CSCI 136
Data Structures & Advanced Programming

Lecture 27

Fall 2019

Instructors: Bill — Sam

Last Time

- Lab 9: Super Lexicon!
- AVL trees

Today's Outline

- Red-black trees
- Graphs

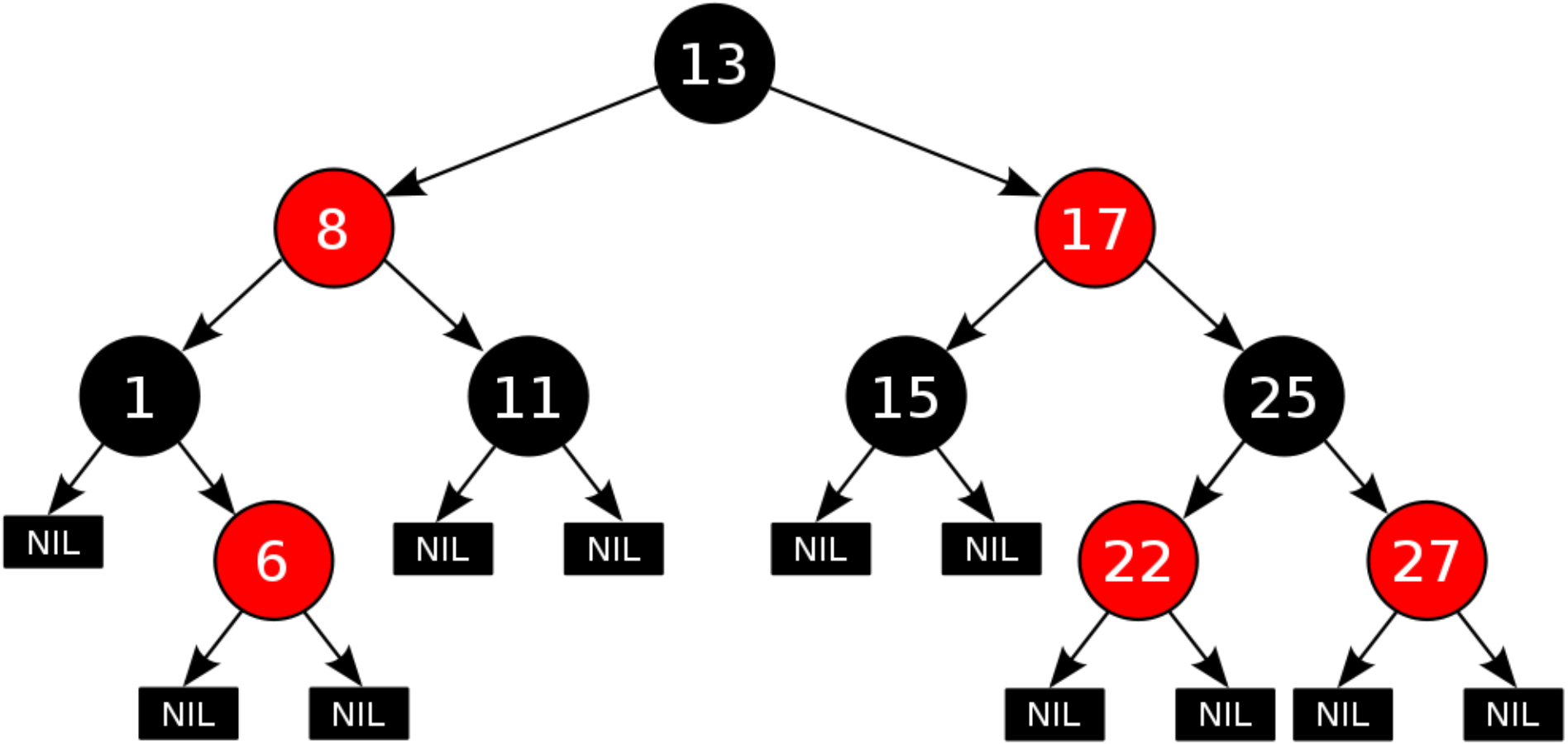
Red-Black Trees

Red-Black trees, like AVL, guarantee shallowness

- Each node is colored *red* or *black*
- Coloring satisfies these rules
  - All empty trees are black
    - We consider them to be the leaves of the tree
  - Children of red nodes are black
  - All paths from a given node to it's descendent leaves have the *same number* of black nodes
    - This is called the *black height* of the node

A Red-Black Tree

(from Wikipedia.org)



Red-Black Trees

The coloring rules lead to the following result

Proposition: No leaf has depth more than twice that of any other leaf.

This in turn can be used to show

Theorem: A Red-Black tree with n internal nodes has height satisfying $h \leq 2\log(n+1)$

- Note: The tree will have *exactly* n+1 (empty) leaves
  - since each internal node has two children

Red-Black Trees

Theorem: A Red-Black tree with n *internal* nodes has height satisfying $h \leq 2\log(n+1)$

Proof sketch: Note: we count empty tree nodes!

- If root is red, recolor it black.
- Now merge red children into (black) parents
  - Now n' ≤ n nodes and height h' ≥ h/2
- New tree has all children with degree 2, 3, or 4
  - All leaves have depth *exactly* h' and there are n+1 leaves
    - So $n+1 \geq 2^{h'}$, so $\log_2(n+1) \geq h' \geq \frac{h}{2}$
- Thus $2\log_2(n+1) \geq h$

Corollary: R-B trees with n nodes have height O(log n)

Red-Black Trees: Insert

- Series of rules such as: "if my parent is red and my uncle is black and I am a right child of my parent, rotate me to the left. Then, rotate my parent to the right, color it black, and color its new children red."

- This works. We won't go over it in detail. The Wikipedia article on this is excellent
- Deletes are significantly worse

Splay Trees

Splay trees are self-adjusting binary trees

- Each time a node is accessed, it is moved to root position via rotations
- No metadata at all. Just rotate up each element you access
- 2 rules to rotate up; that's it

Splay Trees

Splay trees are self-adjusting binary trees

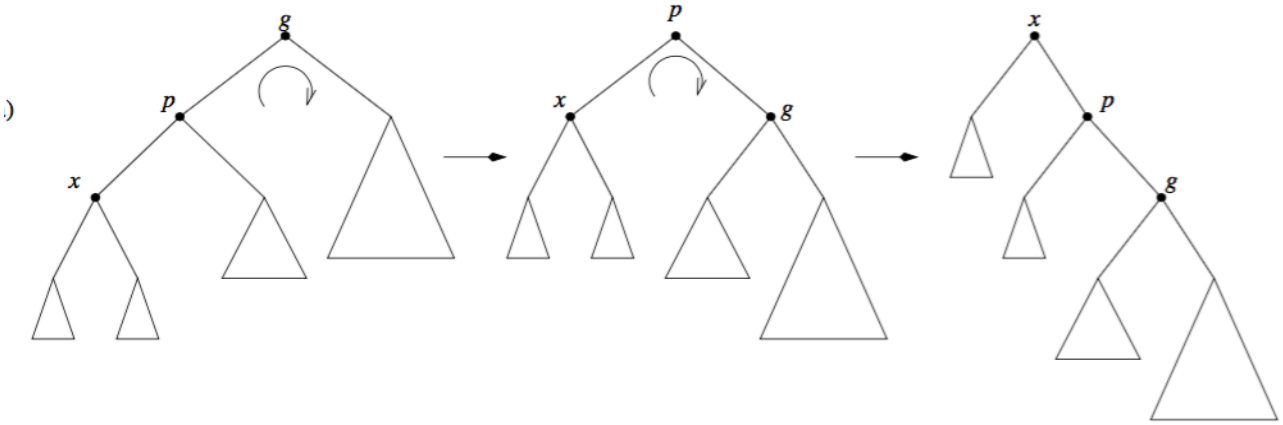
- Each time a node is accessed, it is moved to root position via rotations
- No guarantee of balance (or shallow height)
- But good *amortized* performance

Theorem: Any set of m operations (add, remove, contains, get) on an n-node splay tree take at most O(m log n) time.

