CSE 326: Data Structures
Graphs

James Fogarty

Autumn 2007

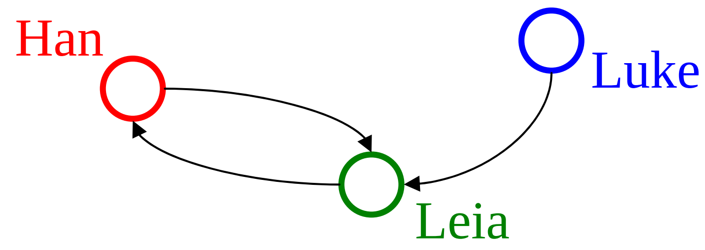
Graph… ADT?

- Not quite an ADT… operations not clear
- A formalism for representing relationships between objects

  Graph **G = (V,E)**

  – *Set of vertices*:
  $\mathbf{V = \{v_1, v_2, \ldots, v_n\}}$

  – *Set of edges*:
  $\mathbf{E = \{e_1, e_2, \ldots, e_m\}}$
  where each $\mathbf{e_i}$ connects two vertices $\mathbf{(v_{i1}, v_{i2})}$



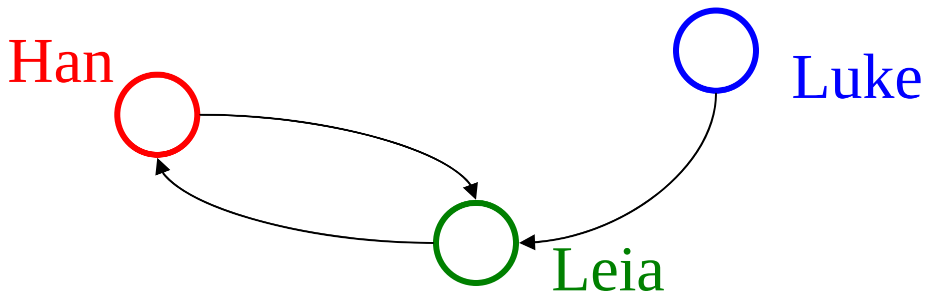

```
V = {Han, Leia, Luke}
E = {(Luke, Leia),
     (Han, Leia),
     (Leia, Han)}
```

Examples of Graphs

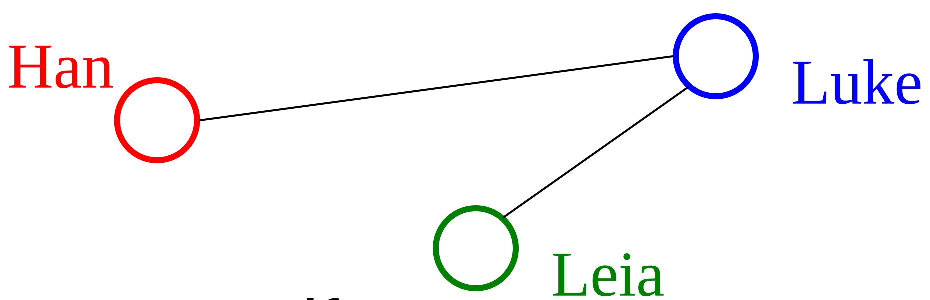
- The web
  - Vertices are webpages
  - Each edge is a link from one page to another
- Call graph of a program
  - Vertices are subroutines
  - Edges are calls and returns
- Social networks
  - Vertices are people
  - Edges connect friends

Graph Definitions

In *directed* graphs, edges have a direction:



