Graph-theoretic Models, Lecture 3, Segment 1

John Guttag

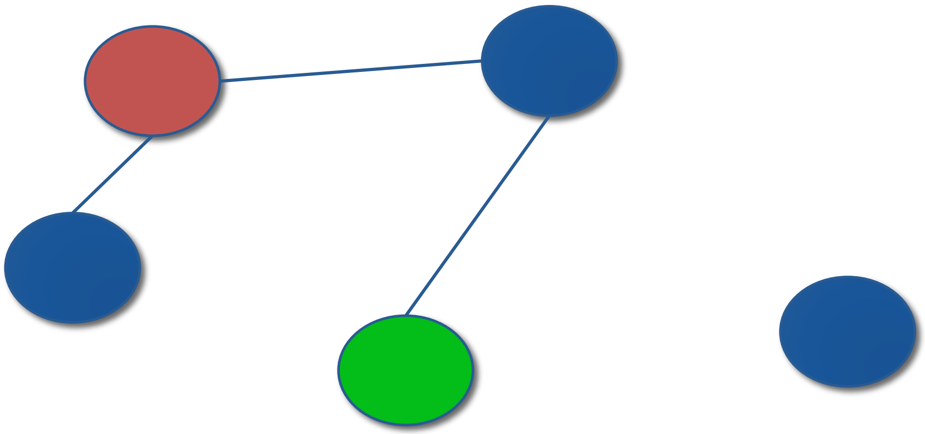
MIT Department of Electrical Engineering and Computer Science

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
- Now want to look at class of models called graphs

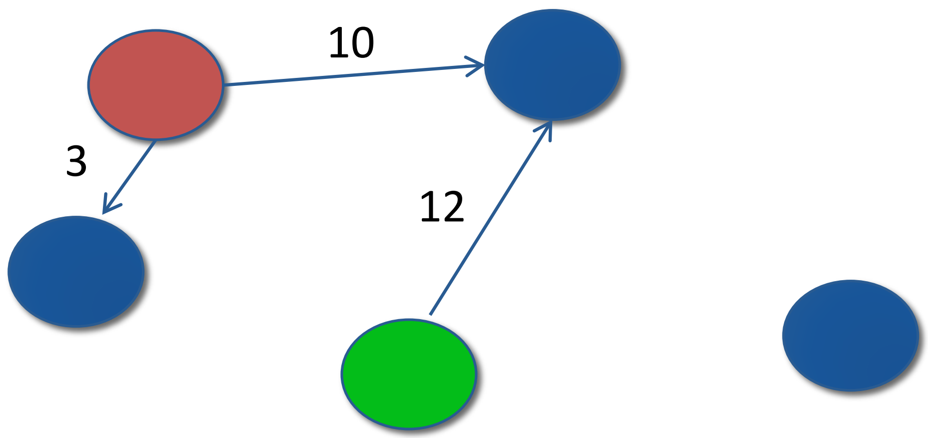
What's a Graph?

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



What's a Graph?

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

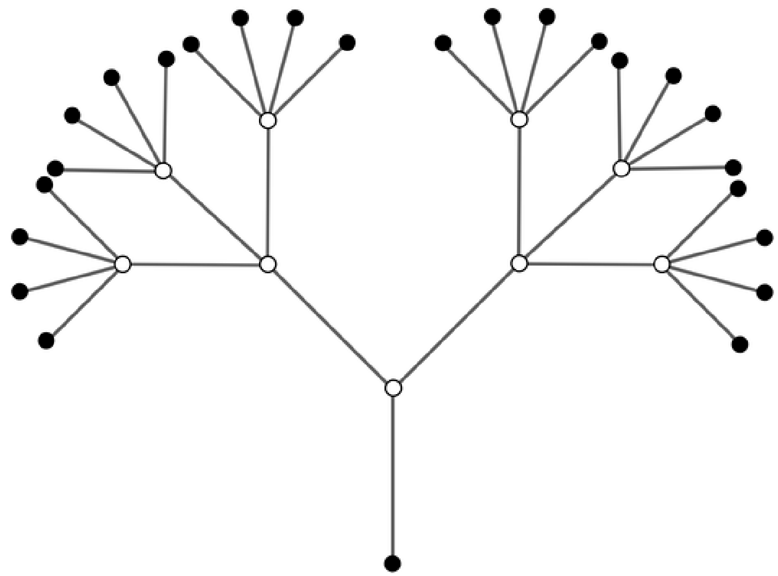


Why Graphs?

