Graph-theoretic Models

(download slides and .py files from Stellar to follow along)

John Guttag

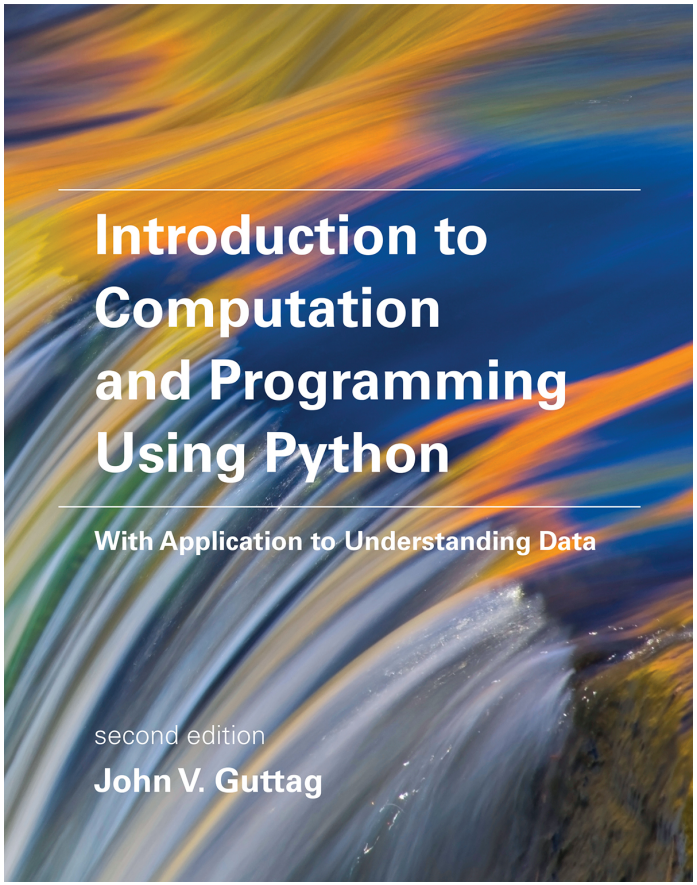
MIT Department of Electrical Engineering and Computer Science

Halloween Doings

- First 6.0002 micro quiz will be on Wednesday
- If you get accommodations, please make sure to have made arrangements by end of the day tomorrow

Relevant Reading

- Today
  - Section 12.2
- Wednesday
  - Chapter 15.1-15.4.1, 15.5

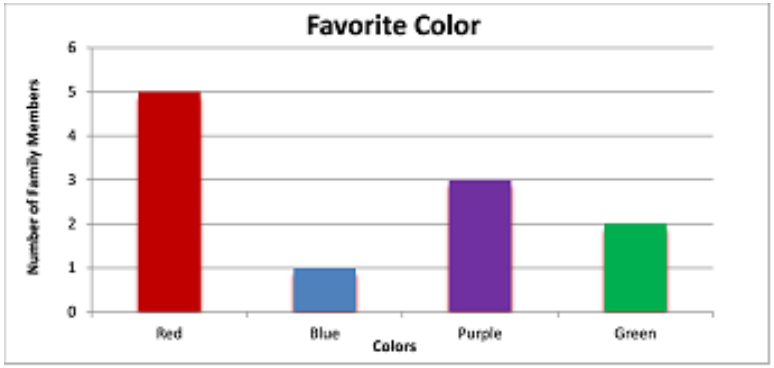


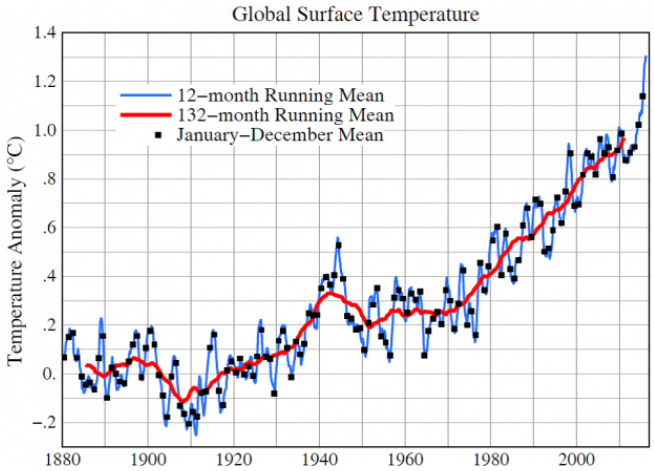
https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf

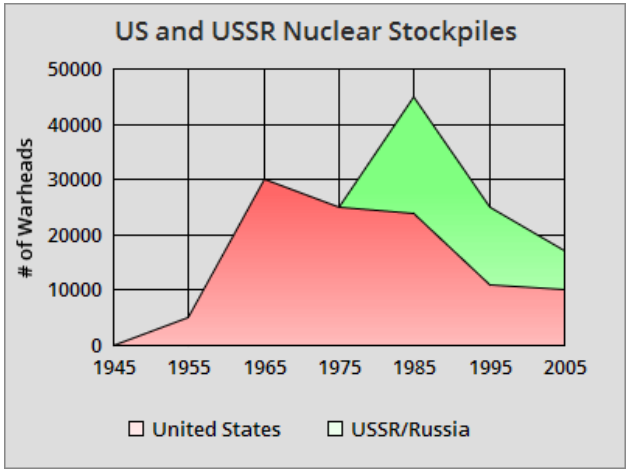
Computational Models

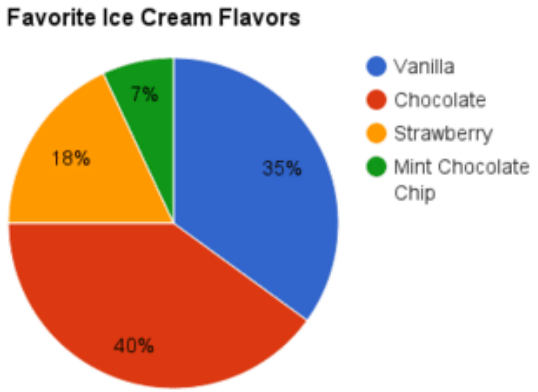
- Programs that help us understand the world and solve practical problems
- Saw how we could map the informal problem of choosing what to eat into an optimization problem, and how we could design a program to solve it
  - Saw how a decision tree can help find a good solution to an optimization problem
- Now want to look at a class of models called graphs
  - Nice way to formulate many problems
  - Often lead to nice optimization problems

What is a Graph?



