

Graph-theoretic Models

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

John Guttag

MIT Department of Electrical Engineering and
Computer Science

Last Week's Brain Twister

```
def getSomething(n):  
    return [p for p in range(2, n+1)\  
            if 0 not in [p%d for d in range(2, p)]]  
  
print([x for x in range(101) if x not in\  
      ({i+1:getSomething(101)[i]\  
       for i in range(len(getSomething(101)))}).values()]])
```

Last Week's Brain Twister

```
def getSomething(n):
    return [p for p in range(2, n+1)\
            if 0 not in [p%d for d in range(2, p)]]

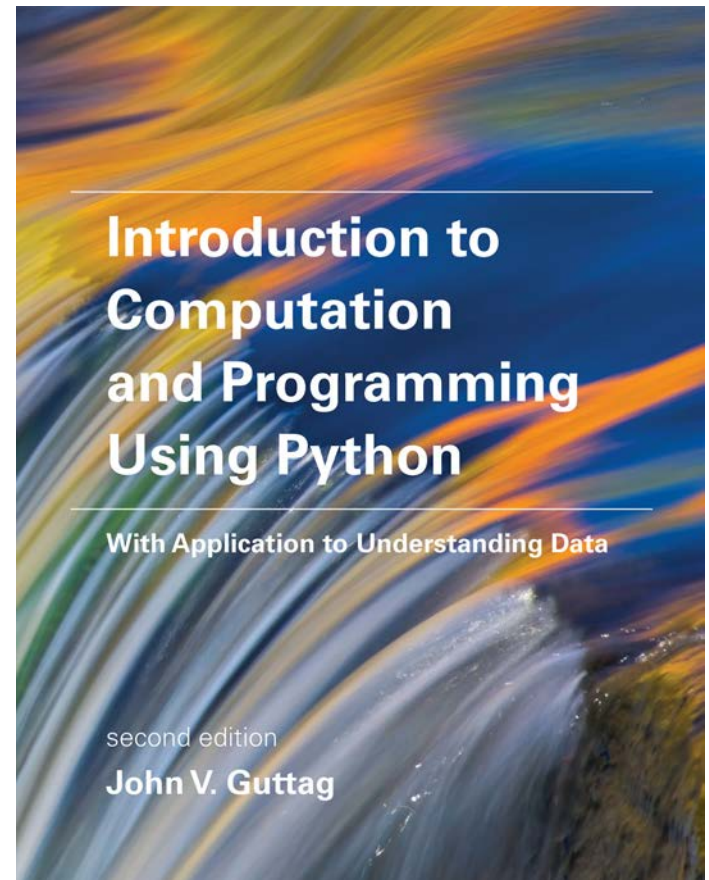
print([x for x in range(101) if x not in\
      ({i+1:getSomething(101)[i]\
       for i in range(len(getSomething(101)))}).values()]])

def getPrimesToN(n):
    def isPrime(p):
        remainders = []
        for d in range(2, p):
            remainders.append(p%d)
        return 0 not in remainders
    primes = []
    for p in range(2, n+1):
        if isPrime(p):
            primes.append(p)
    return primes

D = {}
for i in range(len(getPrimesToN(100))):
    D[i+1] = getPrimesToN(100)[i]
notP = {x for x in range(101) if x not in D.values()}
print(notP)
```