- To capture useful relationships among entities
  - Rail links between Paris and London
  - How the atoms in a molecule related to one another
  - Ancestral relationships

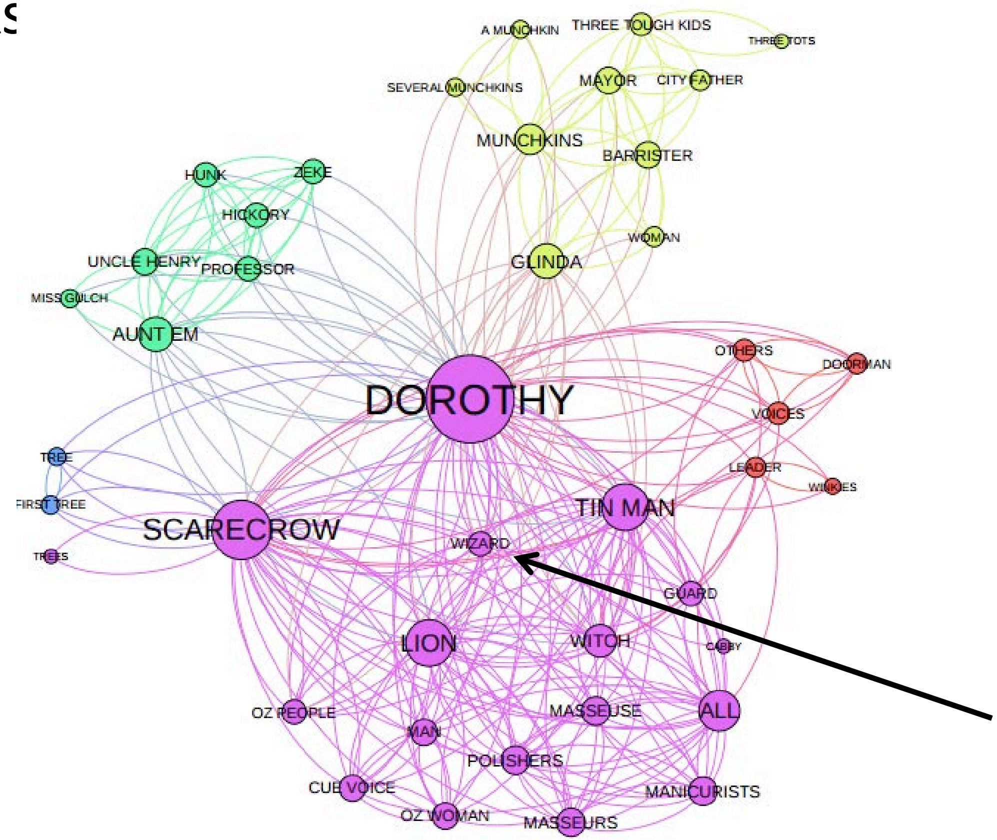
Trees: An Important Special Case

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



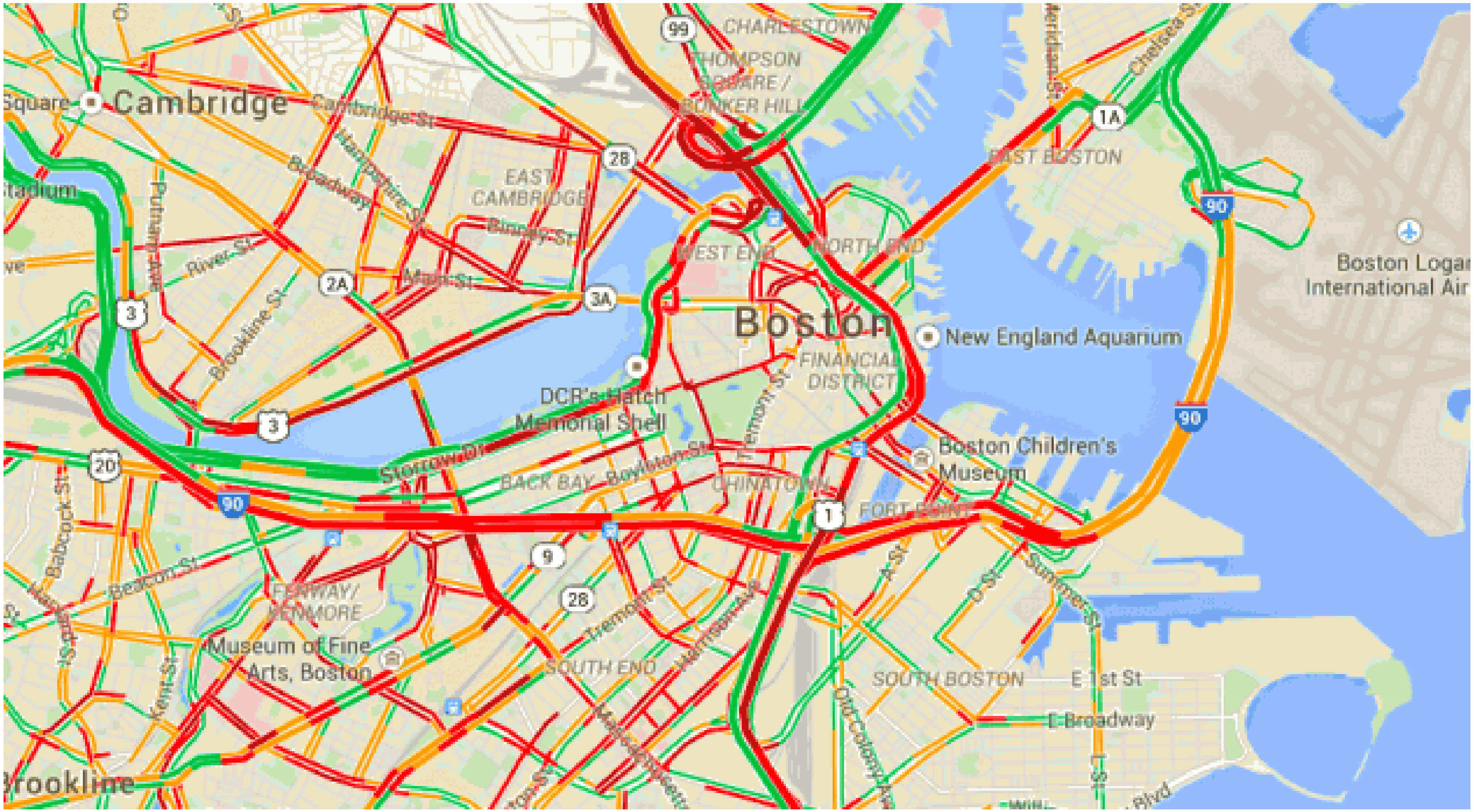
Why Graphs Are So Useful

- World is full of networks based on relationships
  - Computer networks
  - Transportation networks
  - Financial networks
  - Sewer networks
  - Political networks
  - Criminal networks
  - Social networks
  - Etc.