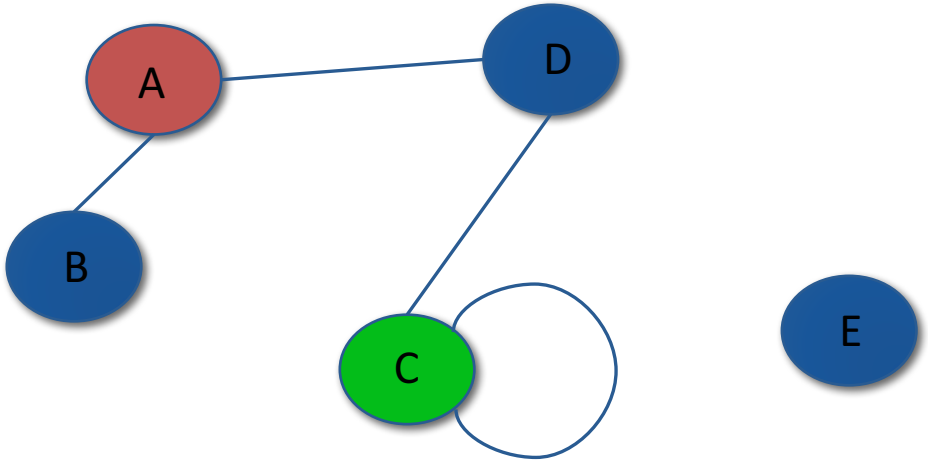






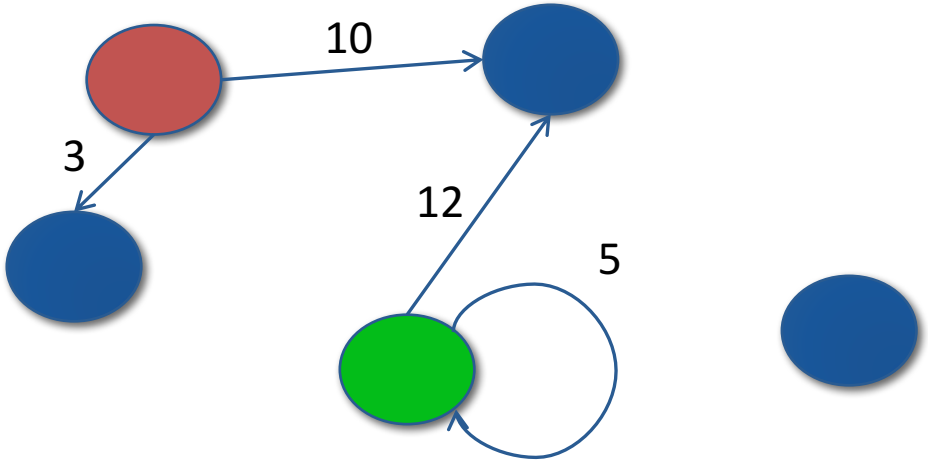
What is a Graph?

- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted



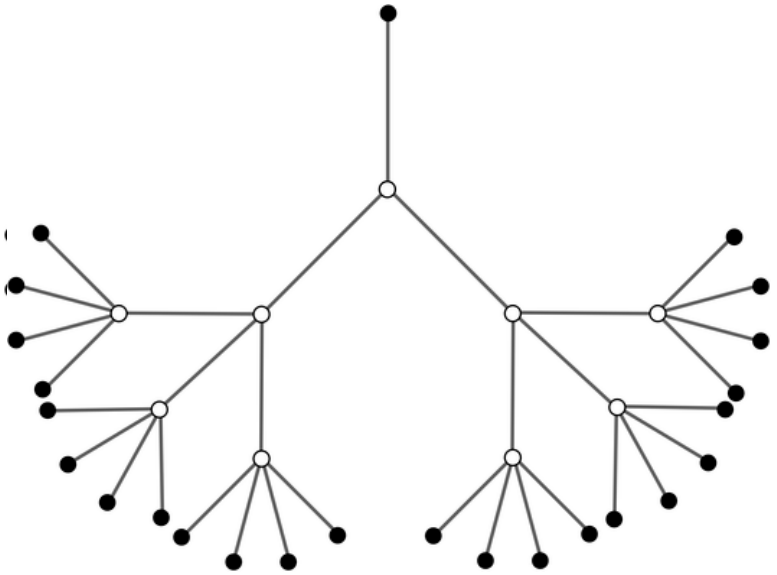
What is a Graph?

- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted



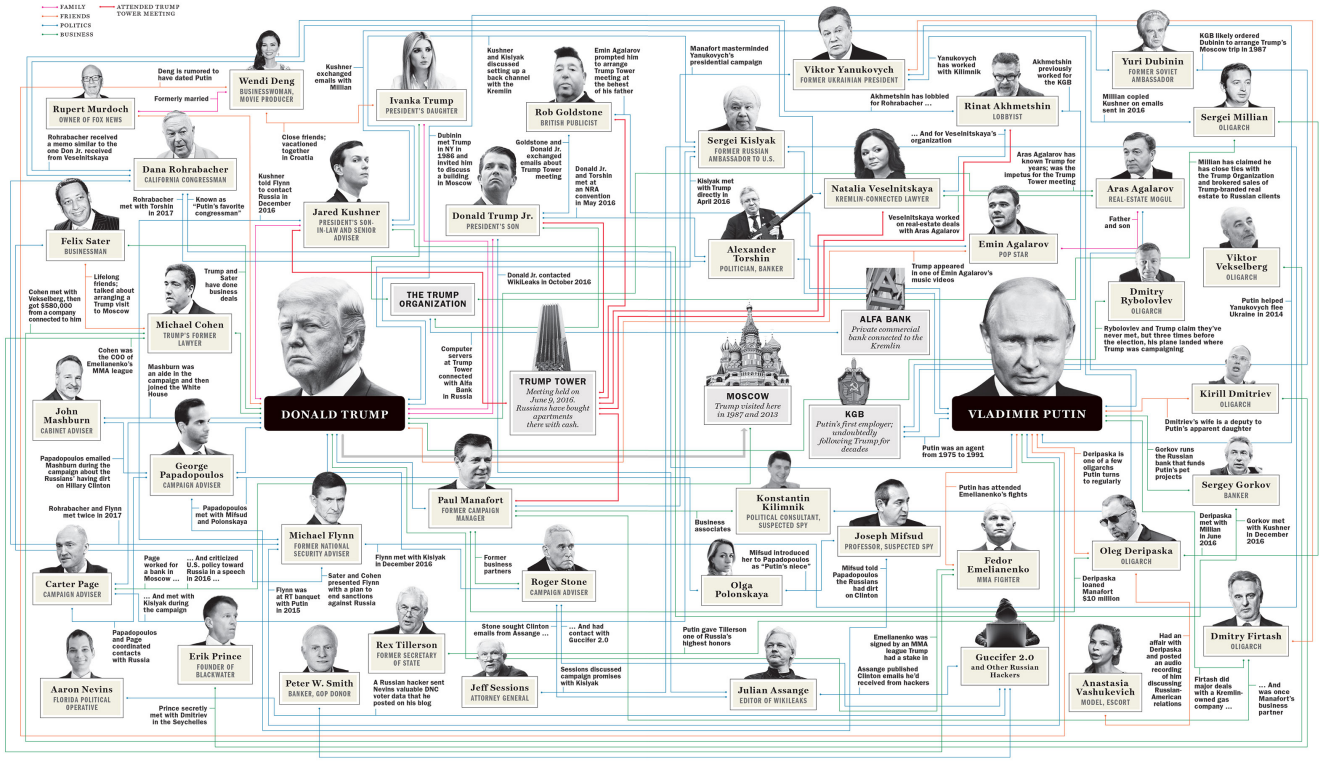
Trees: An Important Special Case

- A special kind of directed graph in which any pair of nodes is connected by a single path
  - Recall the search trees we used to solve knapsack problem



Why Graphs?