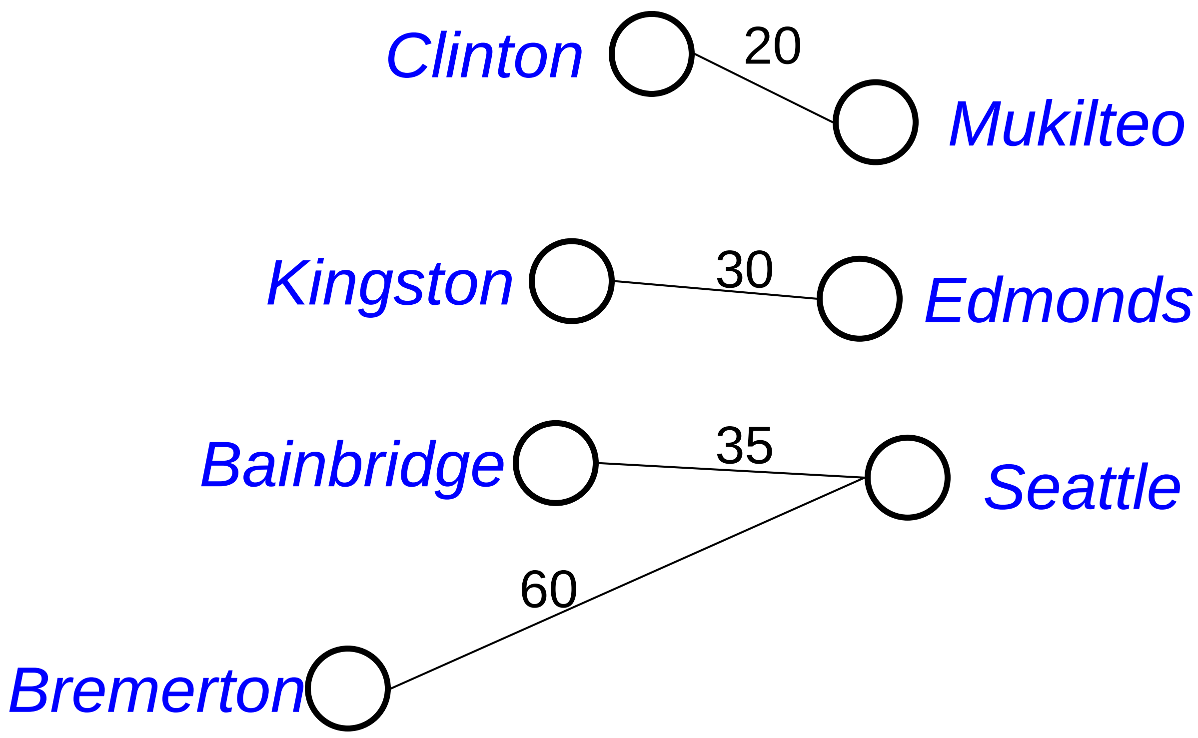
In *undirected* graphs, they don't (are two-way):



v is *adjacent* to **u** if **(u,v)** $\in$ **E**

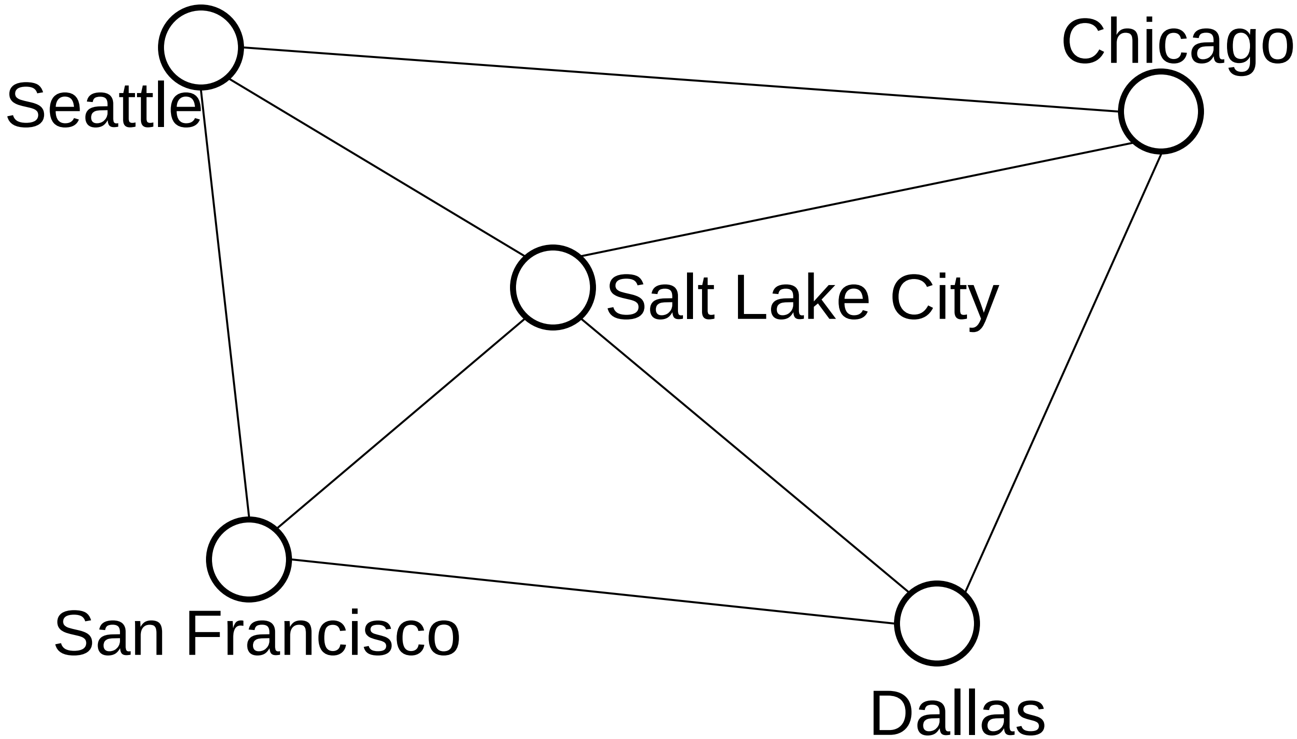
Weighted Graphs

Each edge has an associated weight or cost.