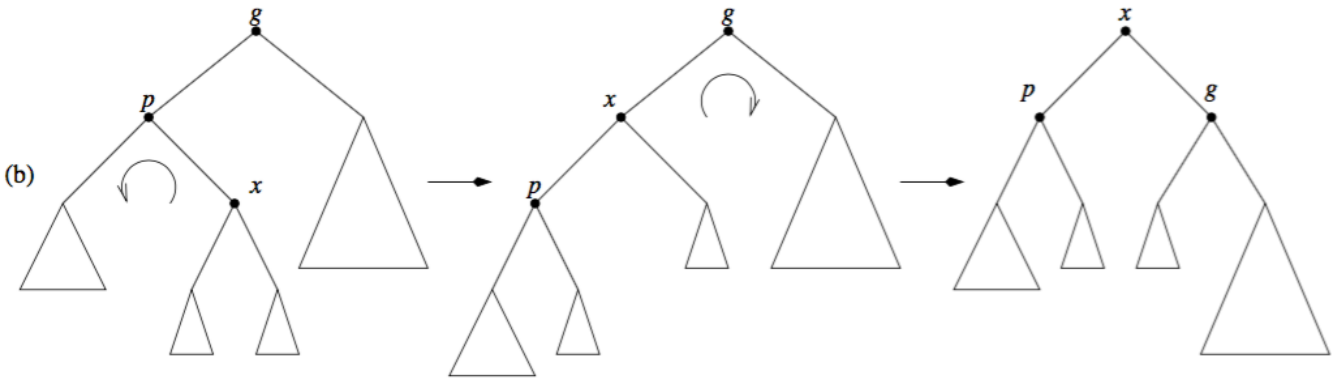
- "As good" as an AVL or Red-Black Tree!

Splay Tree Rotations

Right Zig-Zig Rotation (left version too)



Right Zig-Zag Rotation (left version too)



Dynamic Optimality

- Conjecture: For any sequence of access operations, if the best possible Binary Search Tree takes X operations, then a splay tree takes O(X) operations

- Essentially: keeping no metadata, and with no knowledge of the future, splay trees do as well as a perfect tree that knows the whole sequence in advance

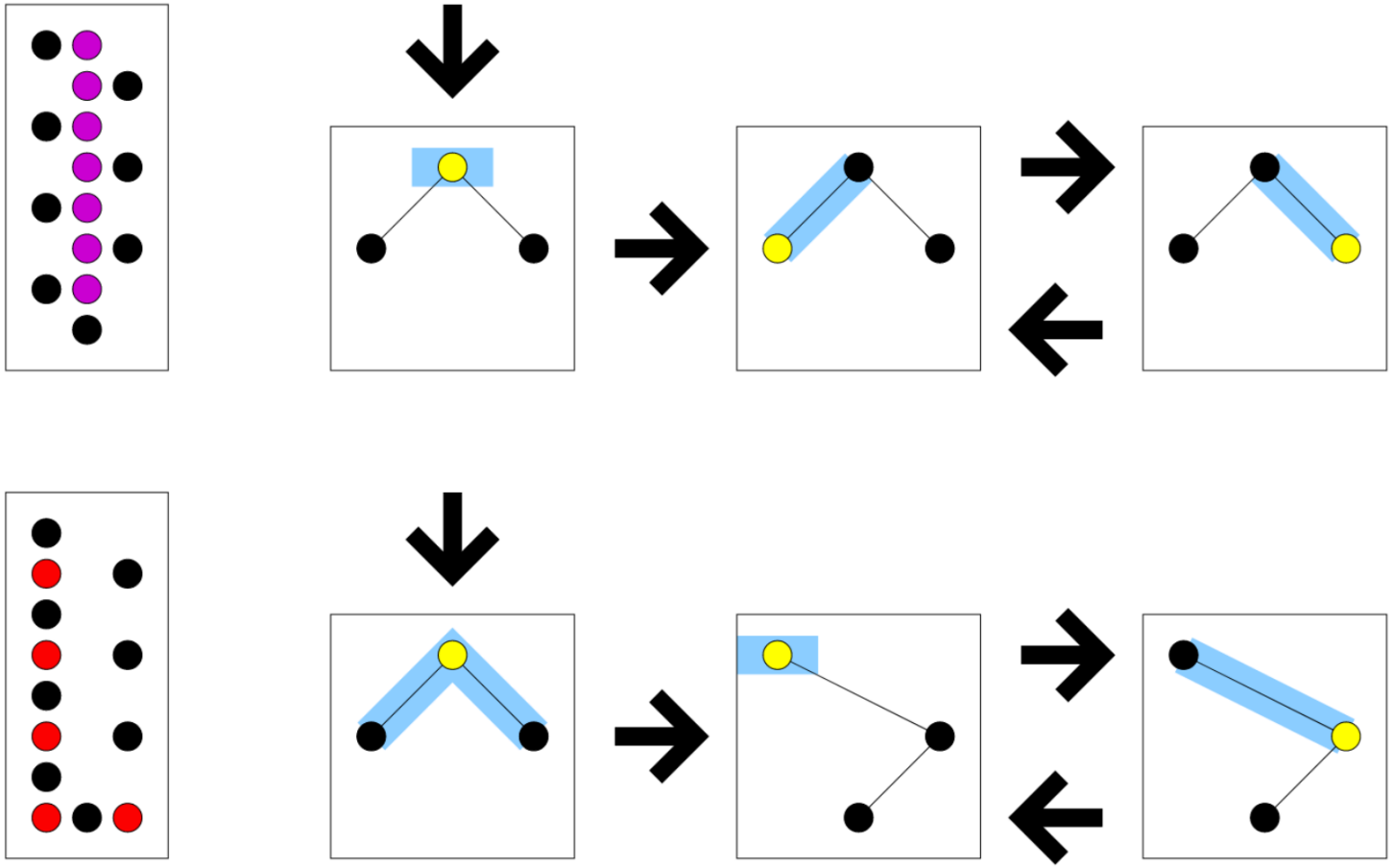
Dynamic Optimality

- Conjecture: For any sequence of access operations, if the best possible Binary Search Tree takes X operations, then a splay tree takes O(X) operations

- One consequence would be: splay trees can handle stack or queue operations in O(1) average operations like a DLL
- Recent progress by Levy and Tarjan in 2019