- To capture relationships among entities
  - Rail links between Paris and London
  - How the atoms in a molecule are related to one another
  - Business/social/political connections
  - …

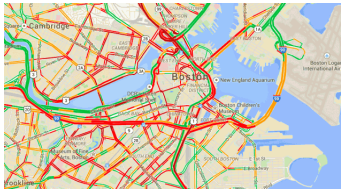


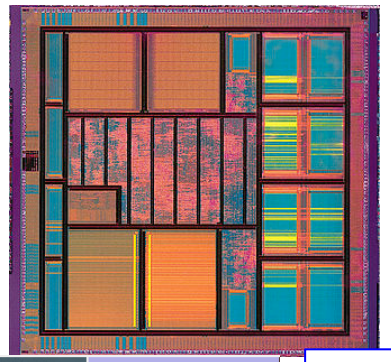
New York Magazine

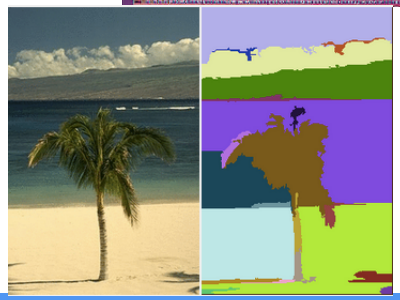
Why Graphs Are So Useful

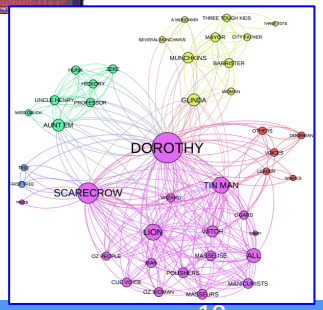
- Not only do graphs capture relationships in connected networks of items, they provide convenient ways to formulate questions about those relationships

- Find sequences of links between elements – is there a path from A to B
- Finding least expensive path between elements (aka shortest path problem)
- Partitioning graph into subgraphs with minimal connections between them (aka graph partition problem or graph clique problem)
- Finding the most efficient way to separate sets of connected elements (aka the min-cut/max-flow problem)



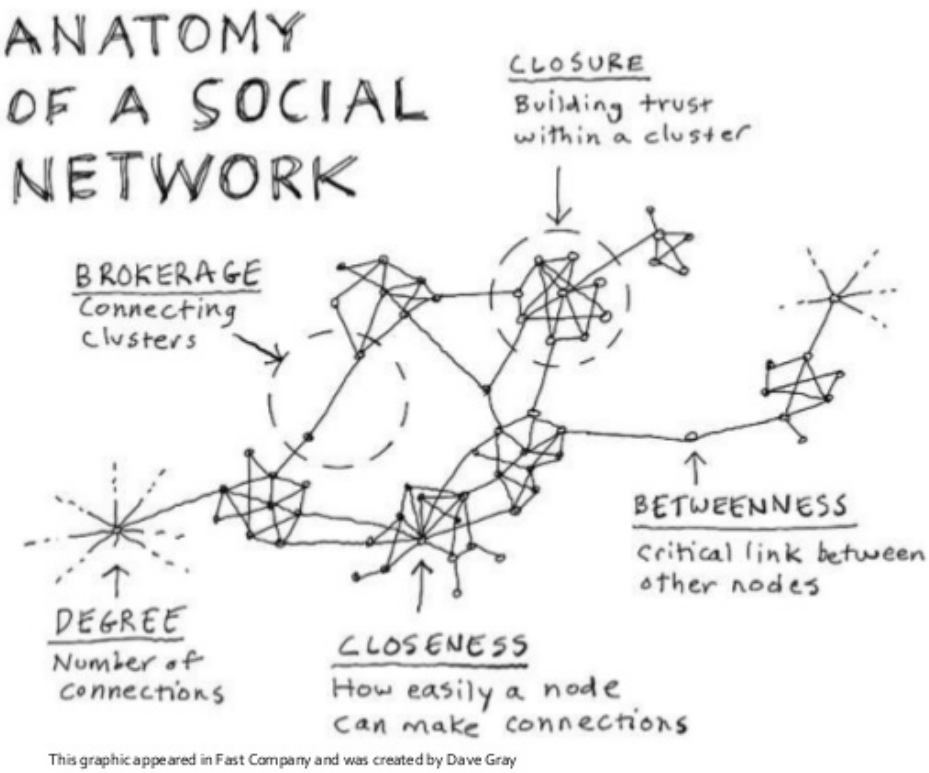






Example: Modeling Social Networks

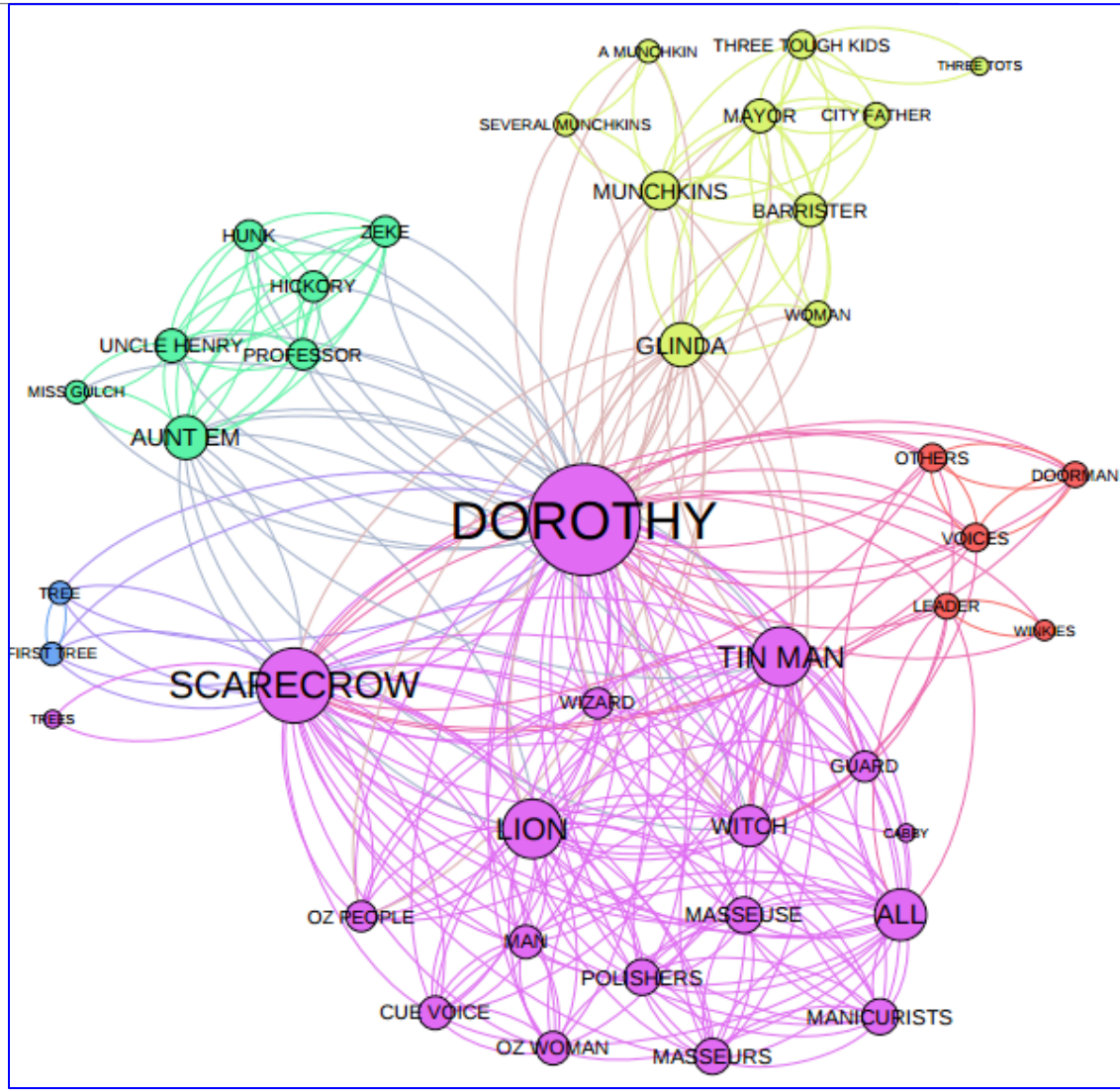
- Understanding a social network
  - Diffusion of misinformation
  - Homophily (clusters of people with similar charactersistics)
  - Bridgers (people who connect different groups)
  - Etc.