Paths and Cycles

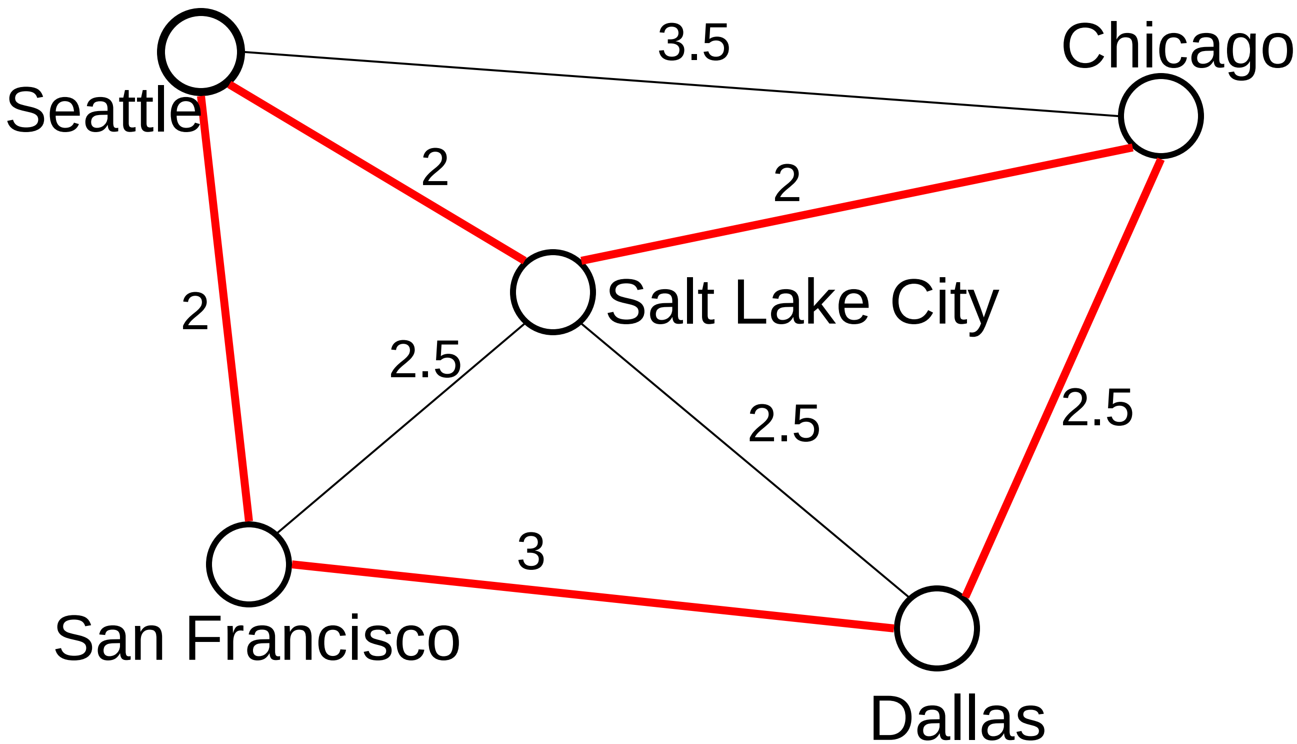
- A *path* is a list of vertices $\mathbf{\{v_1, v_2, \ldots, v_n\}}$ such that $\mathbf{(v_i, v_{i+1})} \in \mathbf{E}$ for all $\mathbf{0} \leq \mathbf{i < n}$.
- A *cycle* is a path that begins and ends at the same node.



- *p = {Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle}*

Path Length and Cost

- *Path length*: the number of edges in the path
- *Path cost*: the sum of the costs of each edge



length(p) = 5

cost(p) = 11.5

More Definitions: Simple Paths and Cycles

A *simple path* repeats no vertices (except that the first can also be the last):

p = {Seattle, Salt Lake City, San Francisco, Dallas}

p = {Seattle, Salt Lake City, Dallas, San Francisco, Seattle}

A *cycle* is a path that starts and ends at the same node:

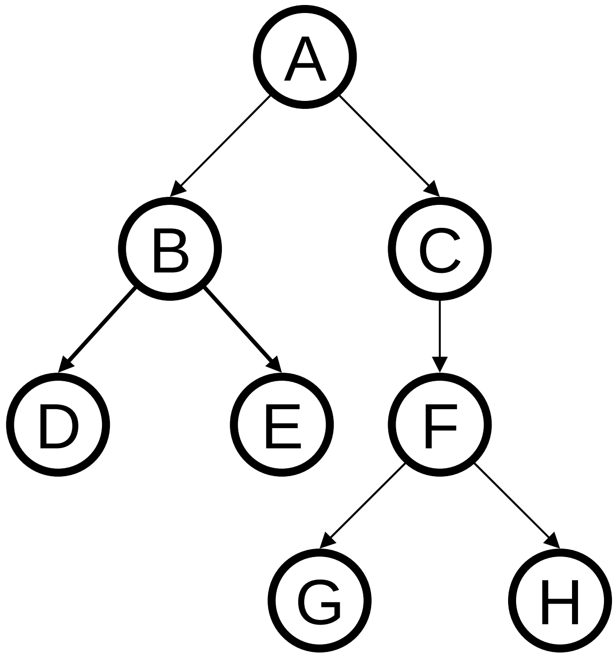
p = {Seattle, Salt Lake City, Dallas, San Francisco, Seattle}

p = {Seattle, Salt Lake City, Seattle, San Francisco, Seattle}

A *simple cycle* is a cycle that is also a simple path (in undirected graphs, no edge can be repeated)