Relevant Reading

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



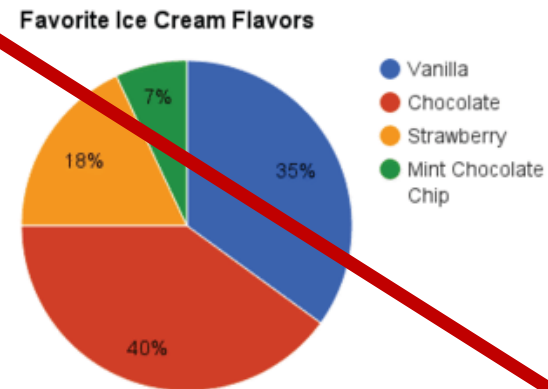
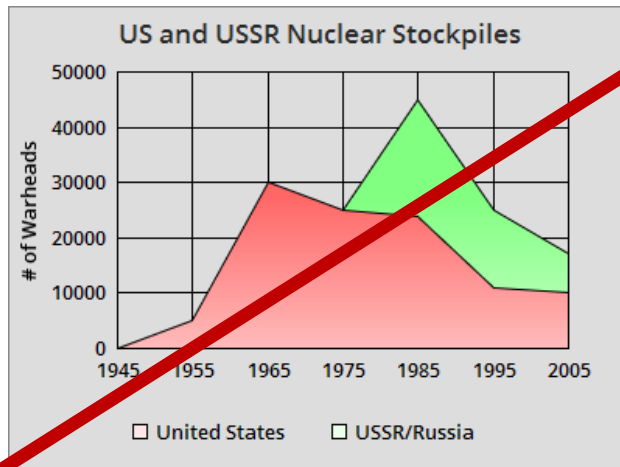
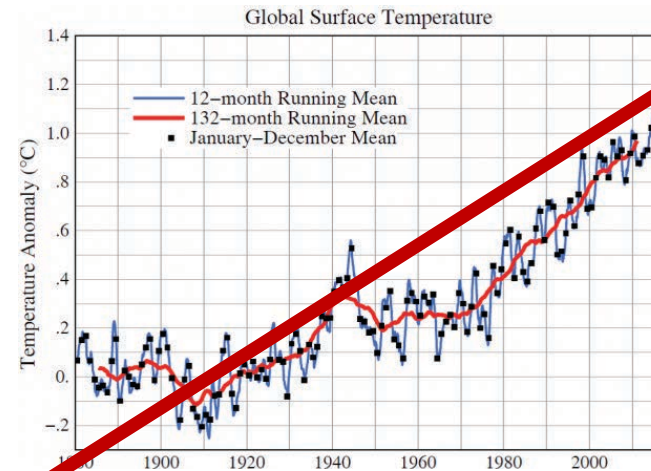
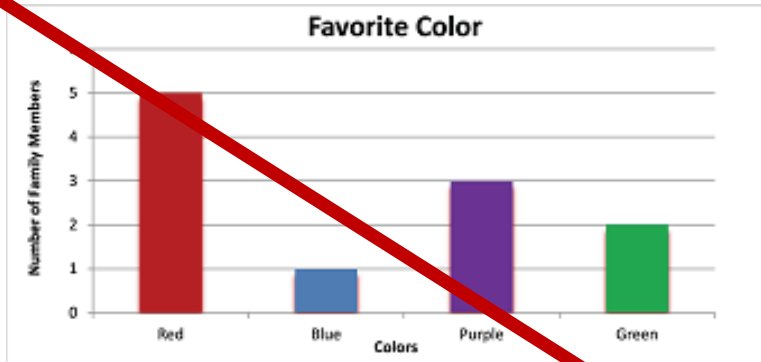
A Reminder

- First micro-quiz in lecture on Wednesday
- Will start around 3:50

Computational Models

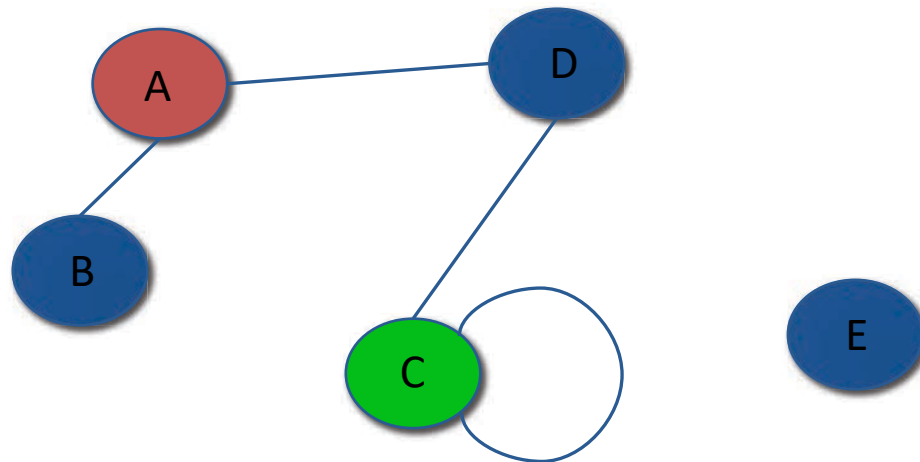
- Programs that help us understand the world and solve practical problems
- Saw how we could map the informal problem of choosing what to eat into an optimization problem, and how we could design a program to solve it
 - Saw how a decision tree can help find a good solution to an optimization problem
- 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?