This graphic appeared in Fast Company and was created by Dave Gray

Analyzing Texts

- *Wizard of Oz*:
  - Size of node reflects number of scenes in which character shares dialogue
  - Color of clusters reflects strong interactions with each other



mapr.com

Some path problems are easier than others

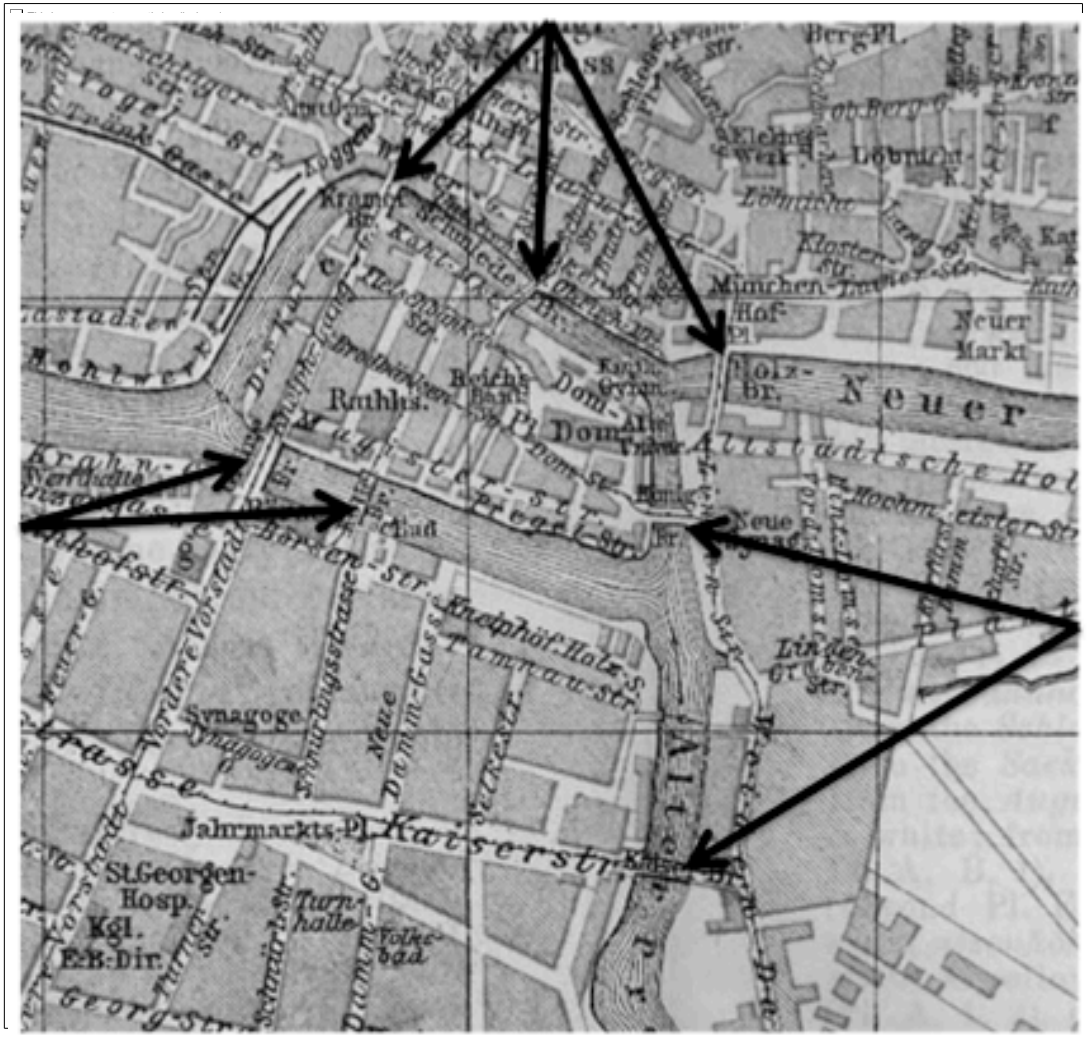
Getting John to his Office

- Model road system using a digraph
  - Nodes: points where roads end or meet
  - Edges: weighted connections between points
    - Expected time between source and destination nodes
    - Distance between source and destination nodes
- Solve a graph optimization problem
  - Shortest weighted path between my house and my office

Leads to different optimization functions

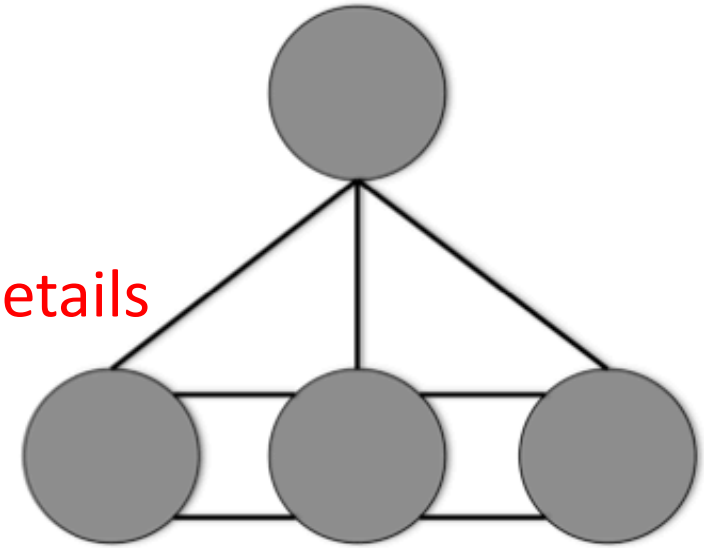
First Reported Use of Graph Theory

- Bridges of Königsberg (1735)
- Possible to take a walk that traverses each of the 7 bridges exactly once?



Leonhard Euler's Model

- Each island a node
- Each bridge an undirected edge
- Model abstracts away irrelevant details
  - Size of islands
  - Length of bridges



  - Is there a path that contains each edge exactly once?
  - **No!**
    - For such a path to exist, each node except the first and last must have an even number edges
    - No node has even number of edges

What's Interesting About This

- Not the Königsberg bridges problem itself
- The way Euler solved it
- A new way to think about a very large class of problems

Implementing Graphs

- Building graphs
  - Nodes
  - Edges
  - Stitching them together to make graphs
- Using graphs
  - Many well know problems and algorithms for solving them

Class Node

```
class Node(object):
    def __init__(self, name):
        """Assumes name is a string"""
        self.name = name
    def getName(self):
        return self.name
    def __str__(self):
        return self.name
```

Class Edge

```
class Edge(object):
    def __init__(self, src, dest):
        """Assumes src and dest are nodes"""
        self.src = src
        self.dest = dest
    def getSource(self):
        return self.src
    def getDestination(self):
        return self.dest
    def __str__(self):
        return self.src.getName() + '->'\
               + self.dest.getName()
```

Common Representations of Digraphs

- Digraph is a directed graph
  - Edges pass in one direction only
- Adjacency matrix
  - Rows: source nodes
  - Columns: destination nodes
  - Cell[s, d] = 1 if there is an edge from s to d
    = 0 otherwise
  - Note that in digraph, matrix is **not** symmetric
- Adjacency list
  - Associate with each node a list of destination nodes

Class Digraph, part 1

```
class Digraph(object):
    """edges is a dict mapping each node to a list of
    its children"""

    def __init__(self):
        self.edges = {}

    def addNode(self, node):
        if node in self.edges:
            raise ValueError('Duplicate node')
        else:
            self.edges[node] = []

    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not (src in self.edges and dest in self.edges):
            raise ValueError('Node not in graph')
        self.edges[src].append(dest)
```

Nodes are used as keys in dictionary

Edges are represented by destinations as values in list associated with a source key

A directed edge

Class Digraph, part 2

```
def childrenOf(self, node):
    return self.edges[node]

def hasNode(self, node):
    return node in self.edges

def getNode(self, name):
    for n in self.edges:
        if n.getName() == name:
            return n
    raise NameError(name)

def __str__(self):
    result = ''
    for src in self.edges:
        for dest in self.edges[src]:
            result = result + src.getName() + '->'\
                     + dest.getName() + '\n'
    return result[:-1] #omit final newline
```