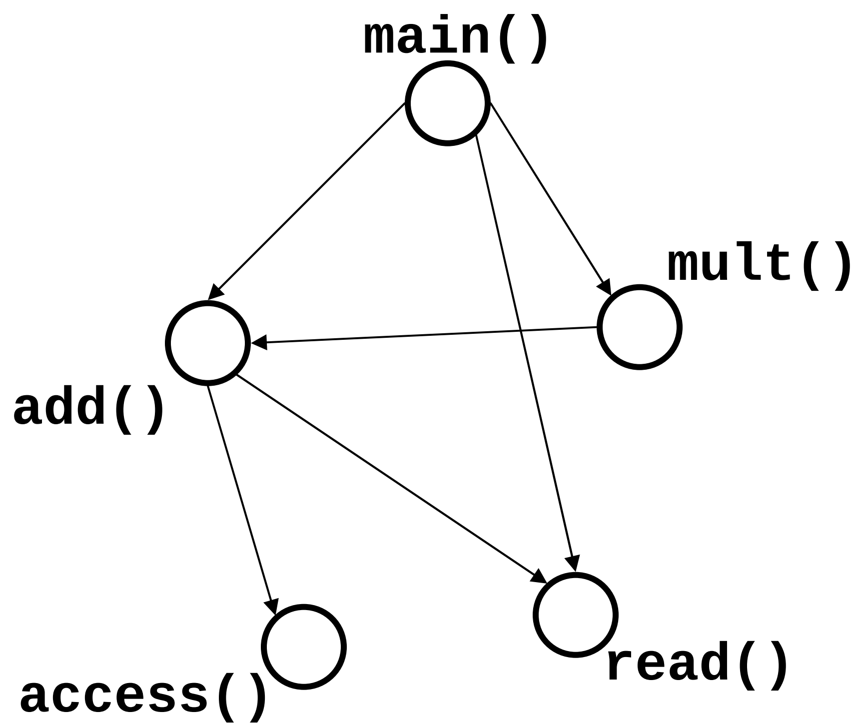
Trees as Graphs

- Every tree is a graph with some restrictions:
  - the tree is *directed*
  - there are *no cycles* (directed or undirected)
  - there is a *directed path from the* root *to every node*



Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.

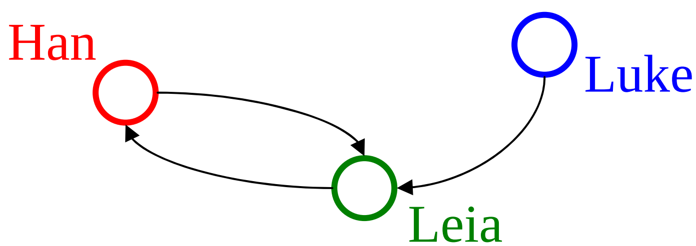


Aside: *If program call-graph is a DAG, then all procedure calls can be in-lined*

{Tree} ⊂ {DAG} ⊂ {Graph}

Rep 1: Adjacency Matrix

A **|V| x |V|** array in which an element **(u,v)** is true if and only if there is an edge from **u** to **v**



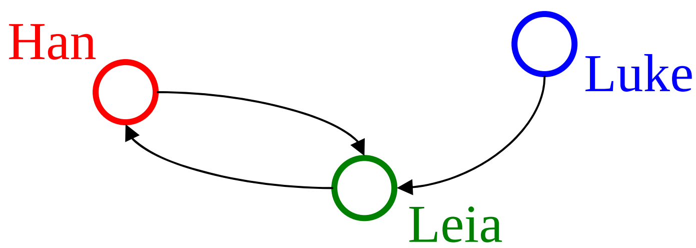
	Han	Luke	Leia
Han			
Luke			
Leia			

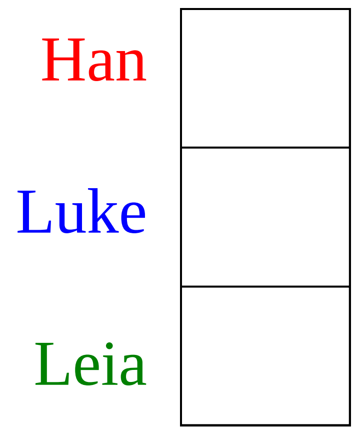
Runtimes:
Iterate over vertices?
Iterate over edges?
Iterate edges adj. to vertex?
Existence of edge?