mapr.com

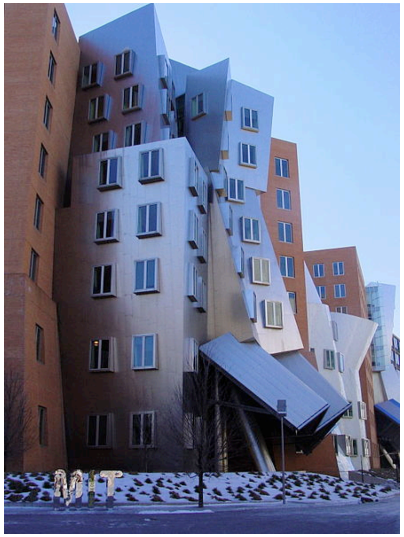
Graph Theory Saves Me Time Every Day



www.google.com

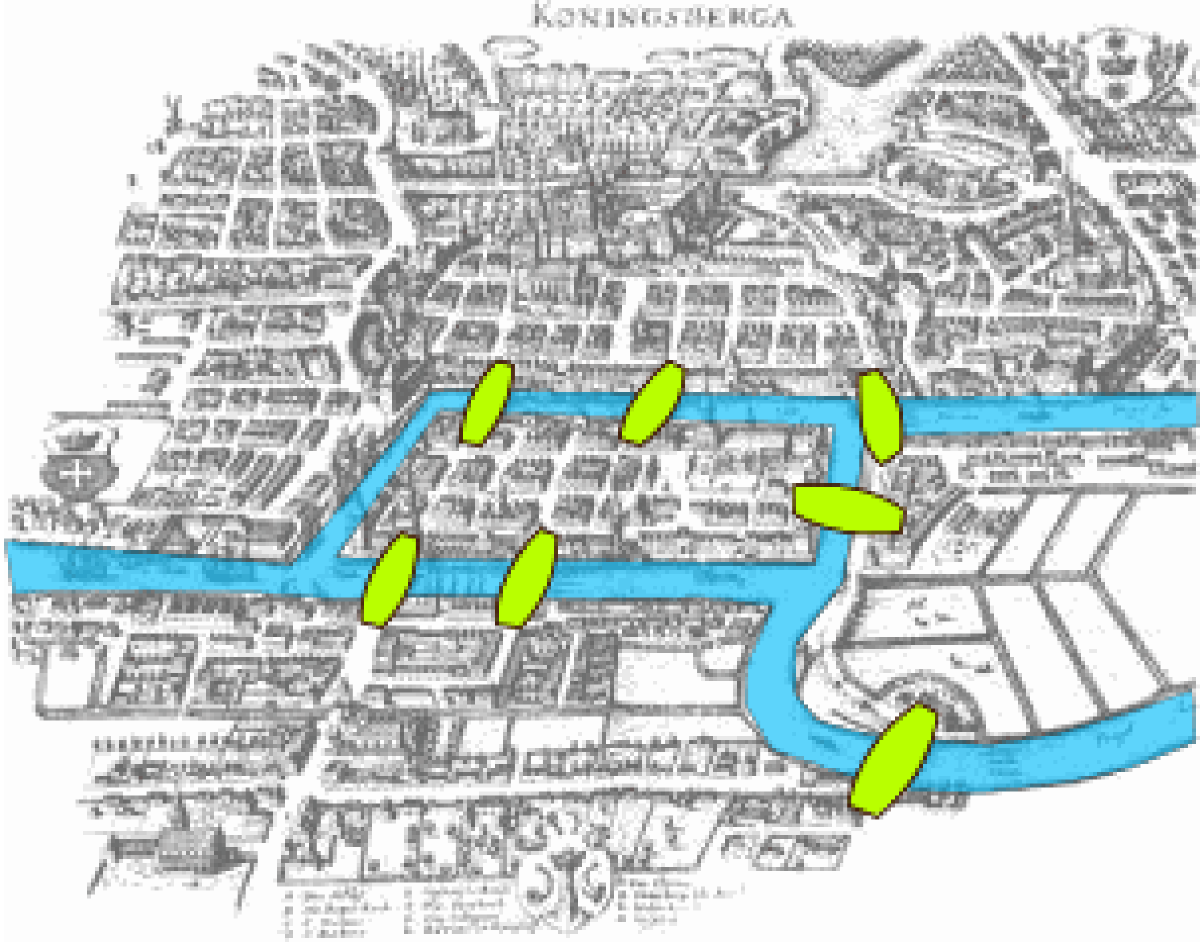
Getting John to the Office

- Model road system using a digraph
  - Nodes: points where roads end or meet
  - Edges: connections between points
    - Each edge has a weight indicating time it will take to get from source node to destination node for that edge
- Solve a graph optimization problem
  - Shortest weighted path between my house and my office