Dynamic Optimality

- Some really cool math in this area



Trees: What to Know

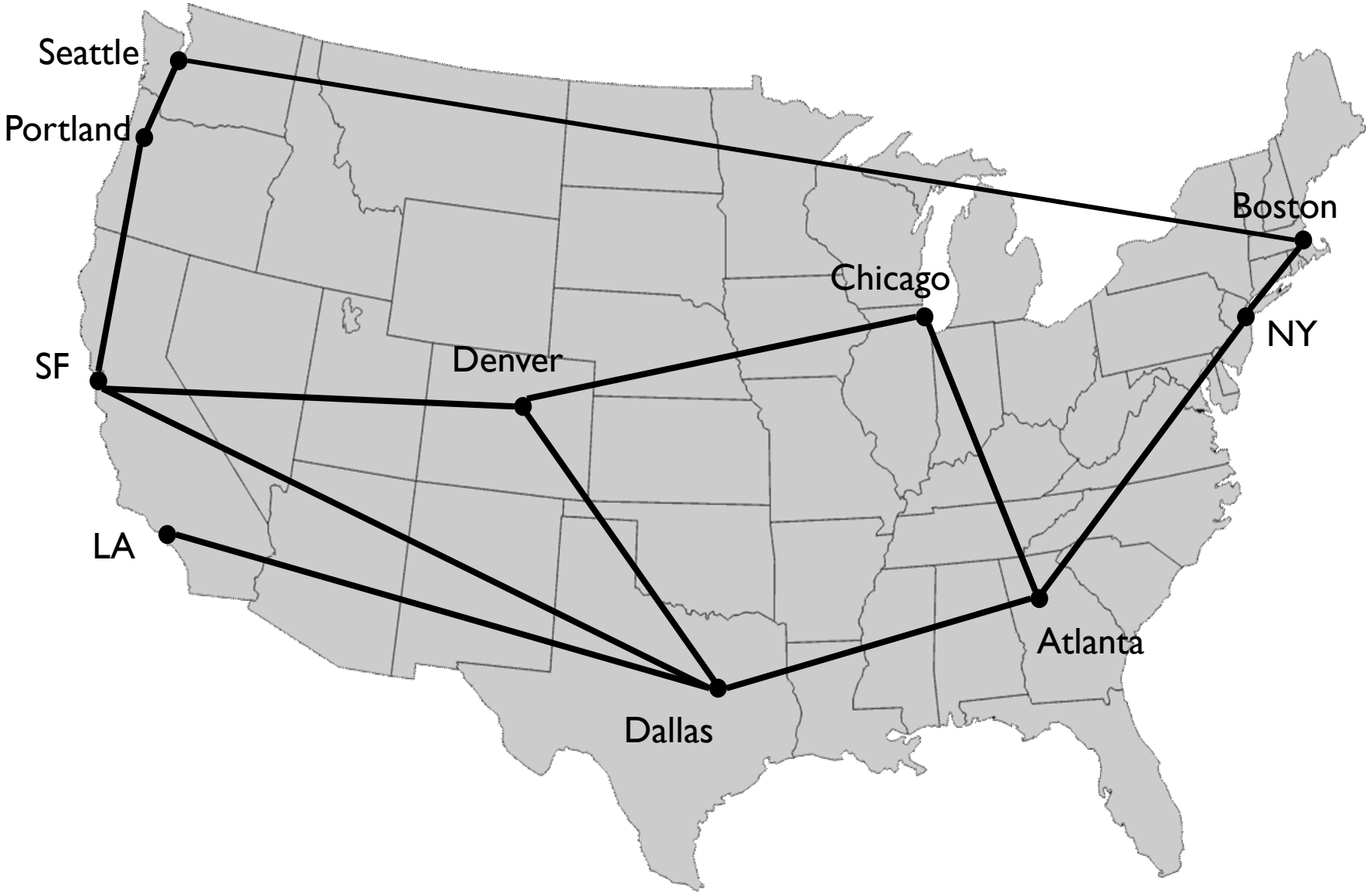
- Time for lookup, insert, delete in AVL tree? In Red-black tree?
- How do we adjust the structure of a tree?
  - Related: how do we move elements to maintain balance?

- How does an AVL tree maintain balance? What must it keep track of?
- How does a red-black tree maintain balance? What must it keep track of?