Space requirements?

Rep 2: Adjacency List

A **|V|**-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices

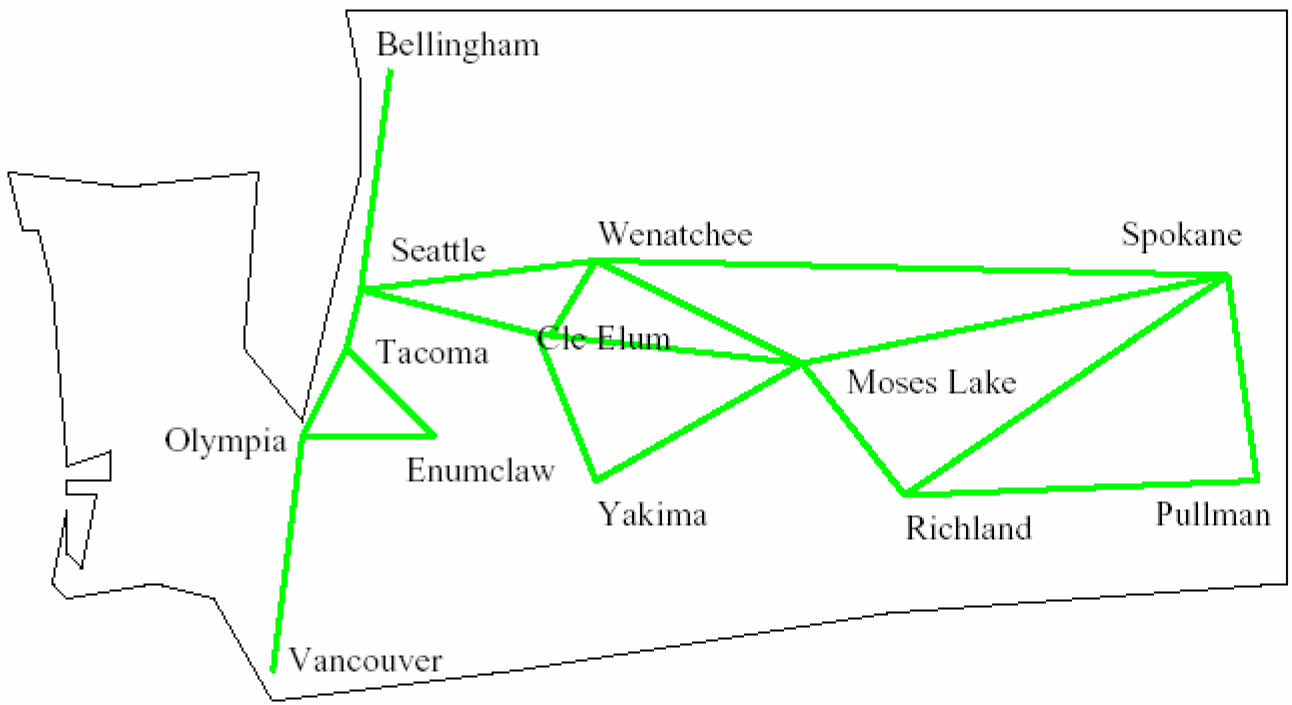




Runtimes:
Iterate over vertices?
Iterate over edges?
Iterate edges adj. to vertex?
Existence of edge?

Space requirements?