Class Graph

```
class Graph(Digraph):
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)
```

Override addEdge from Graph

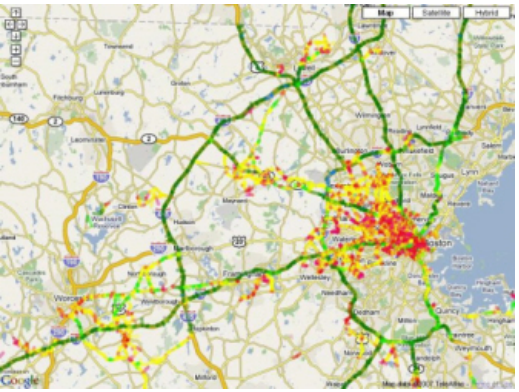
- Graph does not have directionality associated with an edge
  - Edges allow passage in either direction
- Why is Graph a subclass of Digraph?
- Remember the **substitution** rule?
  - If client code works correctly using an instance of the supertype, it should also work correctly when an instance of the subtype is substituted for the instance of the supertype
- Any program that works with a Digraph will also work with a Graph (but not *vice versa*)

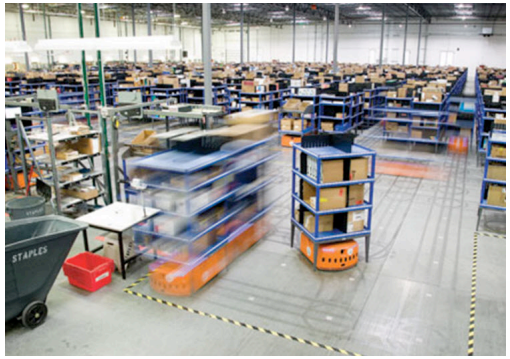
A Classic Graph Optimization Problem

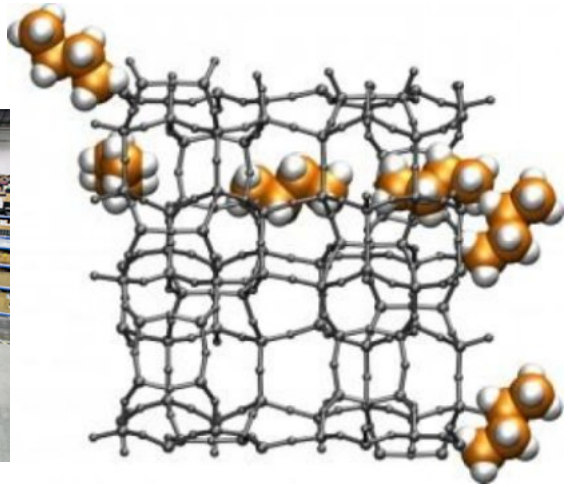
- Shortest path from n1 to n2
  - Shortest sequence of edges such that
    - Source node of first edge is n1
    - Destination of last edge is n2
    - For edges, e1 and e2, in the sequence, if e2 follows e1 in the sequence, the source of e2 is the destination of e1
- Shortest weighted path
  - Minimize the sum of the weights of the edges in the path

Some Shortest Path Problems

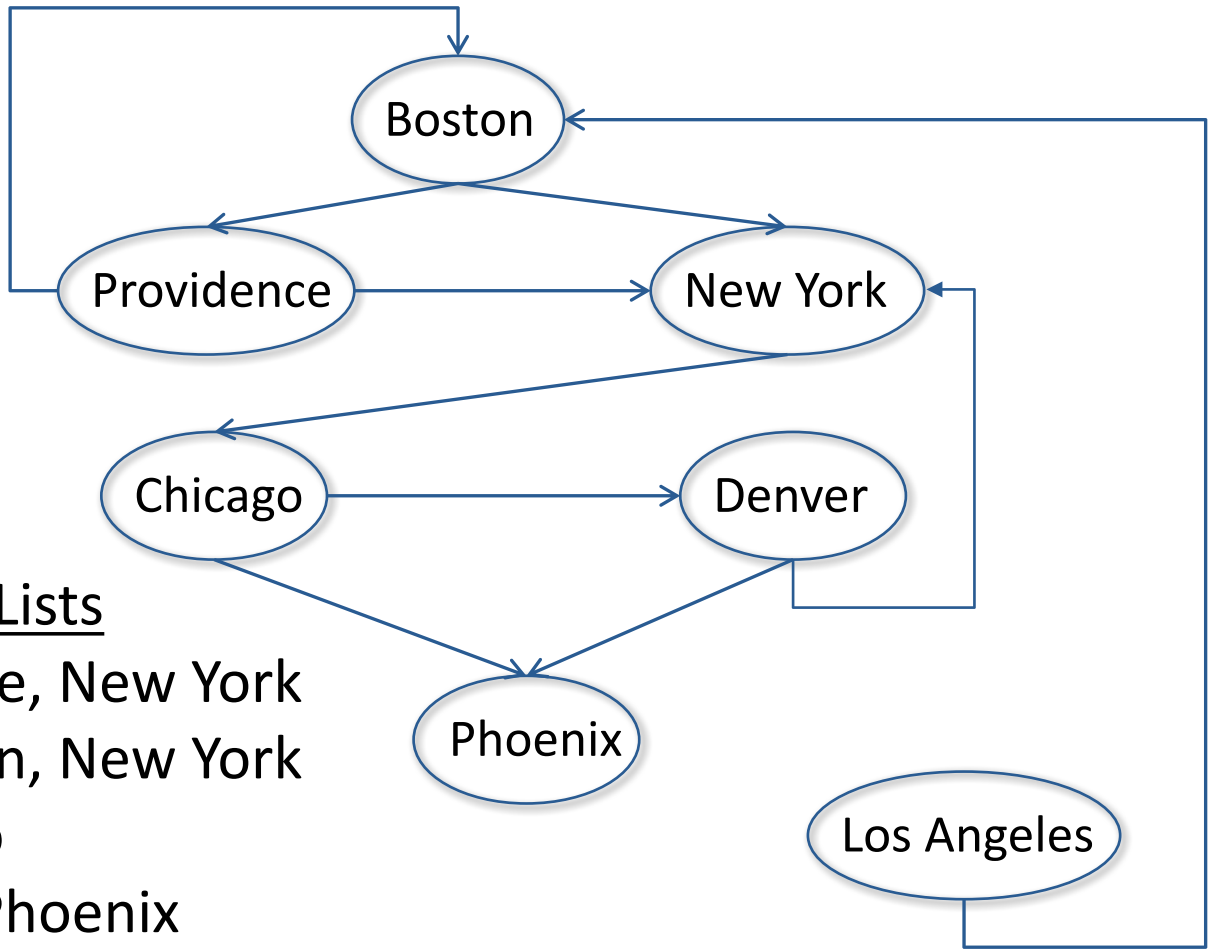
- Finding a route from one city to another
- Designing communication networks
- Logistics of material handling
- Finding a path for a molecule through a chemical labyrinth
- …







An Example



All nodes in graph

Adjacency Lists

Boston: Providence, New York
Providence: Boston, New York
New York: Chicago
Chicago: Denver, Phoenix
Denver: Phoenix, New York
Los Angeles: Boston
Phoenix:

Build the Graph

```
def buildCityGraph(graphType):
    g = graphType()
    for name in ('Boston', 'Providence', 'New York', 'Chicago',
                 'Denver', 'Phoenix', 'Los Angeles'): #Create 7 nodes
        g.addNode(Node(name))
    g.addEdge(Edge(g.getNode('Boston'), g.getNode('Providence')))
    g.addEdge(Edge(g.getNode('Boston'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('Providence'), g.getNode('Boston')))
    g.addEdge(Edge(g.getNode('Providence'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('New York'), g.getNode('Chicago')))
    g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Denver')))
    g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('Los Angeles'), g.getNode('Boston')))
    return g
```

