## Graphs
## Chapter 9 in Weiss

CSE 326
Data Structures
Ruth Anderson



## Today's Outline

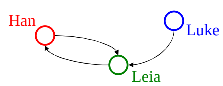
- **Announcements**
  - **Written Homework #6 due NOW**
  - **Project 3 Code due Mon March 1 by 11pm**
  - **Project 3 Benchmarking & Written due Thurs March 4 by 11pm**
- **Today's Topics:**
  - **Sorting**
  - **Graphs**



## Graph… ADT?

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

  Graph `G = (V,E)`
  - *Set of vertices*: $V = \{v_1, v_2, \ldots, v_n\}$
  - *Set of edges*: $E = \{e_1, e_2, \ldots, e_m\}$ where each $e_i$ connects two vertices $(v_{i1}, v_{i2})$

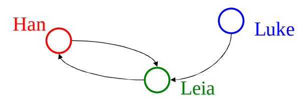



```
V = {Han, Leia, Luke}
E = {(Luke, Leia),
     (Han, Leia),
     (Leia, Han)}
```

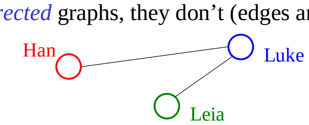


## Graph Definitions

In *directed* graphs, edges have a specific direction:




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




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



## More Definitions: Simple Paths and Cycles

A *simple path* repeats no vertices (except that the first can 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}

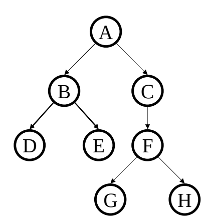
A *simple cycle* is a cycle that repeats no vertices except that the first vertex is also the last (in undirected graphs, no edge can be repeated)



## Trees as Graphs

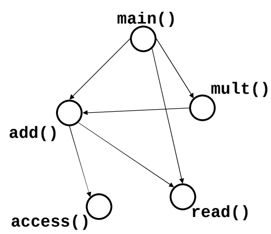
- Every tree is a graph!
- Not all graphs are trees!

A graph is a tree if
- There are *no cycles* (directed or undirected)
- There is a *path* from the root *to every node*



# Directed Acyclic Graphs (DAGs)

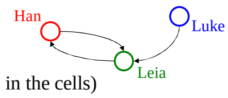
DAGs are directed graphs with no (directed) cycles.




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



# Graph Representations




0. List of vertices + list of edges
1. 2-D matrix of vertices (marking edges in the cells)
   "adjacency matrix"
2. List of vertices each with a list of adjacent vertices
   "adjacency list"

Things we might want to do:

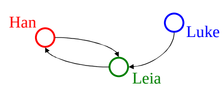
- iterate over vertices
- iterate over edges
- iterate over vertices adj. to a vertex
- check whether an edge exists

Vertices and edges may be labeled



# Representation 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 | | | |

*space requirements:*

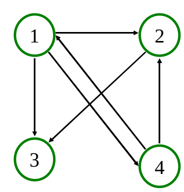
*runtime:*



# Representation

- adjacency **matrix**:

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |

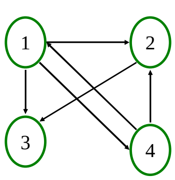


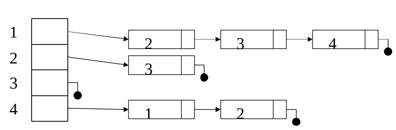

$$A[u][v] = \begin{cases} \text{weight} & \text{, if } (u, v) \in E \\ 0 & \text{, if } (u, v) \notin E \end{cases}$$



# Representation

- adjacency **list:**

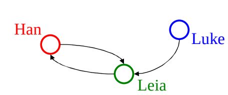





# Representation 2: Adjacency List

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




| | |
|---|---|
| Han | |
| Luke | |
| Leia | |

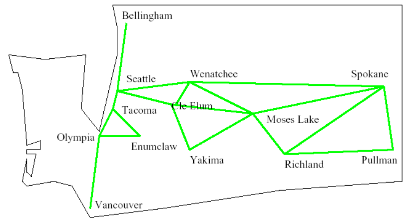
*space requirements:*

*runtime:*

# Good match?

| | List of edges and list of vertices | Adjacency matrix | Adjacency list |
|---|---|---|---|
| **Iterate over vertices** | | | |
| **Iterate over edges** | | | |
| **Check if edge exists** | | | |
| **Iterate over vertices adjacent to a vertex** | | | |

# Some Applications: Moving Around Washington

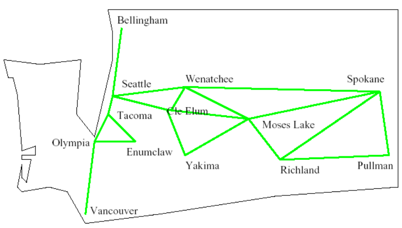



What's the *shortest way* to get from Seattle to Pullman?