Some Applications:
Moving Around Washington

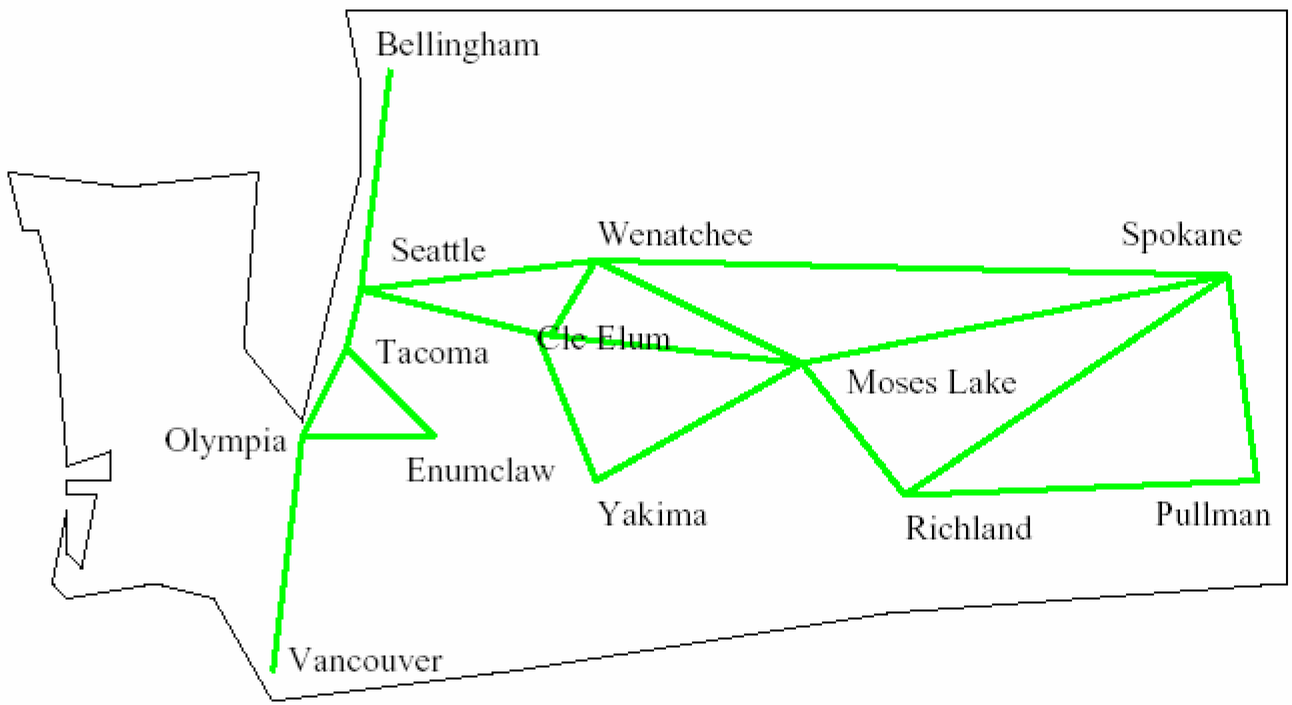


What's the *shortest way* to get from Seattle to Pullman?

Edge labels:

Distance

Some Applications: Moving Around Washington

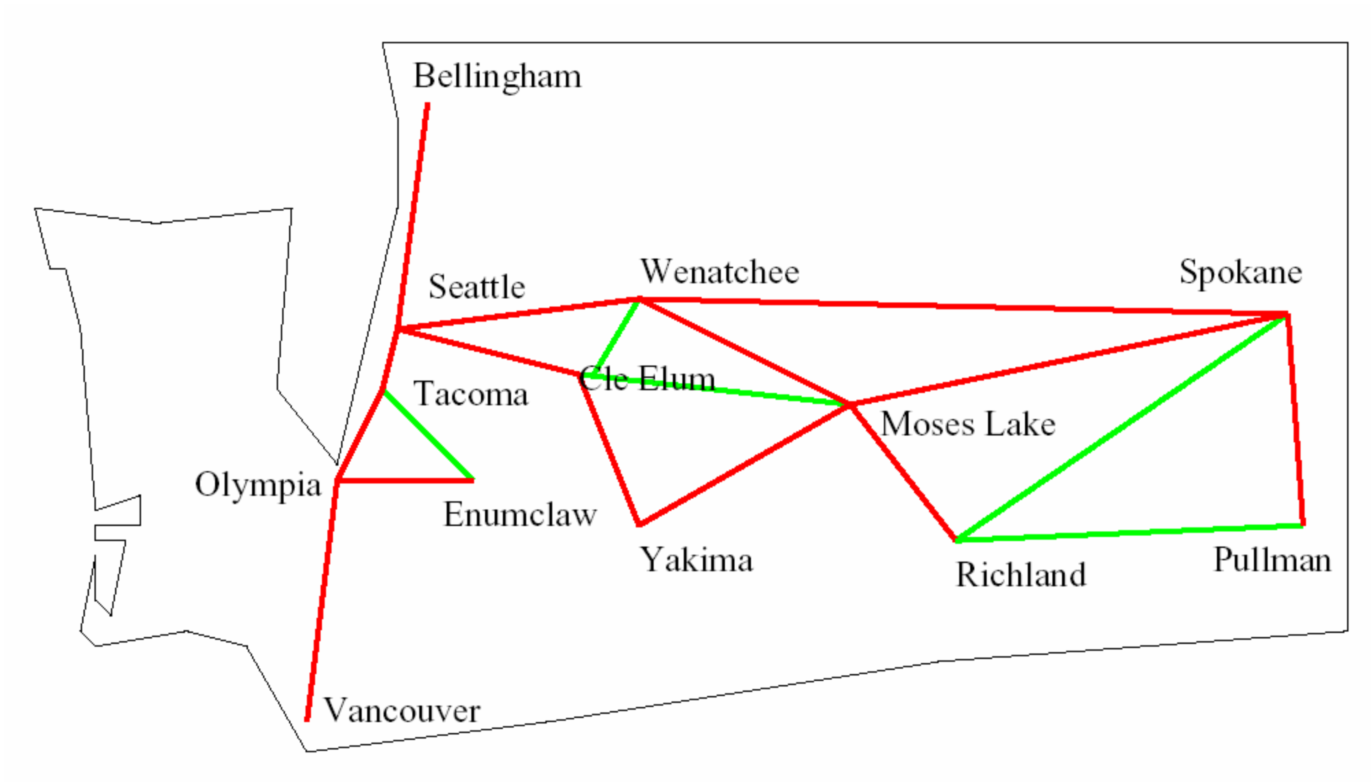


What’s the *fastest way* to get from Seattle to Pullman?