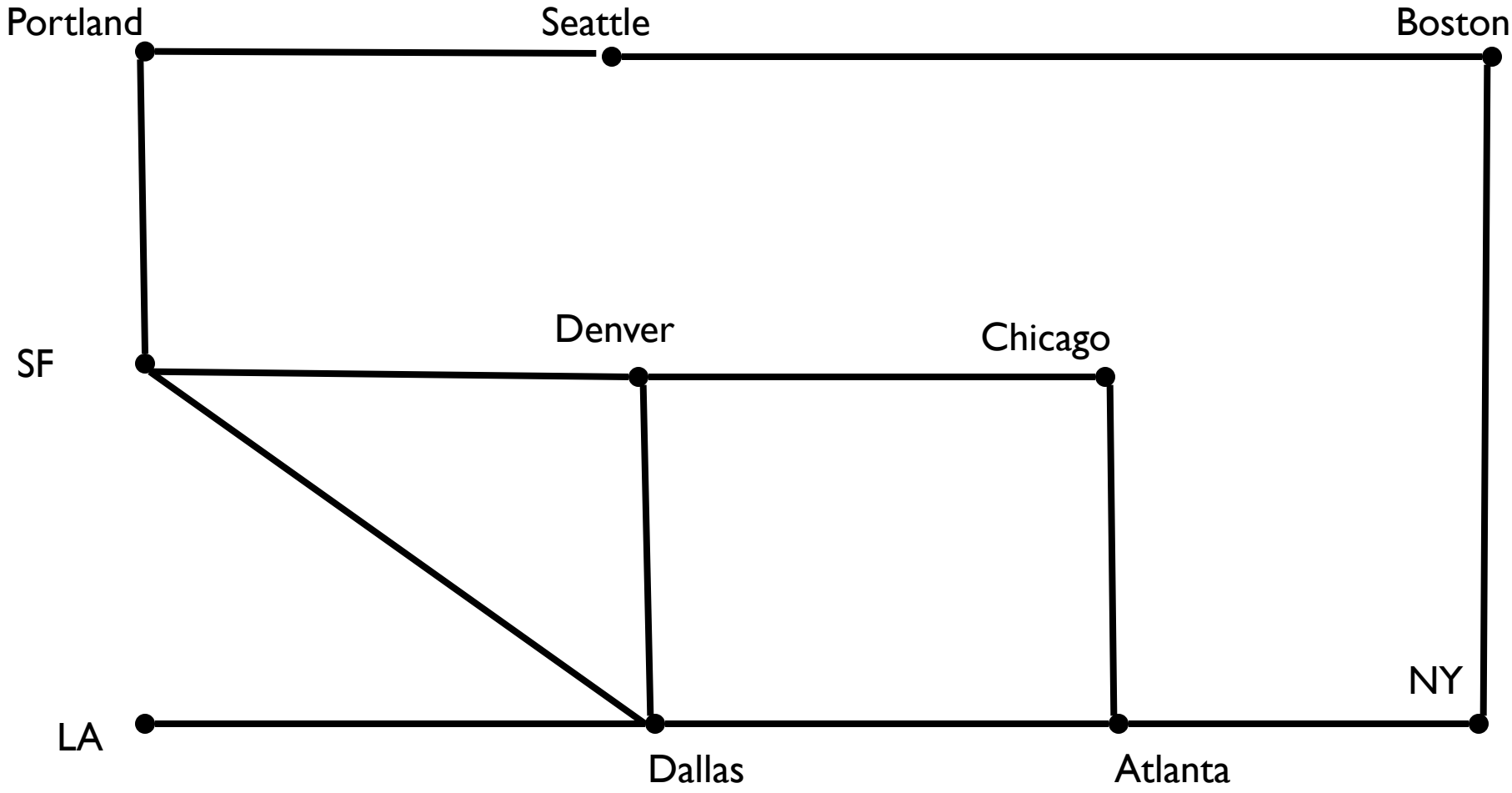
Graphs Describe the World

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

Nodes = subway stops; Edges = track between stops

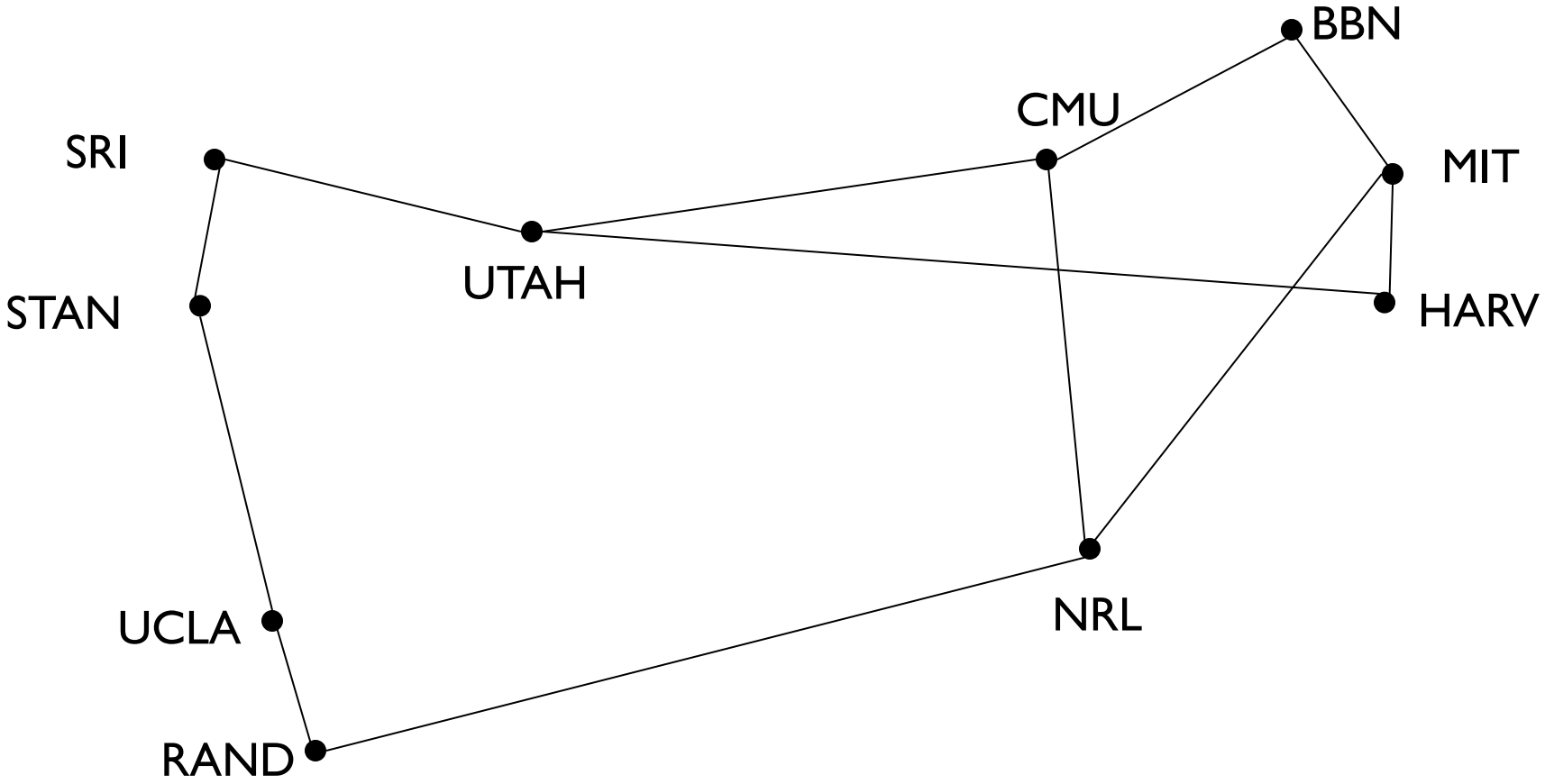


Nodes = cities; Edges = rail lines connecting cities