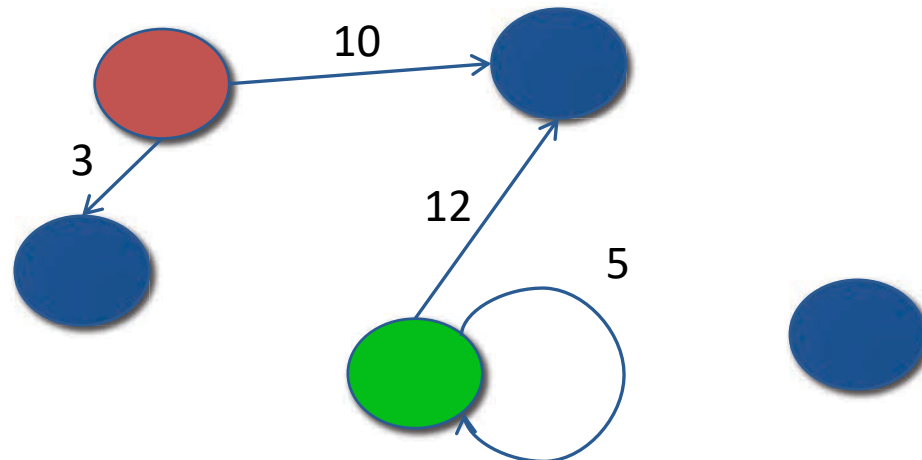
What is a Graph?

- 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



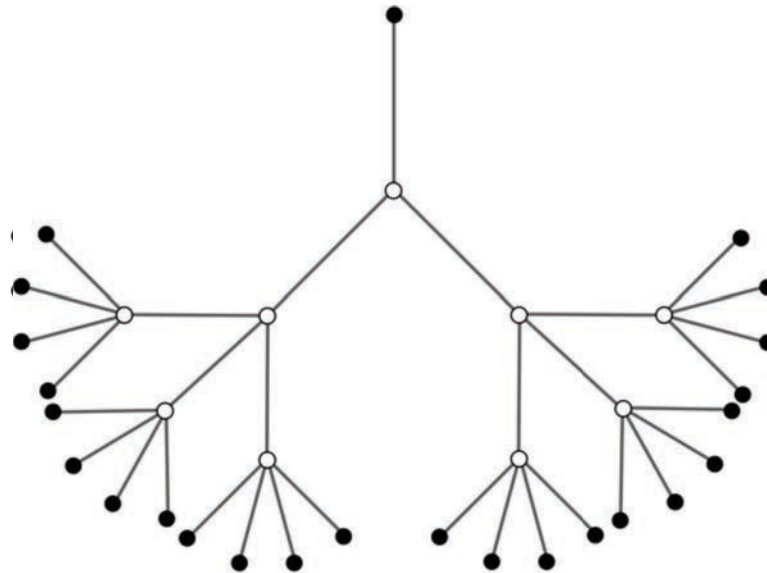
What is a Graph?