Edge labels:

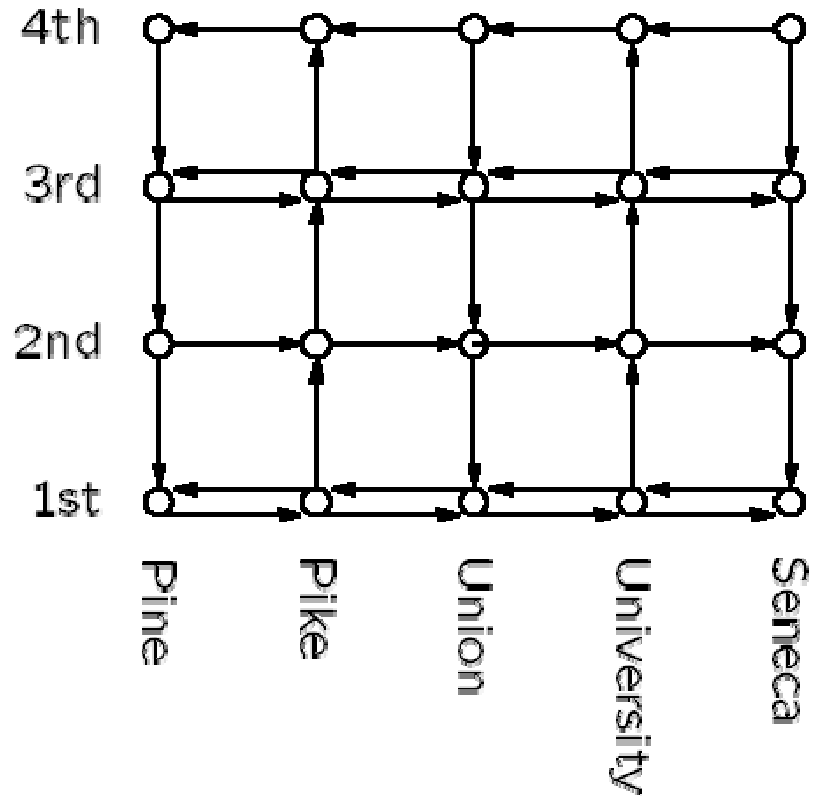
Distance, speed limit

Some Applications: Reliability of Communication



If Wenatchee's phone exchange *goes down*,
can Seattle still talk to Pullman?

Some Applications: Bus Routes in Downtown Seattle

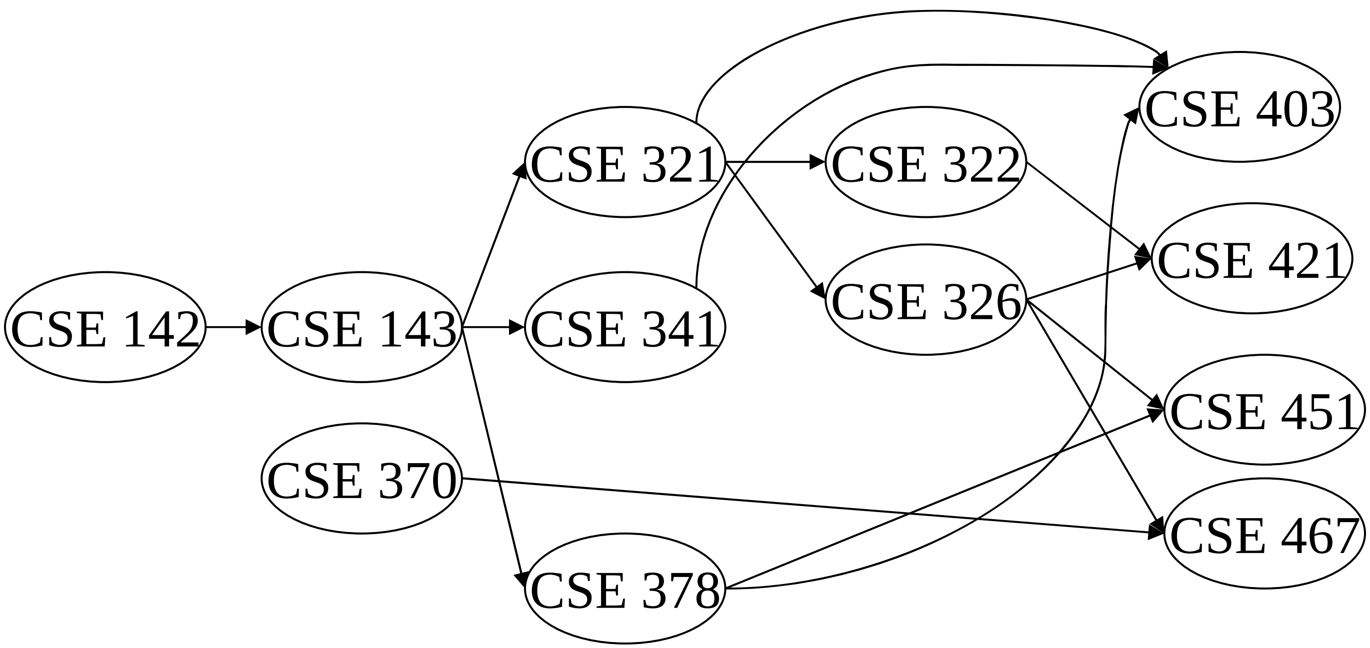


If we're at 3rd and Pine, how can we get to 1st and University using Metro?
How about 4th and Seneca?