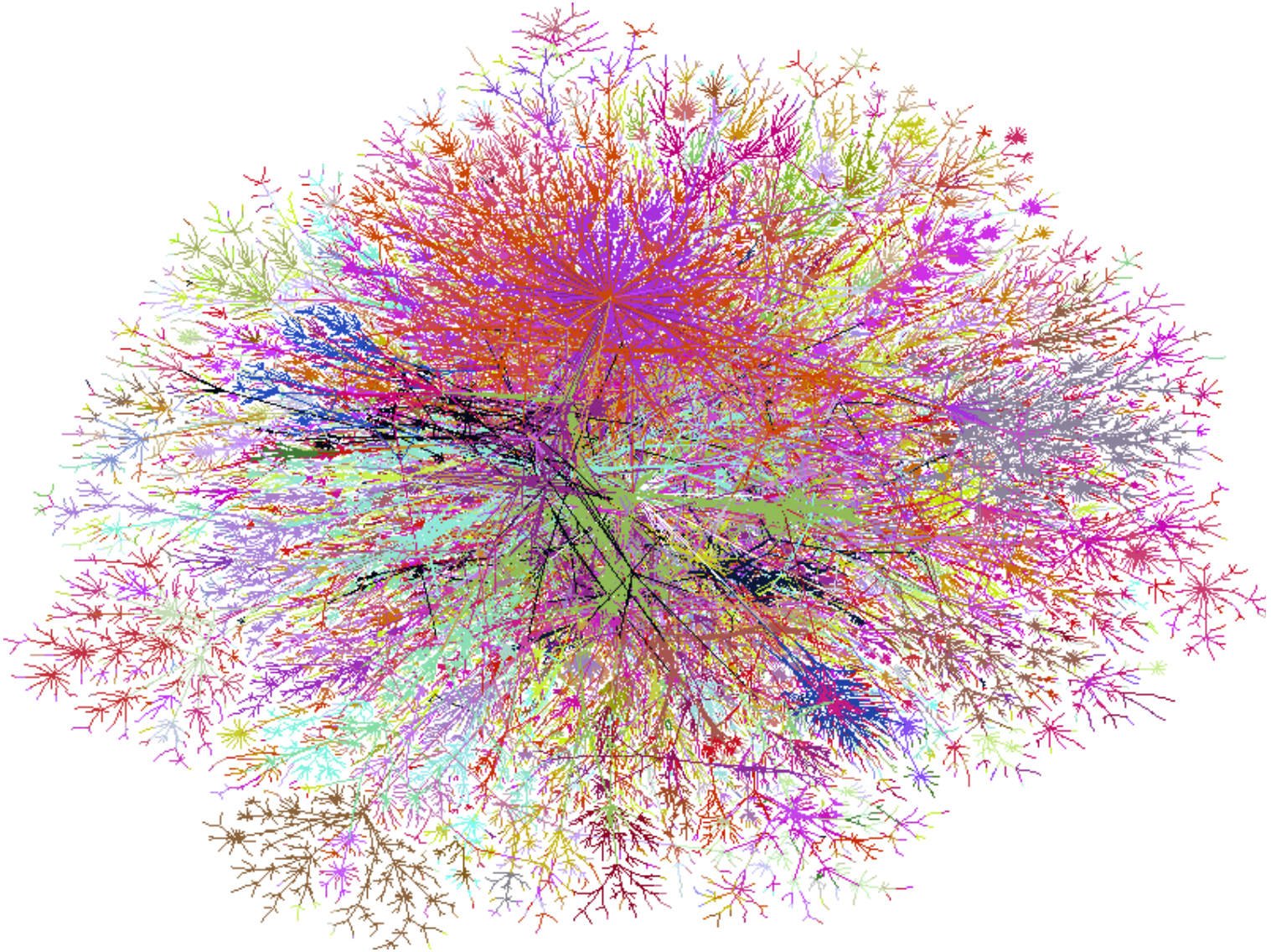


Note: Connections in graph matter, not precise locations of nodes

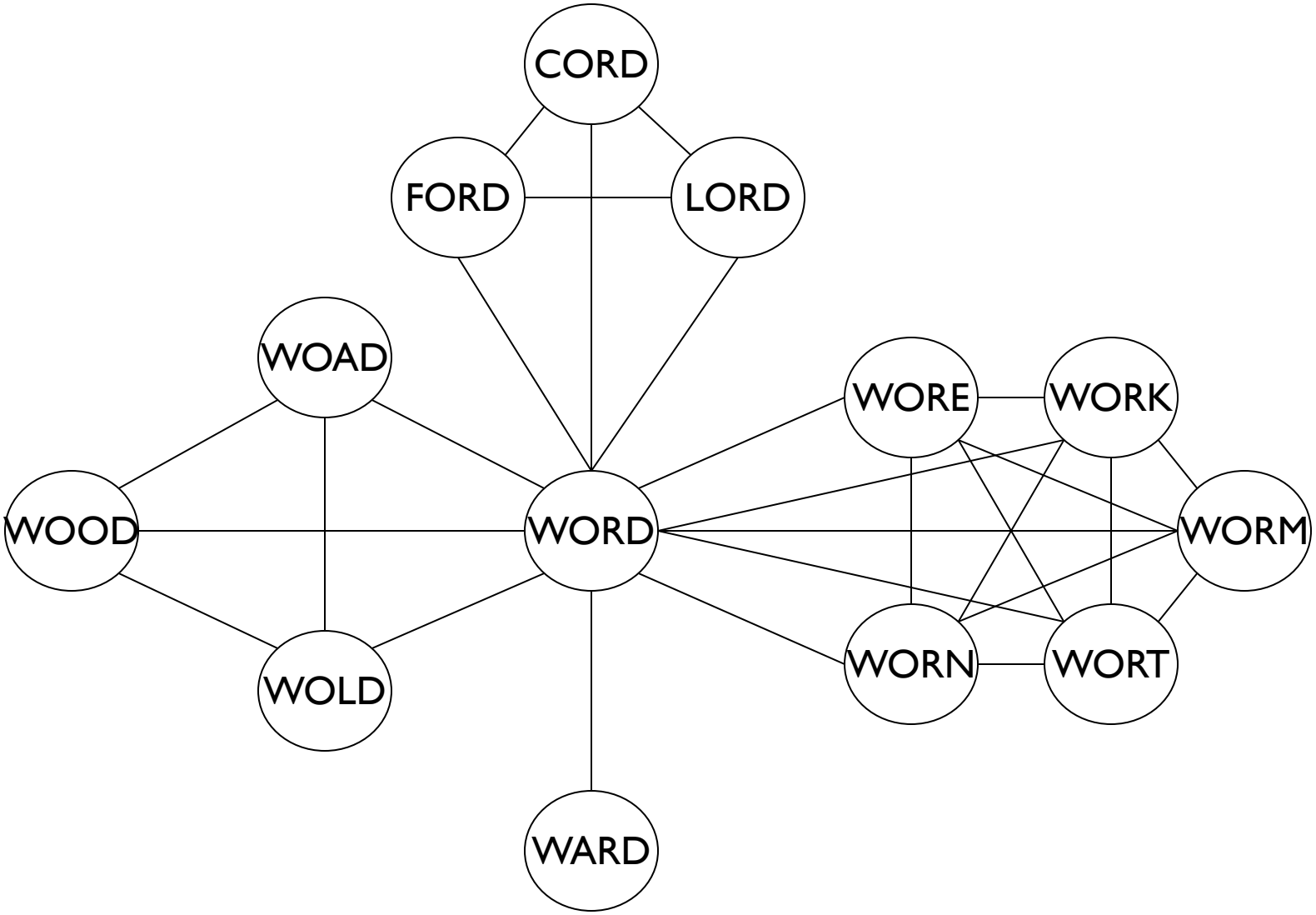
Internet (~1972)



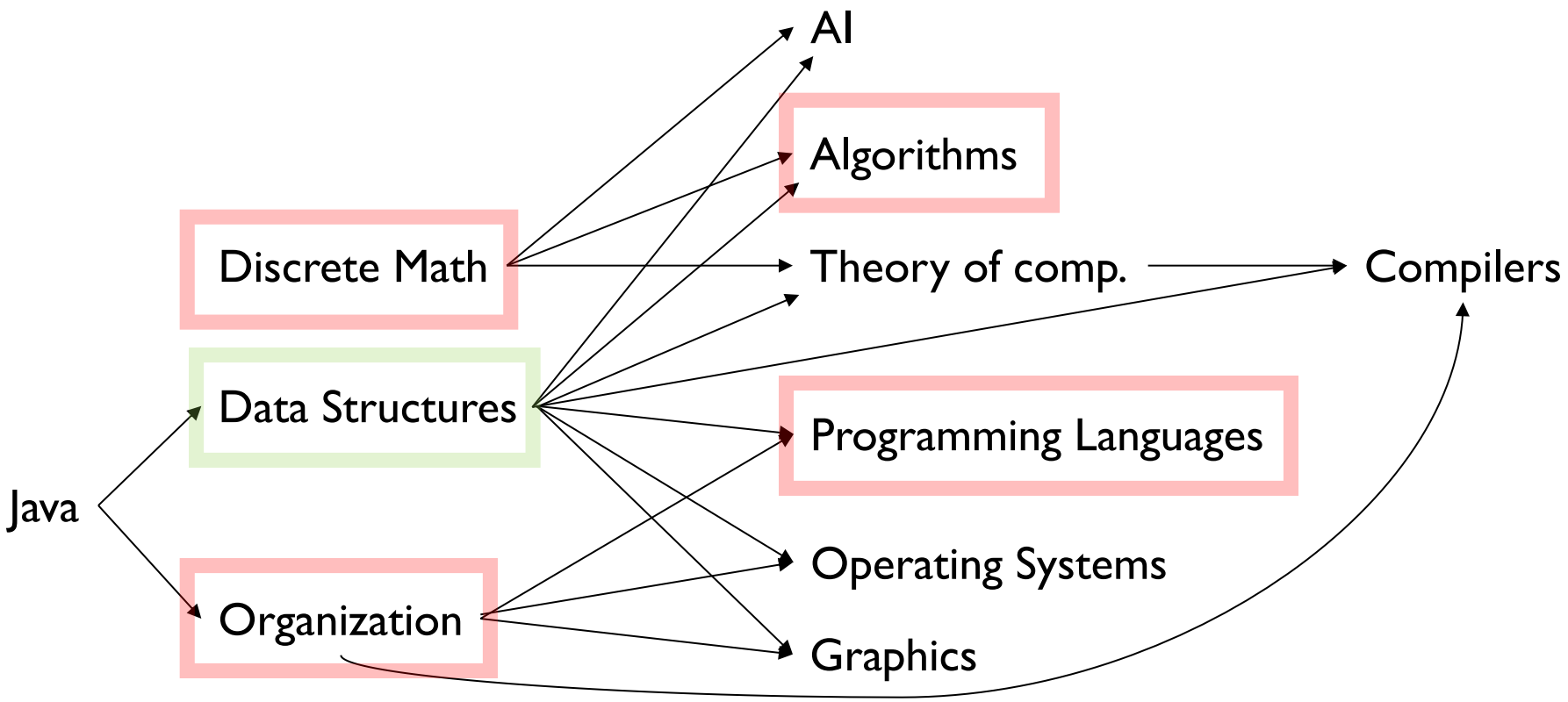
Internet (~1998)