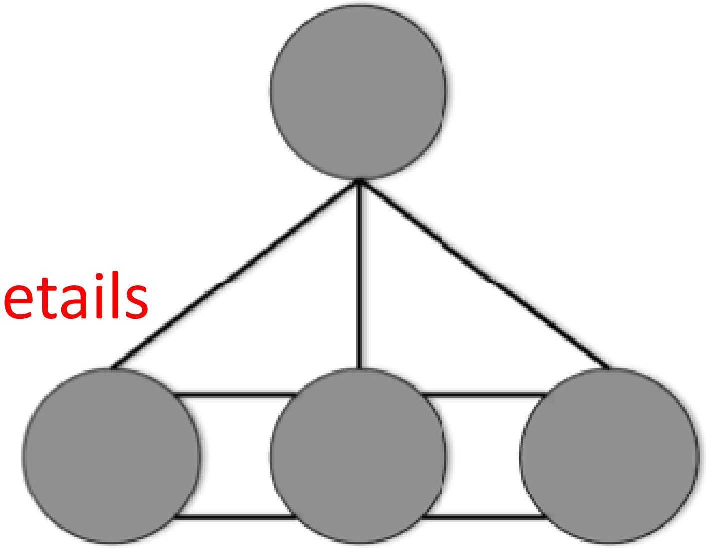
First Reported Use of Graph Theory

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



Leonhard Euler's Model

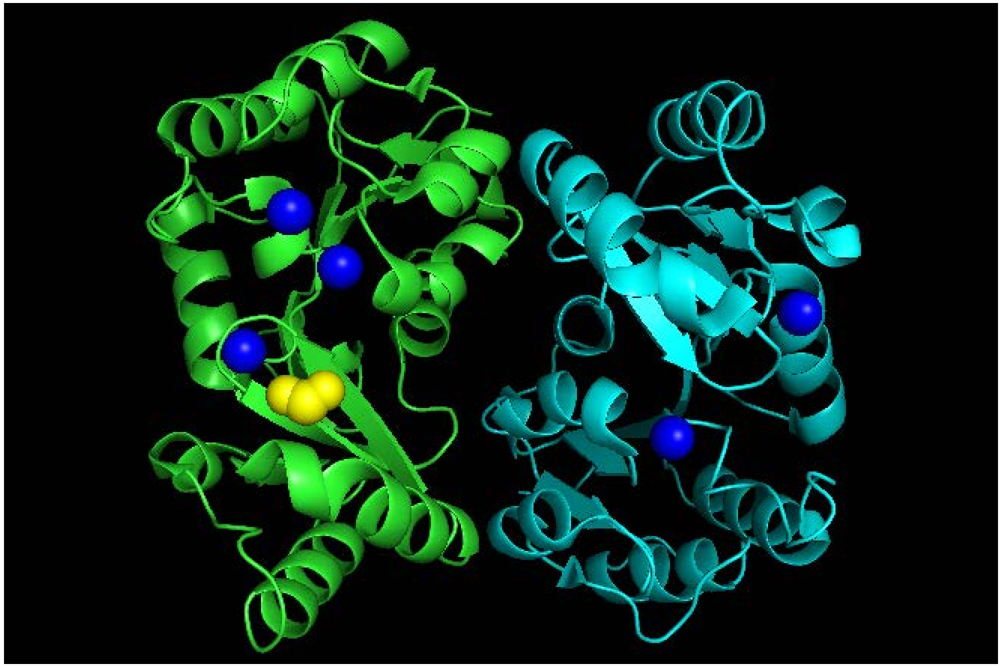
- Each island a node
- Each bridge an undirected edge
- Model abstracts away irrelevant details
  - Size of islands
  - Length of bridges



- Is there a path that contains each edge exactly once?

Next Segment

- Implementing graphs
- Some classic graph optimization problems



Graph-theoretic Models, Lecture 3, Segment 2

John Guttag

MIT Department of Electrical Engineering and Computer Science

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

- Adjacency matrix
  - Rows: source nodes
  - Columns: destination nodes
  - Cell[s, d] = 1 if there is an edge from s to d

    0 otherwise
- 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)
```