Edge labels:



# Some Applications: Moving Around Washington

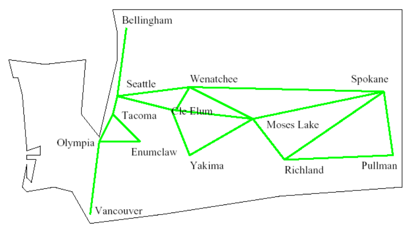



What's the *fastest way* to get from Seattle to Pullman?

Edge labels:



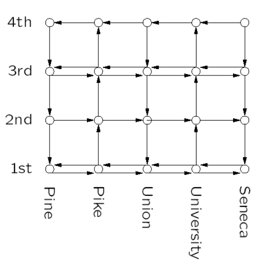
# Some Applications: Reliability of Communication




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



# Some Applications: Bus Routes in Downtown Seattle

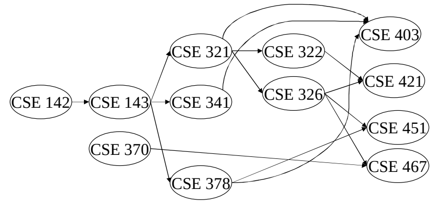



If we're at 3rd and Pine, how can we get to 1st and University using Metro?



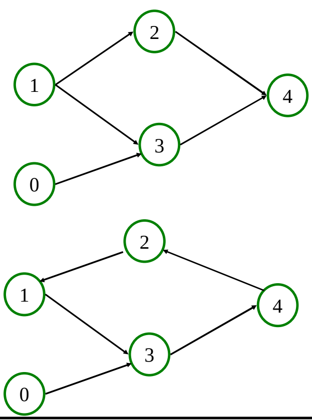
# Application: Topological Sort

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?***

Valid Topological Sorts:




# 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--;
  }
}
```



# 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*)

Note: 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();                 intialize the queue
  for each vertex v
    if (v.indegree == 0)
      q.enqueue(v);

  while (!q.isEmpty()){          get a vertex with indegree 0
    v = q.dequeue();
    v.topologicalNum = ++counter;
    for each w adjacent to v
      if (--w.indegree == 0)     insert new eligible vertices
        q.enqueue(w);
  }
}
```

*Runtime:*

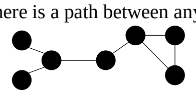


# Graph Traversals

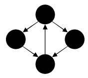
- Breadth-first search (and depth-first search) work for arbitrary (directed or undirected) graphs - not just mazes!
  - Must mark visited vertices so you do not go into an infinite loop!
- Either can be used to determine connectivity:
  - Is there a path between two given vertices?
  - Is the graph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

# Graph Connectivity

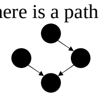
Undirected graphs are *connected* if there is a path between any two vertices



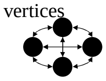
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



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





# The Shortest Path Problem

Given a graph $G$, edge costs $c_{i,j}$, and vertices $s$ and $t$ in $G$, find the shortest path from $s$ to $t$.

For a path $p = v_0\ v_1\ v_2 \dots v_k$

– *unweighted length* of path $p = k$ (a.k.a. *length*)

– *weighted length* of path $p = \sum_{i=0..k-1} c_{i,i+1}$ (a.k.a *cost*)

Path length equals path cost when ?



# Single Source Shortest Paths (SSSP)

Given a graph $G$, edge costs $c_{i,j}$, and vertex $s$, find the shortest paths from $s$ to all vertices in G.

– Is this harder or easier than the previous problem?



# All Pairs Shortest Paths (APSP)

Given a graph $G$ and edge costs $c_{i,j}$, find the shortest paths between all pairs of vertices in G.

– Is this harder or easier than SSSP?

– Could we use SSSP as a subroutine to solve this?



# Variations of SSSP

– Weighted vs. unweighted
– Directed vs undirected
– Cyclic vs. acyclic
– Positive weights only vs. negative weights allowed
– Shortest path vs. longest path
– …



# Applications

– Network routing
– Driving directions
– Cheap flight tickets
– Critical paths in project management (see textbook)
– …

# SSSP: Unweighted Version

*Ideas?*



```
void Graph::unweighted (Vertex s){
  Queue q(NUM_VERTICES);
  Vertex v, w;
  q.enqueue(s);
  s.dist = 0;

  while (!q.isEmpty()){
    v = q.dequeue();
    for each w adjacent to v
      if (w.dist == INFINITY){
        w.dist = v.dist + 1;
        w.path = v;
        q.enqueue(w);
      }
    }
  }
```

**each edge examined at most once – if adjacency lists are used**

**each vertex enqueued at most once**

total running time: O(            )