Word Game

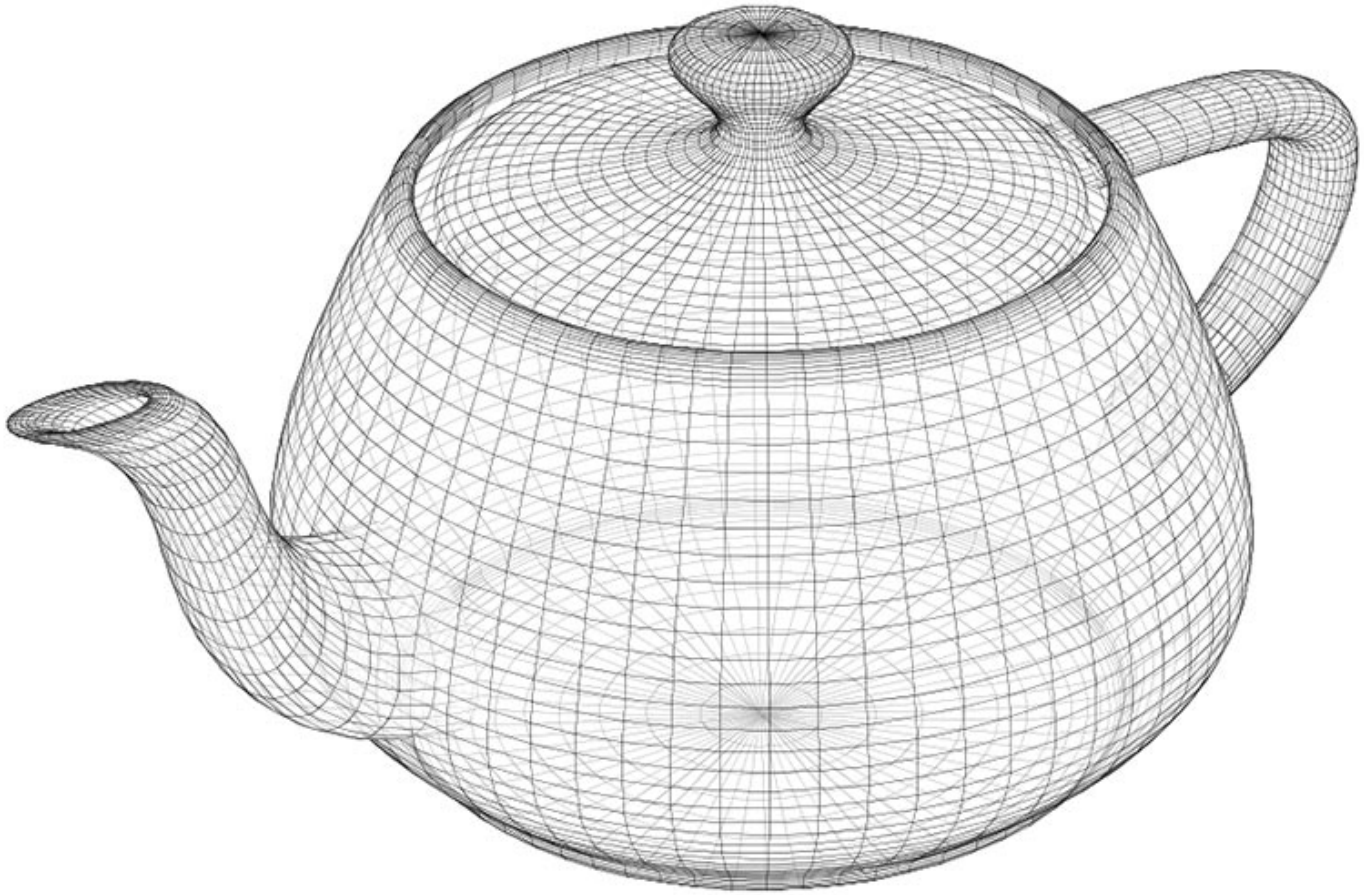

CORD
FORD
LORD
WOAD
WORE
WORK
WOOD
WORD
WORM
WOLD
WORN
WORT
WARD

CS Pre-requisite Structure (subset)

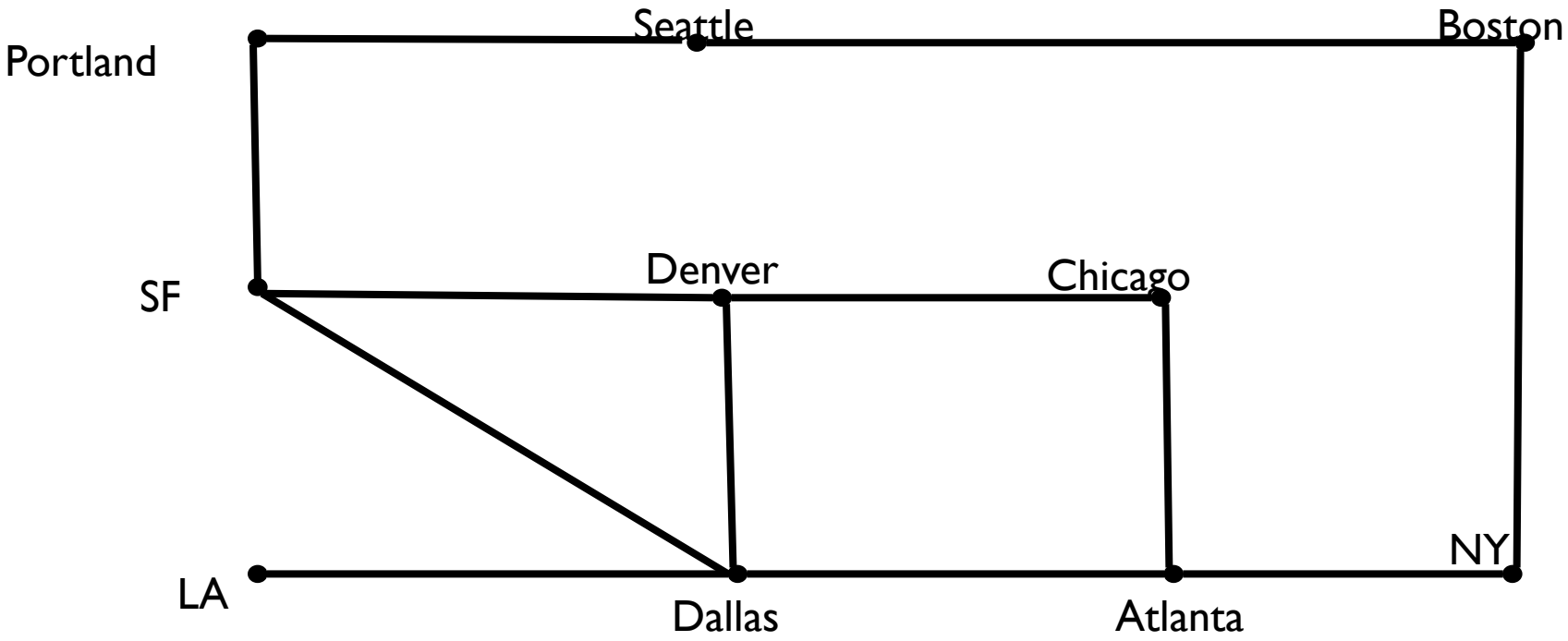


Nodes = courses; Edges = prerequisites ***

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

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

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

- 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 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)*

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

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 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*

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 subgraph connecting all vertices
      - Called a *connected, spanning subgraph*
    - What is a cheapest path from u to v?
  - And more....

Operations 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

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 BFS(G,v) to size of V

BFS Theorem

Thm. BFS(G,v) visits exactly those vertices u reachable from v.

Proof: We'll show that if u is reachable from v then 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 BFS(G,v) visits v
- Induction Hypothesis: For some d ≥ 0, if d(u,v) = d then BFS(G,v) visits u.

BFS Theorem

- Induction Step: Assume now that d(u,v) = d+1
  - Let $v = v_0, e_1, v_1, e_2, v_2, \ldots, v_d, e_{d+1}, v_{d+1} = u$ be a path of length d+1 from v to u
  - Then $v = v_0, e_1, v_1, e_2, v_2, \ldots, v_d$ is a path of length d from v to v_d
  - By I.H., v_d is visited by 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 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)

Reachability: Depth-First Search

```
DFS(G, v)       // Do a depth-first search of G starting at v
// pre: all vertices are marked as unvisited
count ←0;
Create empty stack S; push v; mark v as visited; count++;
While S isn't empty
        current ←S.pop();
        for each unvisited neighbor u  of current :
                add u to S; mark u as visited; count++
return count;
```

Now compare value returned from DFS(G,v) to size of V

DFS Reflections

- The DFS algorithm traced out a tree different from that produced by BFS
  - It still consists of the edges connecting a visited vertex to (as yet) unvisited neighbors
- It is called a DFS *tree of G with root v* (or *from v*)
- Vertices are processed in pre-order w.r.t. the tree
- By manipulating the stack differently, we could produce a post-order version of DFS
- And perhaps write DFS recursively….