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)
```

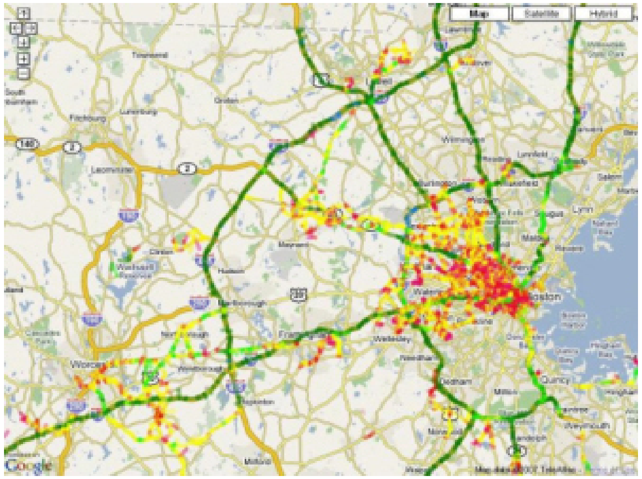
- Why is Graph a subclass of digraph?
- Remember the substitution rule from 6.00.1x?
  - If client code works correctly using an instance of the supertype, it should also work correctly when an instance of the subtype is substituted for the instance of the supertype
- Any program that works with a Digraph will also work with a Graph (but not *vice versa*)

A Classic Graph Optimization Problem

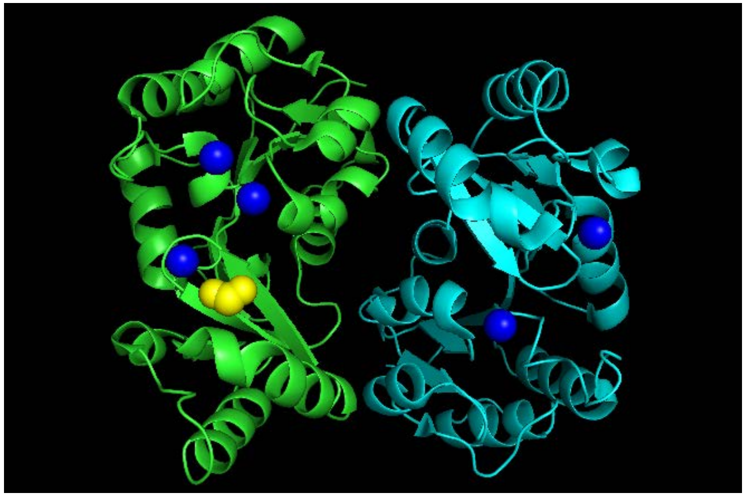
- Shortest path from n1 to n2
  - Shortest sequence of edges such that
    - Source node of first edge is n1
    - Destination of last edge is n2
    - For edges, e1 and e2, in the sequence, if e2 follows e1 in the sequence, the source of e2 is the destination of e1
- Shortest weighted path
  - Minimize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