Finding the Shortest Path

- Algorithm 1, **depth-first search** (DFS)
- Similar to left-first depth-first method of enumerating a search tree (Lecture 2)
- Main difference is that graph might have cycles, so we must keep track of what nodes we have visited to avoid going in infinite loops

Note that we are using divide-and-conquer: if we can find a path from a source to an intermediate node, and a path from the intermediate node to the destination, the combination is a path from source to destination

Depth First Search

- Start at an initial node
- Consider all the edges that leave that node, in some order
- Follow the first edge, and check to see if at goal node
- If not, repeat the process from new node
- Continue until either find goal node, or run out of options
  - When run out of options, backtrack to the previous node and try the next edge, repeating this process

Depth First Search (DFS)

```
def DFS(graph, start, end, path, shortest, toPrint = False):
    path = path + [start]
    if toPrint:
        print('Current DFS path:', printPath(path))
    if start == end:
        if toPrint:
            print('Found path:', printPath(path))
        return path
    for node in graph.childrenOf(start):
        if node not in path: #avoid cycles
            if shortest == None or len(path) < len(shortest):
                newPath = DFS(graph, node, end, path, shortest, toPrint)
                if newPath != None:
                    shortest = newPath
            elif toPrint:
                print('Already visited', node)
    return shortest
```

initially None

... popping recursion stack to try next node

Note that it explores all paths through first node, before ...

Shortest path cannot contain a cycle

DFS called from a wrapper function

Gets recursion started properly

Provides appropriate abstraction

Five Minute Break

How do you fix a broken pumpkin?

Five Minute Break

How do you fix a broken pumpkin?

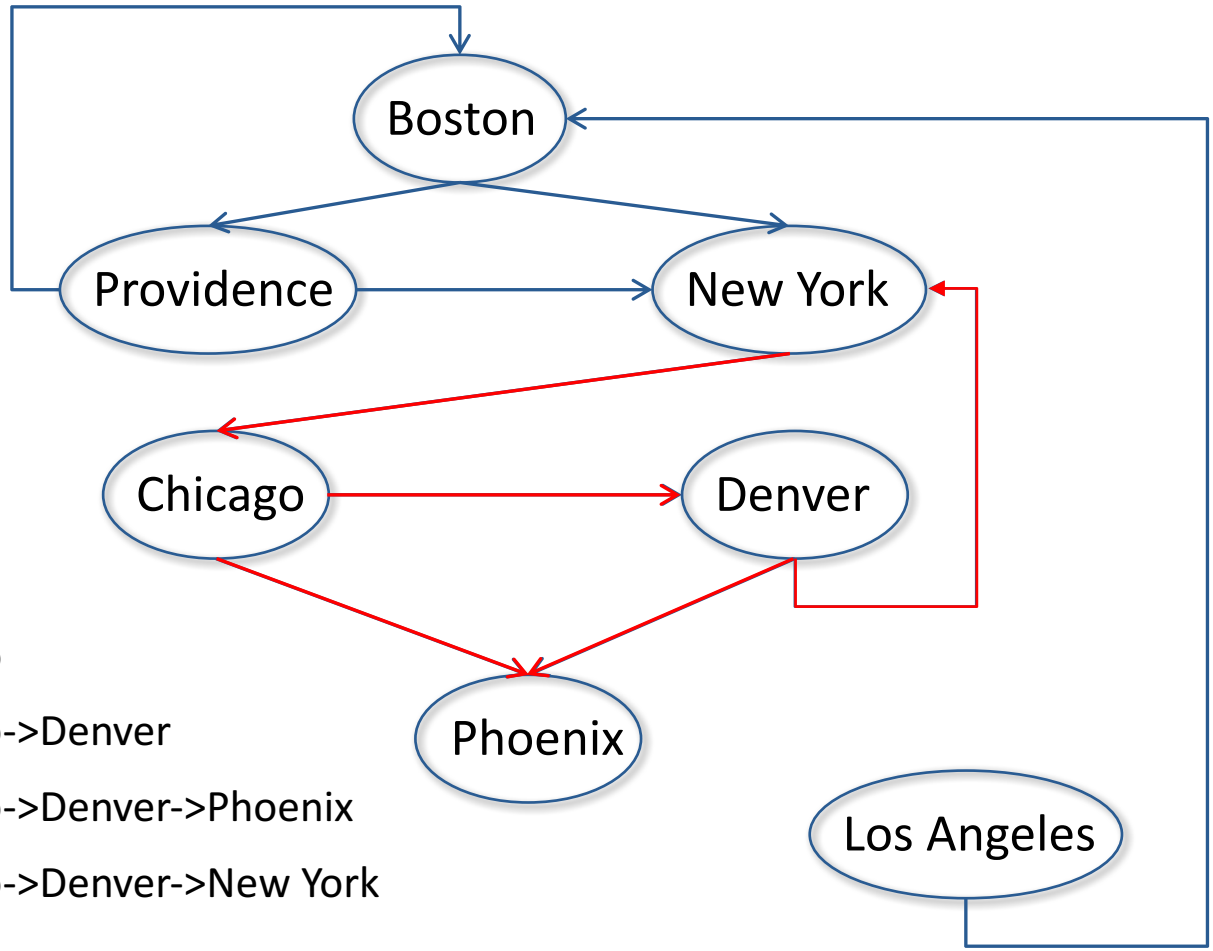
With a pumpkin patch, of course.

Test DFS

```
def shortestPath(g, start, end, alg, toPrint = True):
    """Assumes graph is a Digraph; start and end are nodes
       Returns a shortest path from start to end in graph"""
    sp = alg(g, g.getNode(start), g.getNode(end), toPrint)
    if sp != None:
        print('Shortest path from', start, 'to',
              end, 'is', printPath(sp))
    else:
        print('There is no path from', start, 'to', end)

G = buildCityGraph(Digraph)
shortestPath(G, 'Chicago', 'Boston', dfsRec, False)
```

Output (Chicago to Boston)