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;
```

Is it even clear that this method does what we want?!

Let's prove some facts about it....

What *Exactly* Does DFS Do?

- Given a graph G = (V, E), a vertex v, let $X \subseteq V$, where $v \notin X$.
- Assume X are exactly the vertices of V that have been marked as visited
- Claim: DFS(G,v) will visit exactly those vertices that are in the connected component of G – X that contains v
  - G – X is the graph obtained by deleting the vertices of X–and edges using X–from G
  - Prove by induction on |V – X|

Recursive Depth-First Search

Claim: DFS visits all vertices w reachable from v

•Proof: Induction on length d of shortest path from v to w

- Base case: d = 0: Then v = w ✓
- Ind. Hyp.: Assume DFS visits all vertices w of distance at most d from v (for some d >= 0).
- Ind. Step: Suppose now that w is distance d+1 from v. Consider a path of length d+1 from v to w and let u be the next-to-last vertex on the path

Recursive Depth-First Search

Claim: DFS visits all vertices w reachable from v

- Proof: Induction on length d of shortest path from v to w
  - The path is $v = v_0, v_1, v_2, \ldots, v_d = u, v_{d+1} = w$
    - The edges are implied so not explicitly written!
  - By Ind. Hyp., u is visited. At this point, if w has not yet been visited, it will be one of the unvisited vertices on which DFS() is recursively called, so it will then be visited.

Recursive Depth-First Search

Claim: DFS visits *only* vertices reachable from v

•Idea: Prove by induction on number of times DFS is called that DFS is only called on vertices w reachable from v

Claim: DFS counts correctly the number of vertices reachable from v

- Idea: Induction on number of unvisited vertices reachable from v
  - DFS will never be called on same vertex twice

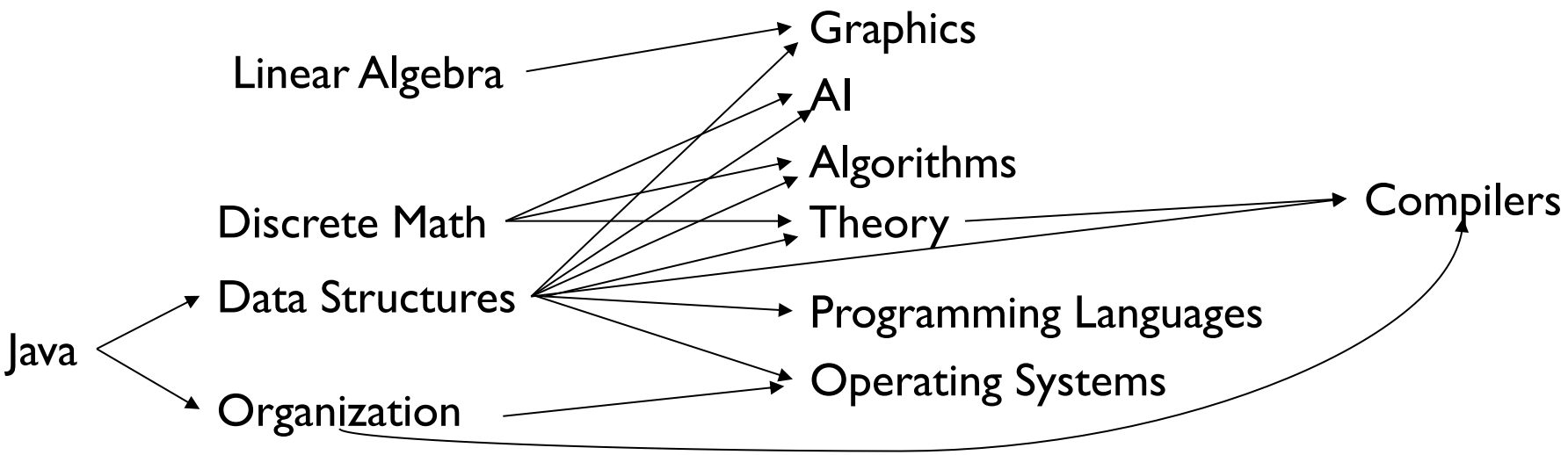
Recursive Depth-First Search

Claim: DFS(G,v) returns the number of unvisited nodes reachable from v

Proof: Uses previous two observations

- DFS visits every node reachable from v
- DFS doesn't visit any node *not* reachable from v

Directed Graphs

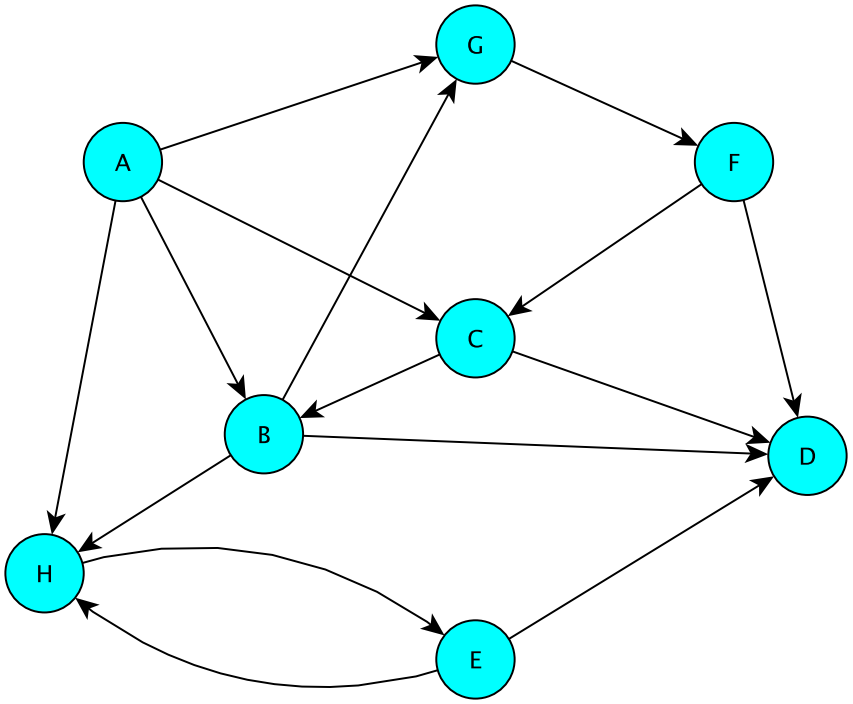


Def'n: In a *directed graph G = (V,E)*, each edge e in E is an *ordered* pair: e = (u,v) vertices: its *incident vertices*. The *source* of e is u; the *destination/target* is v.

Note: (u,v) ≠ (v,u)

Directed Graphs

- The (out) neighbors of B are D, G, H: B has out-degree 3
- The in neighbors of B are A, C: B has in-degree 2
- A has in-degree 0: it is a *source* in G; D has out-degree 0: it is a *sink* in G



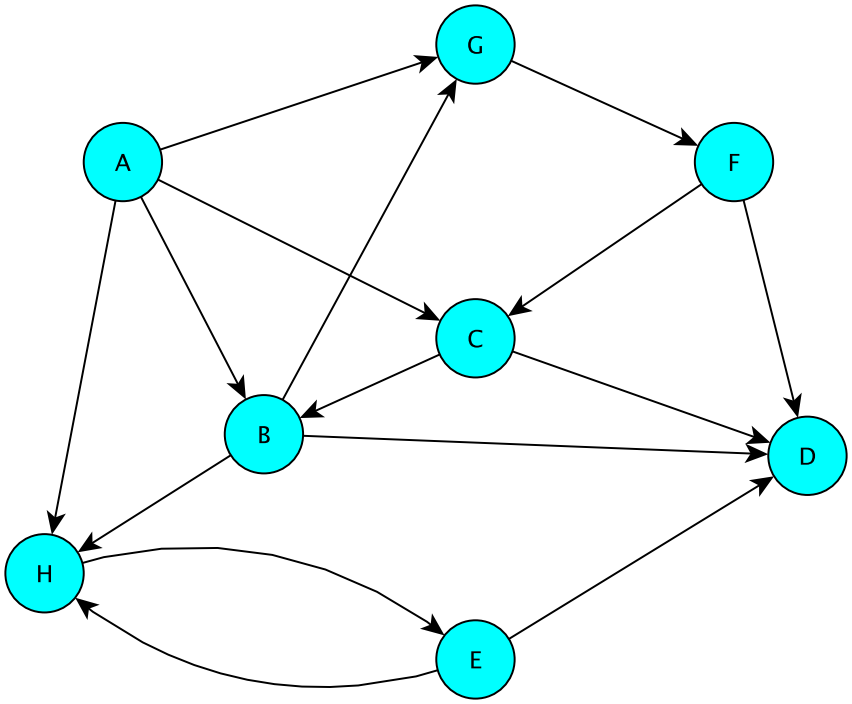
A walk is still an alternating sequence of vertices and edges

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

but now $e_i = (v_{i-1}, v_i)$: all edges *point along direction* of walk

Directed Graphs

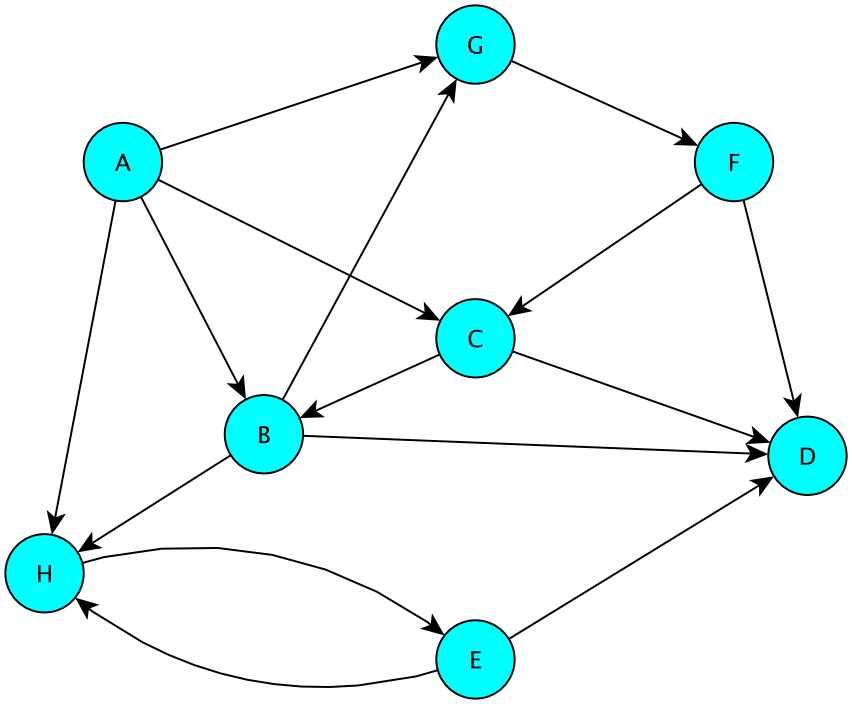
- A, B, H, E, D is a walk from A to D
- It's also a (simple) path
- D, E, H, B, A is *not* a walk from D to A