Some Shortest Path Problems

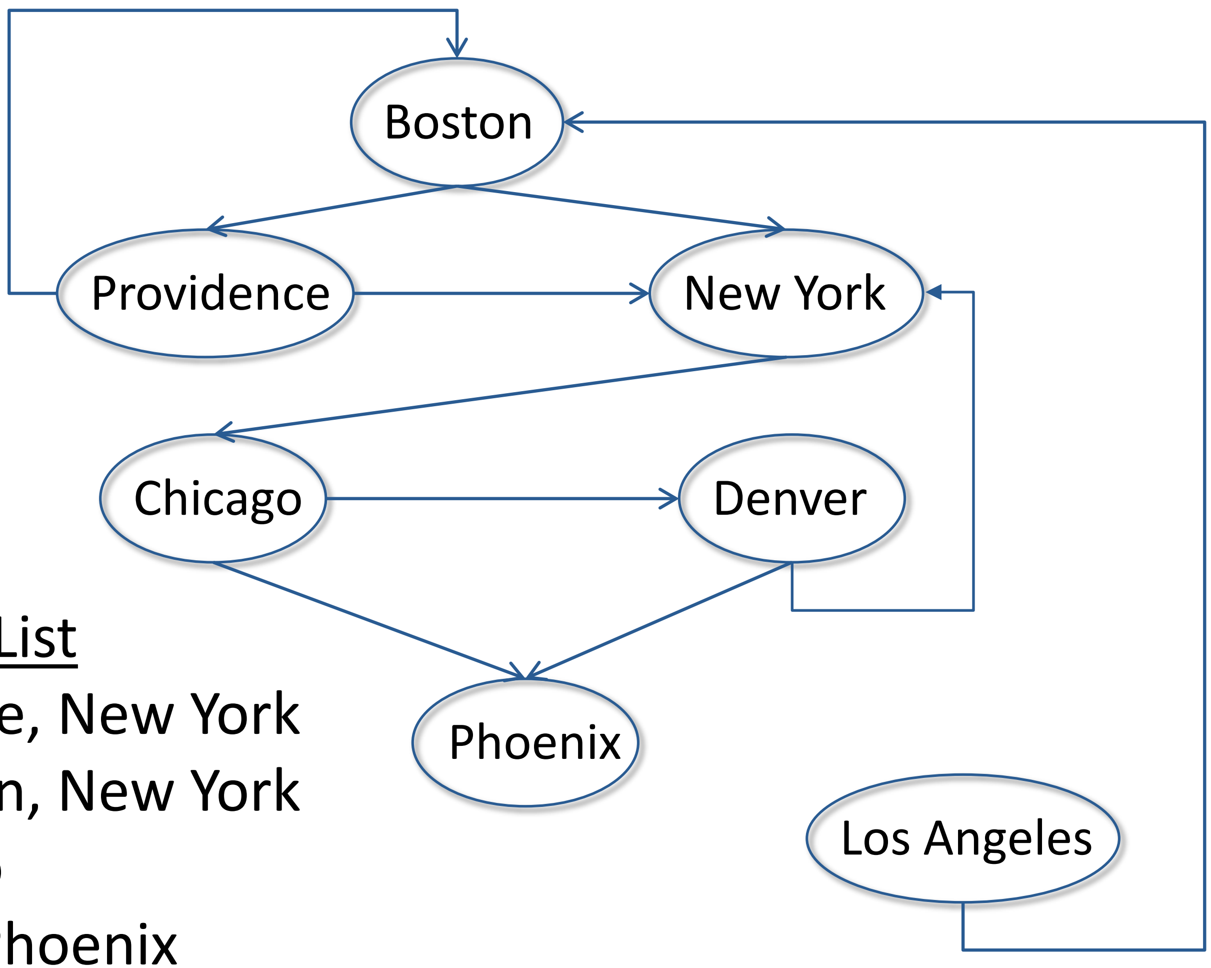
- Finding a route from one city to another
- Designing communication networks
- Finding a path for a molecule through a chemical labyrinth
- …



www.google.com



An Example



Adjacency List

Boston: Providence, New York

Providence: Boston, New York

New York: Chicago

Chicago: Denver, Phoenix

Denver: Phoenix, New York

Los Angeles: Boston

Build the Graph

```
def buildCityGraph():
    g = Digraph()
    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('Denver'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode(’New York')))
    g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Los Angeles'), g.getNode('Boston')))
```

Coming Up

- Solutions to shortest path problem

Graph-theoretic Models, Lecture 3, Segment 3

John Guttag

MIT Department of Electrical Engineering and Computer Science

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

Depth First Search (DFS)

```
def DFS(graph, start, end, path, shortest):
    path = path + [start]
    if start == end:
        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
    return shortest

def shortestPath(graph, start, end):
    return DFS(graph, start, end, [], None, toPrint)
```