Current DFS path: Chicago

Current DFS path: Chicago->Denver

Current DFS path: Chicago->Denver->Phoenix

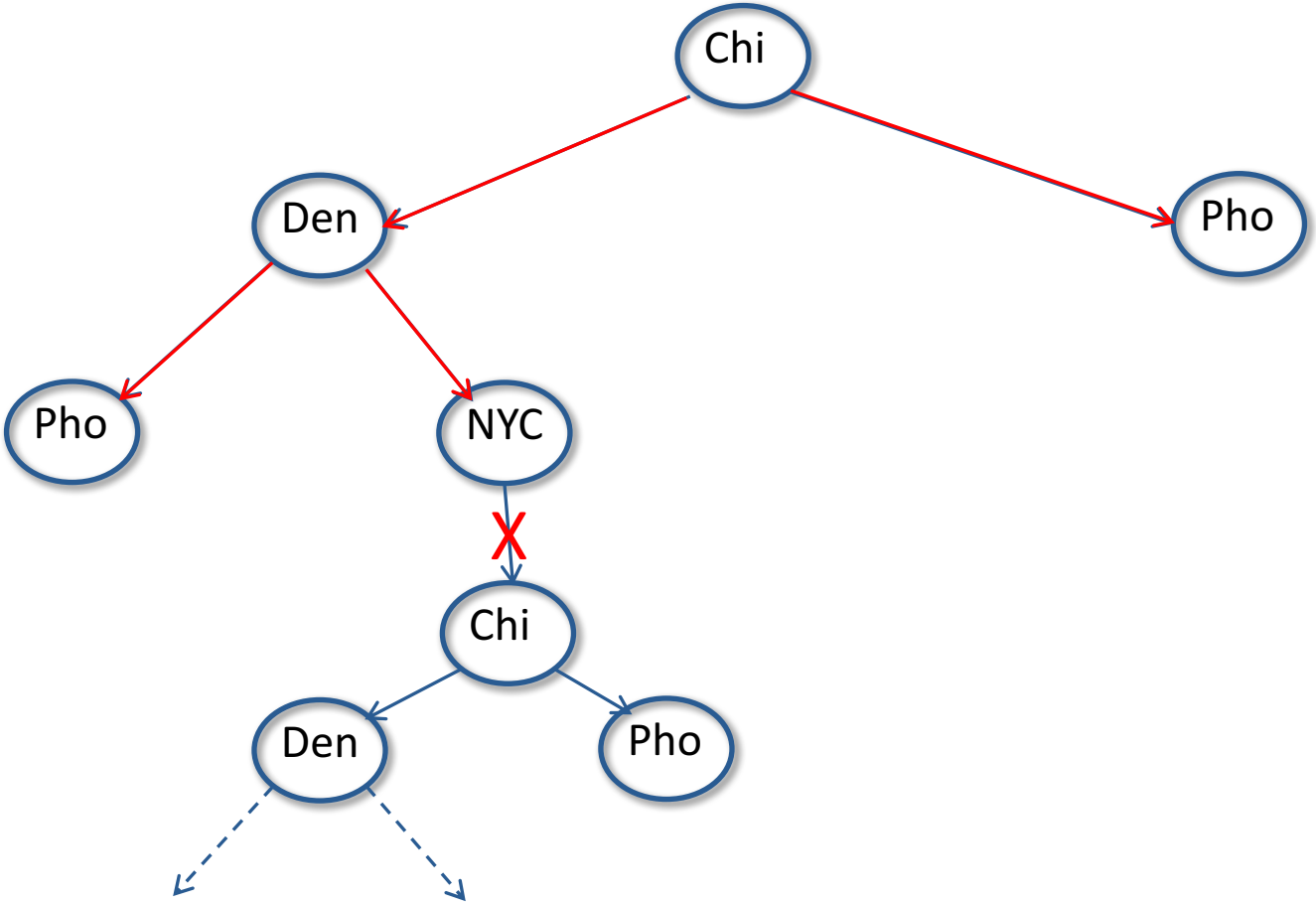
Current DFS path: Chicago->Denver->New York

Already visited Chicago

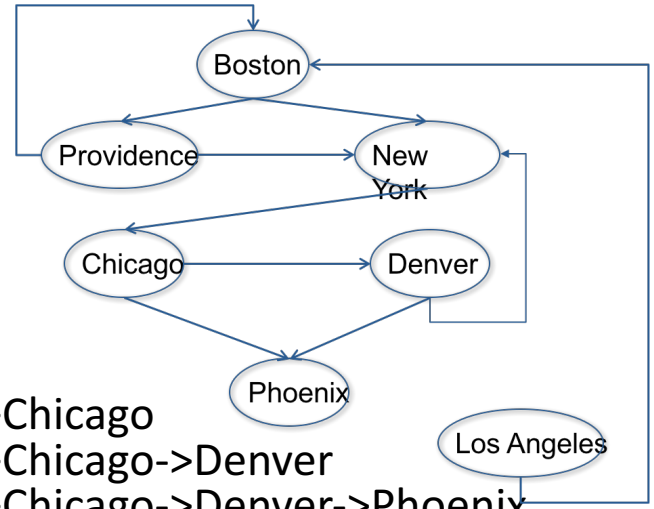
Current DFS path: Chicago->Phoenix

There is no path from Chicago to Boston

Visualizing as a tree search



Output (Boston to Phoenix)



Current DFS path: Boston
Current DFS path: Boston->Providence
Already visited Boston
Current DFS path: Boston->Providence->New York
Current DFS path: Boston->Providence->New York->Chicago
Current DFS path: Boston->Providence->New York->Chicago->Denver
Current DFS path: Boston->Providence->New York->Chicago->Denver->Phoenix
Found path: Boston->Providence->New York->Chicago->Denver->Phoenix
Already visited New York
Current DFS path: Boston->Providence->New York->Chicago->Phoenix
Found path: Boston->Providence->New York->Chicago->Phoenix
Current DFS path: Boston->New York
Current DFS path: Boston->New York->Chicago
Current DFS path: Boston->New York->Chicago->Denver
Current DFS path: Boston->New York->Chicago->Denver->Phoenix
Found path: Boston->New York->Chicago->Denver->Phoenix
Already visited New York
Current DFS path: Boston->New York->Chicago->Phoenix
Found path: Boston->New York->Chicago->Phoenix
Shortest path from Boston to Phoenix is Boston->New York->Chicago->Phoenix

Can Also Be Implemented Iteratively

- Instead of keeping track of where we are with stack of recursive call, use an explicit **LIFO** (last-in-first-out) data structure
- List of pairs (current vertex, path so far)
  - Push by calling list.append
  - Pop by calling list.pop

Try a Bigger Graph

```
def genGraph(GraphType, numNodes, numEdges):
    G = GraphType()
    for node in range(numNodes):
        G.addNode(Node(str(node)))
    for edge in range(numEdges):
        source = G.getNode(str(random.randint(0, numNodes-1)))
        destination = G.getNode(str(random.randint(0, numNodes-1)))
        G.addEdge(Edge(source, destination))
    return G

G = genGraph(Digraph, 100, 300)
shortestPath(G, '1', '99', dfsRec, False)
```

Why So Slow?

- If there are multiple paths from the start node to another node, it computes the shortest path from that node to the end node multiple times?
- Not an inherent feature of DFS
- Possible to implement it far more efficiently, e.g., by using a memo

Breadth First Search

- Start at an initial node
- Consider all the edges that leave that node, in some order
- Follow the first edge, and check to see if at goal node
- If not, try the next edge from the current node
- Continue until either find goal node, or run out of options
  - When run out of edge options, move to next node at same distance from start, and repeat
  - When run out of node options, move to next level in the graph (all nodes one step further from start), and repeat

Breadth-first Search (BFS)