This is a partial ordering, for sorting we had a total ordering

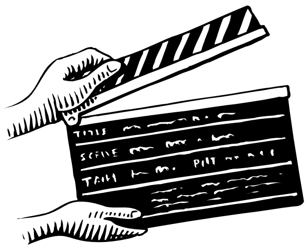
Application: Topological Sort

Given a directed graph, **`G = (V,E)`**, output all the vertices in **`V`** such that no vertex is output before any other vertex with an edge to it.



Is the output unique?

Minimize and
DO a topo sort



Topological Sort: Take One

1. Label each vertex with its *in-degree* (# of inbound edges)
2. **While** there are vertices remaining:
   a. Choose a vertex *v* of *in-degree zero*; output *v*
   b. Reduce the in-degree of all vertices adjacent to *v*
   c. Remove *v* from the list of vertices

Runtime:

```
void Graph::topsort(){
  Vertex v, w;

  labelEachVertexWithItsIn-degree();

  for (int counter=0; counter < NUM_VERTICES;
                                    counter++){
    v = findNewVertexOfDegreeZero();

    v.topologicalNum = counter;
    for each w adjacent to v
      w.indegree--;
  }
}
```

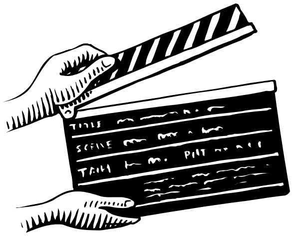
Time?

Time?

Time?

What's the bottleneck?

O(depends)



Topological Sort: Take Two

1. Label each vertex with its in-degree
2. Initialize a queue *Q* to contain all in-degree zero vertices
3. While *Q* not empty
   a. *v* = *Q*.dequeue; output *v*
   b. Reduce the in-degree of all vertices adjacent to *v*
   c. If new in-degree of any such vertex *u* is zero *Q*.enqueue(*u*)

<u>Note</u>: could use a stack, list, set, box, … instead of a queue

Runtime:

```
void Graph::topsort(){
  Queue q(NUM_VERTICES);  int counter = 0; Vertex v, w;
   labelEachVertexWithItsIn-degree();

  q.makeEmpty();
  for each vertex v
    if (v.indegree == 0)
      q.enqueue(v);

  while (!q.isEmpty()){
    v = q.dequeue();
    v.topologicalNum = ++counter;
    for each w adjacent to v
      if (--w.indegree == 0)
        q.enqueue(w);
  }
}
```

intialize the queue

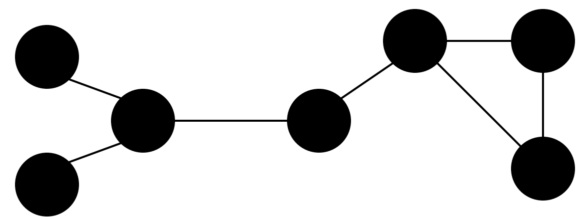
get a vertex with indegree 0

insert new eligible vertices

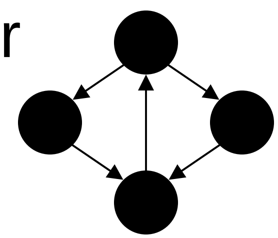
Runtime: O(|V| + |E|)

Graph Connectivity

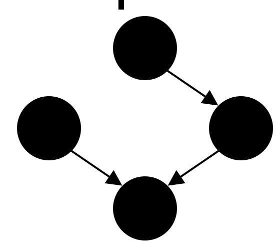
Undirected graphs are *connected* if there is a path between any two vertices



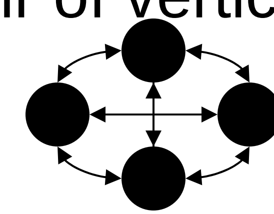
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



A *complete* graph has an edge between every pair of vertices



Graph Traversals

- Breadth-first search (and depth-first search) work for arbitrary (directed or undirected) graphs - not just mazes!
  - Must mark visited vertices so you do not go into an infinite loop!
- Either can be used to determine connectivity:
  - Is there a path between two given vertices?
  - Is the graph (weakly) connected?
- Which one:
  - Uses a queue?
  - Uses a stack?
  - Always finds the shortest path (for unweighted graphs)?