E> g, V src) {
  int count = 0; Queue<V> todo = new QueueList<V>();
  todo.enqueue(src);
  g.visit(src); count++;
  while (!todo.isEmpty()) {
    V vertex = todo.dequeue();
    Iterator<V> neighbors = g.neighbors(vertex);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next)) {
          todo.enqueue(next);
          g.visit(next); count++;
       }
    }
  }
  return count;
}
```

Breadth-First Search of Edges

```
int BFS(Graph<V,E> g, V src) {
  int count = 0; Queue<V> todo = new QueueList<V>();
  todo.enqueue(src);
  g.visit(src); count++;
  while (!todo.isEmpty()) {
    V vertex = todo.dequeue();
    Iterator<V> neighbors = g.neighbors(vertex);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisitedEdge(vertex, next))
              g.visitEdge(vertex, next);
       if (!g.isVisited(next)) {
          todo.enqueue(next);
          g.visit(next); count++;
       }
    }
  }
  return count;
}
```

Recursive Depth-First Search

```
// Before first call to DFS, set all vertices to unvisited
//Then call DFS(G,v)
DFS(G, v)
        Mark v as visited; count=1;
        for each unvisited neighbor u of v:
                count += DFS(G,u);
        return count;
```

How does this translate to code?

Recursive Depth-First Search

```
int depthFirstSearch(Graph<V,E> g, V src) {
    g.visit(src);
    int count = 1;
    Iterator<V> neighbors = g.neighbors(src);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next))
              count += depthFirstSearch(g, next);
    }
  return count;
}
```

Next Class

- This was a lot of definitions and jargon
- Next class we will look at 2 concrete designs: an adjacency list and an adjacency matrix
  - How would you implement them?
  - What is their performance?
  - In what types of situations would you choose one design over the other?