DFS called from a wrapper function: shortestPath

Gets recursion started properly

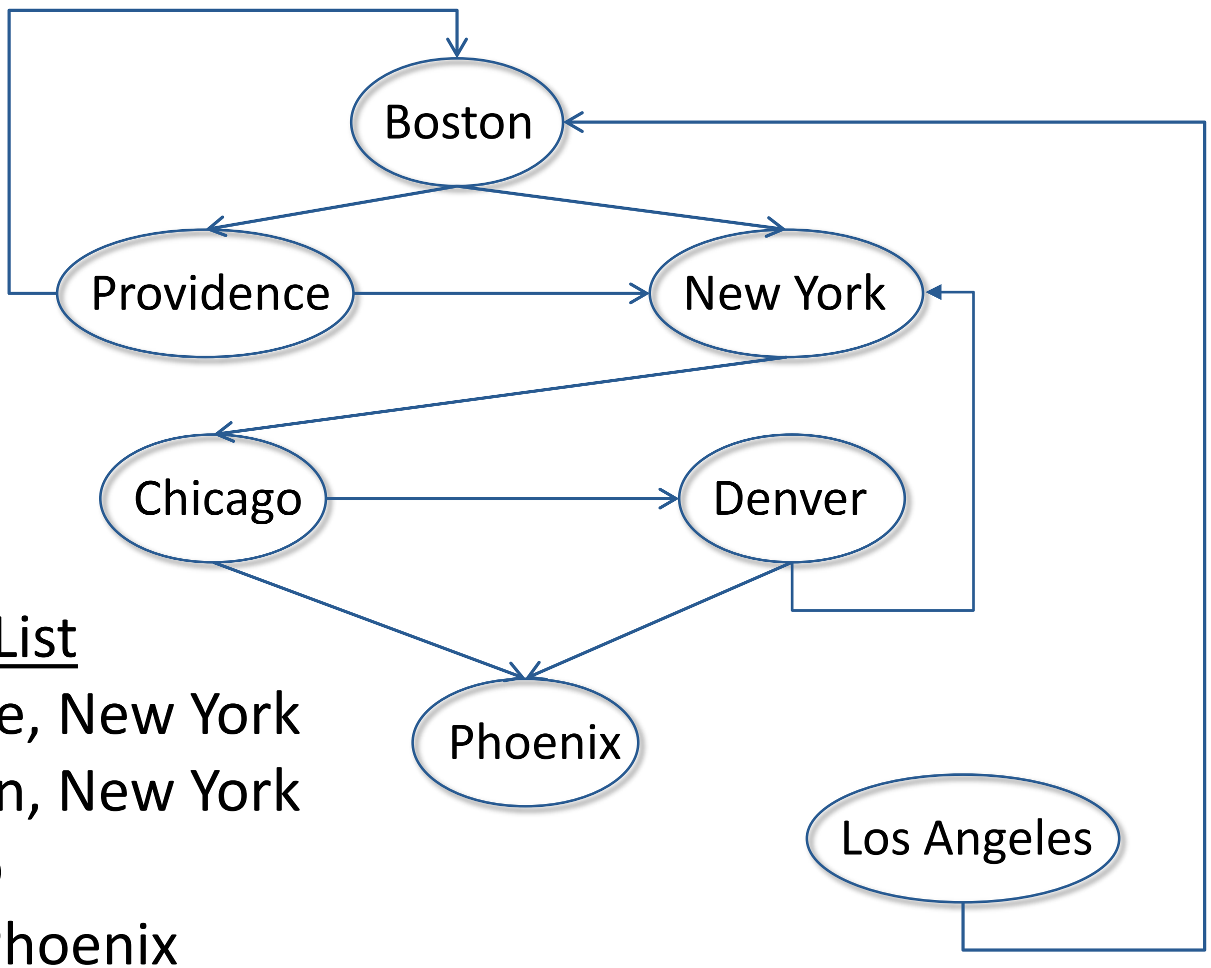
Provides appropriate abstraction

Test DFS

```
def testSP(source, destination):
    g = buildGraph()
    sp = shortestPath(g, g.getNode(source), g.getNode(destination)
    if sp != None:
        print('Shortest path from', source, 'to',
              destination, 'is', printPath(sp))
    else:
        print('There is no path from', source, 'to', destination)

testSP('Boston', ’Chicago')
```