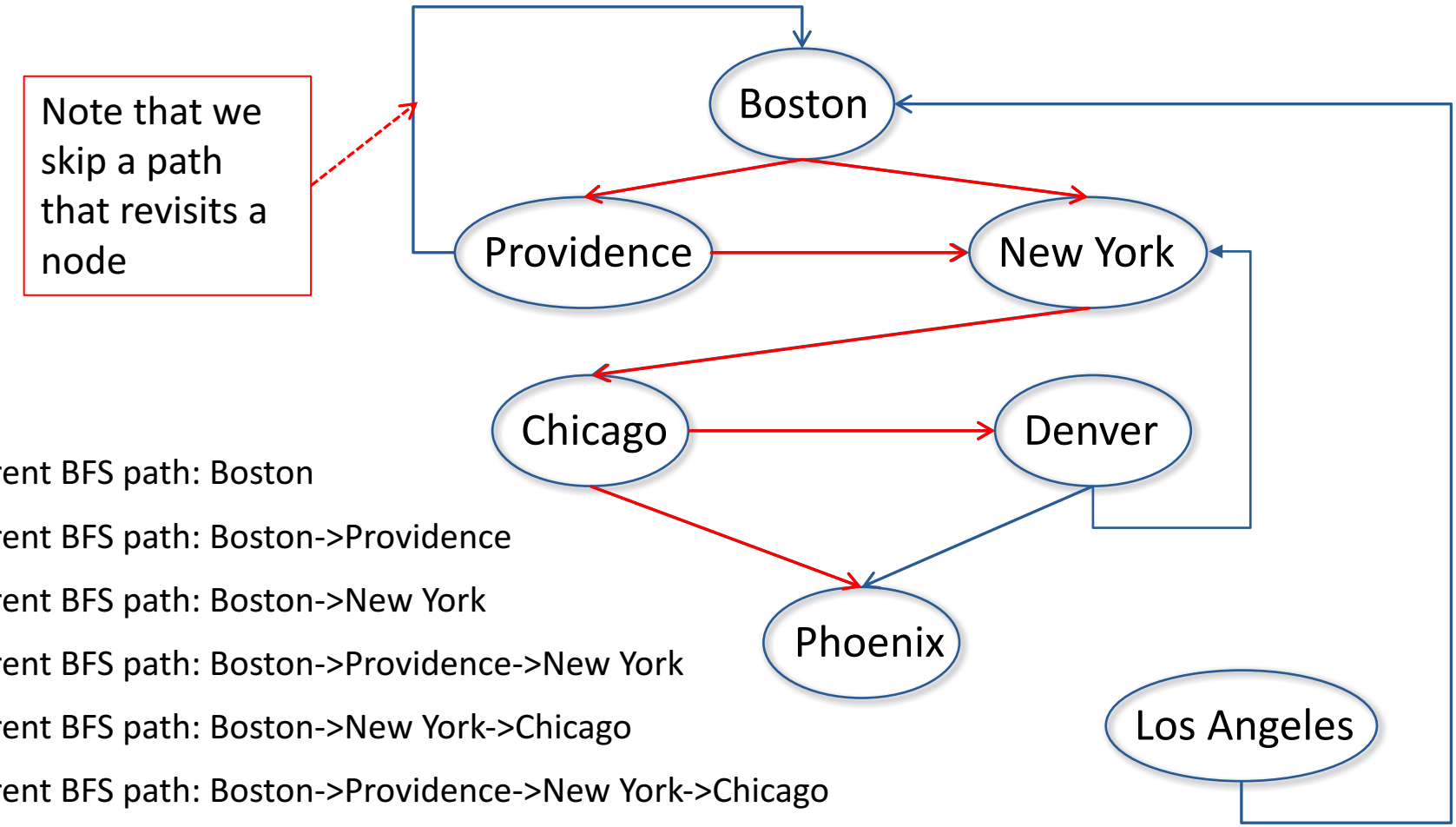
```
def BFS(graph, start, end, toPrint = False):
    initPath = [start]
    pathQueue = [initPath] #FIFO
    while len(pathQueue) != 0:
        #Get and remove oldest element in pathQueue
        tmpPath = pathQueue.pop(0)
        if toPrint:
            print('Current BFS path:', printPath(tmpPath))
        lastNode = tmpPath[-1]
        if lastNode == end:
            return tmpPath
        for nextNode in graph.childrenOf(lastNode):
            if nextNode not in tmpPath:
                newPath = tmpPath + [nextNode]
                pathQueue.append(newPath)
    return None
```

First-in-first-out

Why is it ok to stop?

Explore all paths with n hops before exploring any path with more than n hops

Output (Boston to Phoenix)



Current BFS path: Boston

Current BFS path: Boston->Providence

Current BFS path: Boston->New York

Current BFS path: Boston->Providence->New York

Current BFS path: Boston->New York->Chicago

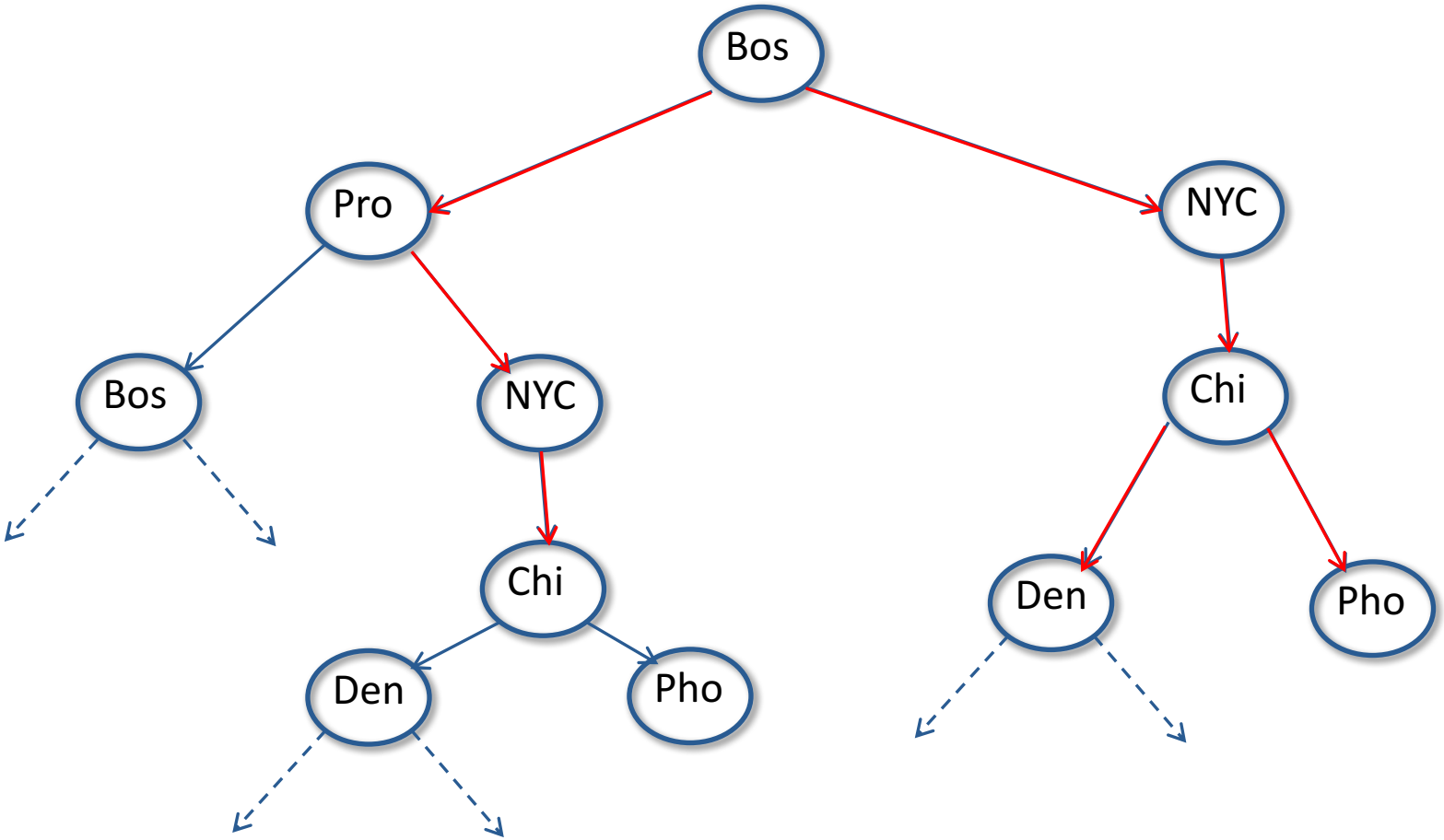
Current BFS path: Boston->Providence->New York->Chicago

Current BFS path: Boston->New York->Chicago->Denver

Current BFS path: Boston->New York->Chicago->Phoenix

Shortest path from Boston to Phoenix is Boston->New York->Chicago->Phoenix

Visualizing as a tree search



Still have the problem of visiting same node multiple times

What About a Weighted Shortest Path

- Want to minimize the sum of the weights of the edges, not the number of edges
- DFS can be easily modified to do this, if we assume that weights are non-negative numbers
- BFS cannot, since shortest weighted path may have more than the minimum number of hops

Recap

- Graphs are cool
  - Best way to create a model of many things
    - Capture relationships among objects
  - Many important problems can be posed as graph optimization problems we already know how to solve
- Depth-first and breadth-first search are important algorithms
  - Can be used to solve many problems

Get Ready for Halloween Microquiz