- B, G, F, C, B is a (directed) cycle (it's a 4-cycle)
- So is H, E, H (a 2-cycle)



- D is reachable from A (via path A, B, D), but A is not reachable from D
- In fact, every vertex is reachable from A

Directed Graphs

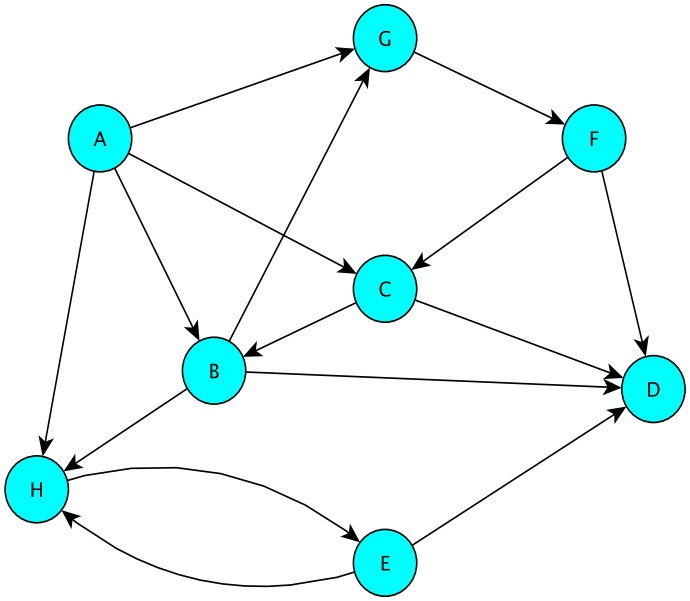
- A BFS of *G* from A visits every vertex
- A BFS of *G* from F visits all vertices but A
- A BFS of *G* from E visits only E, H, D

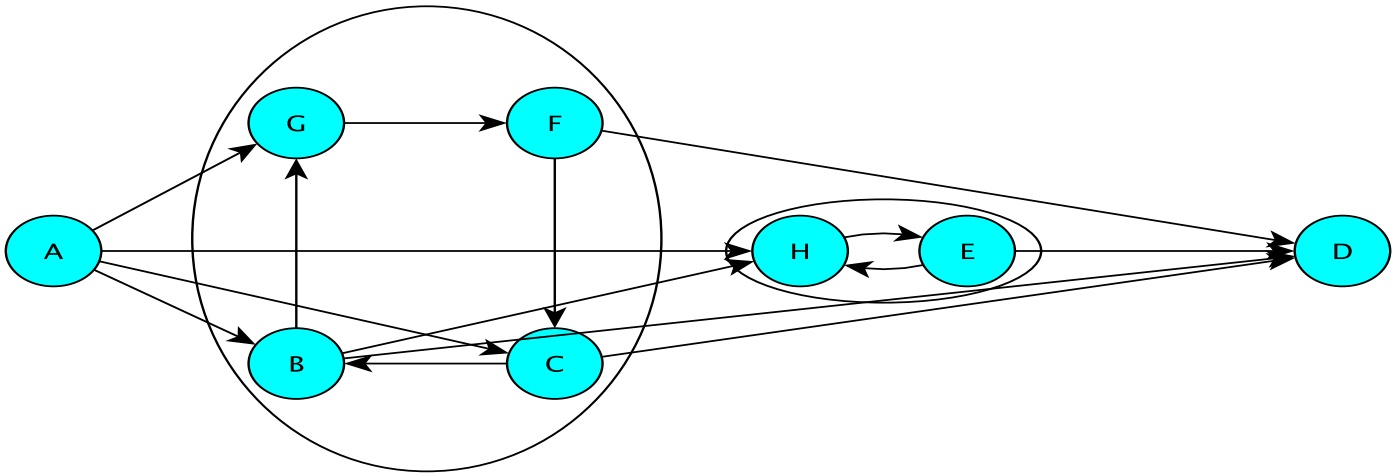


- Connectivity in directed graphs is more subtle than in undirected graphs!

Directed Graphs

- Vertices u and v are *mutually reachable* vertices if there are paths from u to v and v to u
- *Maximal* sets of mutually reachable vertices form *the strongly connected components* of G





Implementing Graphs

- Involves a number of implementation decisions, depending on intended uses
  - What kinds of graphs will be availabe?
    - 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

- We want to store information at vertices and at edges, but we 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
- Type E holds a *label* for an (available) edge type
  - Label: Application-specific data for a vertex/edge

Graphs in structure5

- So, the methods described in the Structure<V> 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

Recall: Desired Functionality

- 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 neighbors of v (or the edges incident with v)

Graph Interface Methods

- void add(V vtx), V remove(V vtx)
  - Add/remove vertex to/from graph
- void addEdge(V vtx1, V vtx2, E edgeLabel),
  E removeEdge(V vtx1, V vtx2)
  - Add/remove edge between vtx1 and vtx2
- boolean containsEdge(V vtx1, V vtx2)
  - Returns true iff there is an edge between vtx1 and vtx2
- Edge<V,E> getEdge(V vtx1, V vtx2)
  - Returns edge between vtx1 and vtx2
- void clear()
  - Remove all nodes (and edges) from graph

Graph Interface Methods

- boolean visit(V vertexLabel)
  - Mark vertex as "visited" and return *previous* value of visited flag
- boolean visitEdge(Edge<V,E> e)
  - Mark edge as "visited"
- boolean isVisited(V vtx), boolean isVisitedEdge(Edge<V,E> e)
  - Returns true iff vertex/edge has been visited
- Iterator<V> neighbors(V vtx1)
  - Get iterator for all neighbors of vtx1
  - For directed graphs, out-edges only
- Iterator<V> iterator()
  - Get vertex iterator
- void reset()
  - Remove visited flags for all nodes/edges

Edge Class

- Graph *edges* are defined in their own public class
  - `Edge<V,E>(  V vtx1, V vtx2,`
    `E label, boolean directed)`
  - Construct a (possibly directed) edge between two labeled vertices (vtx1->vtx2)
- Useful methods:

  `label(), here(), there()`

  `setLabel(), isVisited(), isDirected()`

Reachability: Breadth-First Traversal

```
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 BFS(G,v) to size of V

Breadth-First Traversal

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

Breadth-First Traversal of Edges

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