An Example



Adjacency List

Boston: Providence, New York

Providence: Boston, New York

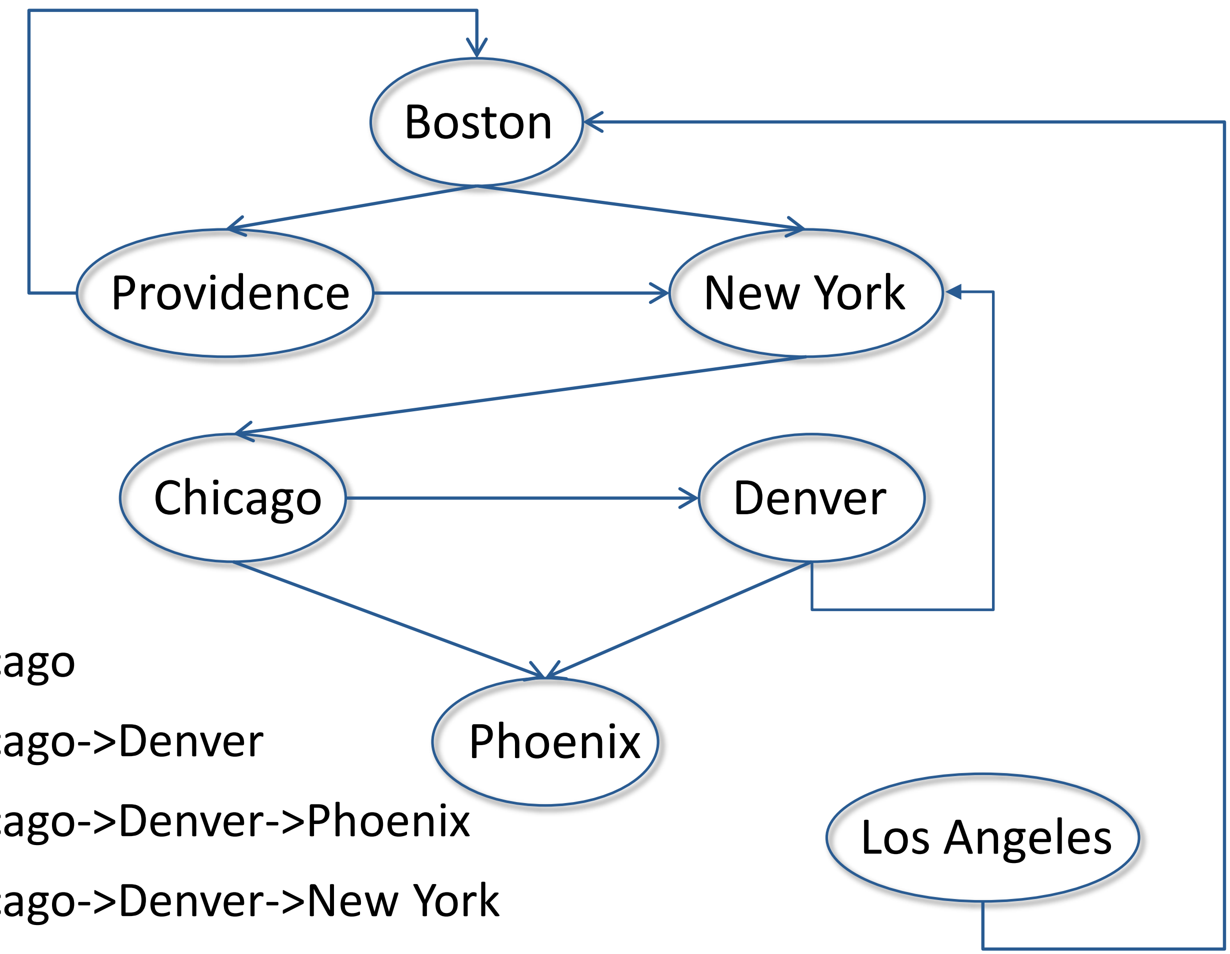
New York: Chicago

Chicago: Denver, Phoenix

Denver: Phoenix, New York

Los Angeles: Boston

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

There is no path from Chicago to Boston

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

Already visited New York

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 a shorter path

Already visited New York

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

Algorithm 2: Breadth-first Search (BFS)

```
def BFS(graph, start, end, toPrint = False):
    initPath = [start]
    pathQueue = [initPath]
    if toPrint:
        print('Current BFS path:', printPath(pathQueue))
    while len(pathQueue) != 0:
        #Get and remove oldest element in pathQueue
        tmpPath = pathQueue.pop(0)
        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
```

?

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

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
- 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

Coming Up

- Modeling situations with unpredictable events
- Will make heavy use of plotting
  - Lecture 4 is about plotting in Python
  - Identical to a lecture in 6.00.1x, feel free to skip it if you took 6.00.1x