- 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



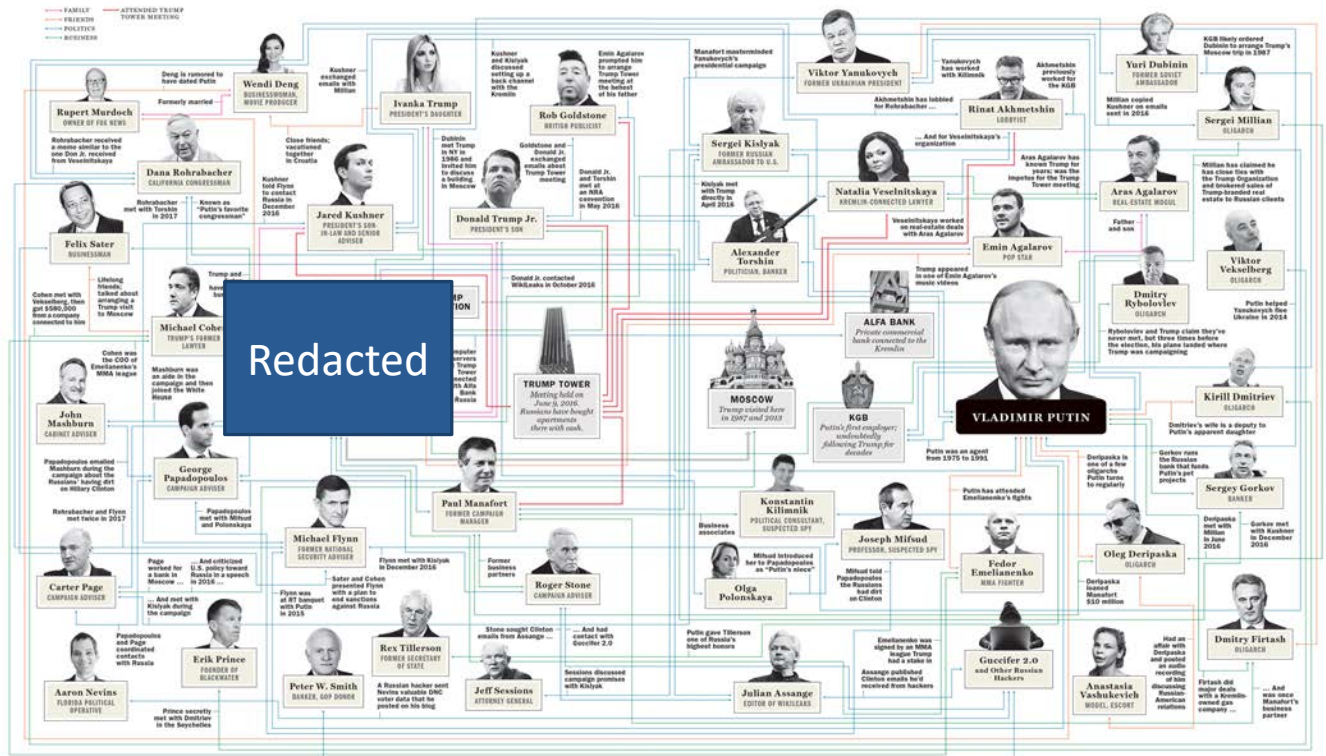
Trees: An Important Special Case

- 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?

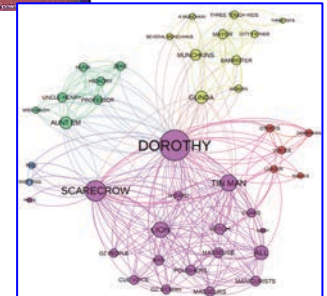
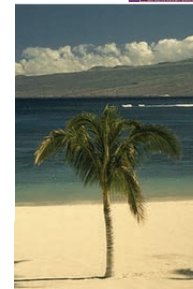
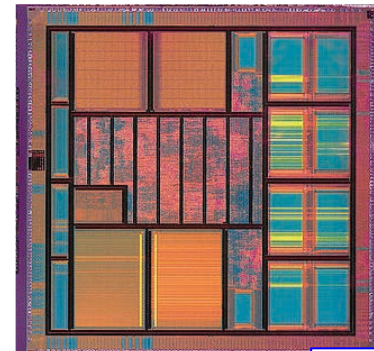
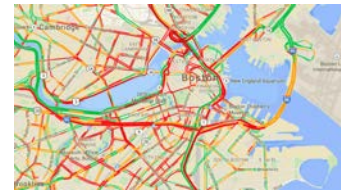
- To capture relationships among entities
 - Rail links between Paris and London
 - How the atoms in a molecule are related to one another
 - Business/social/political connections
 - ...



New York Magazine

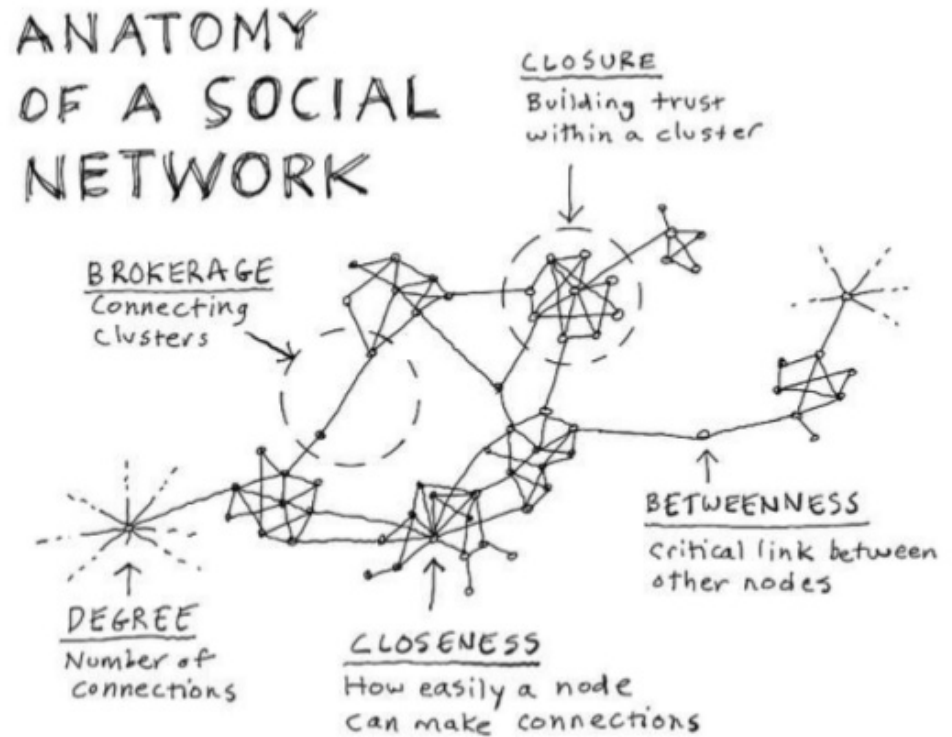
Why Graphs Are So Useful

- 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

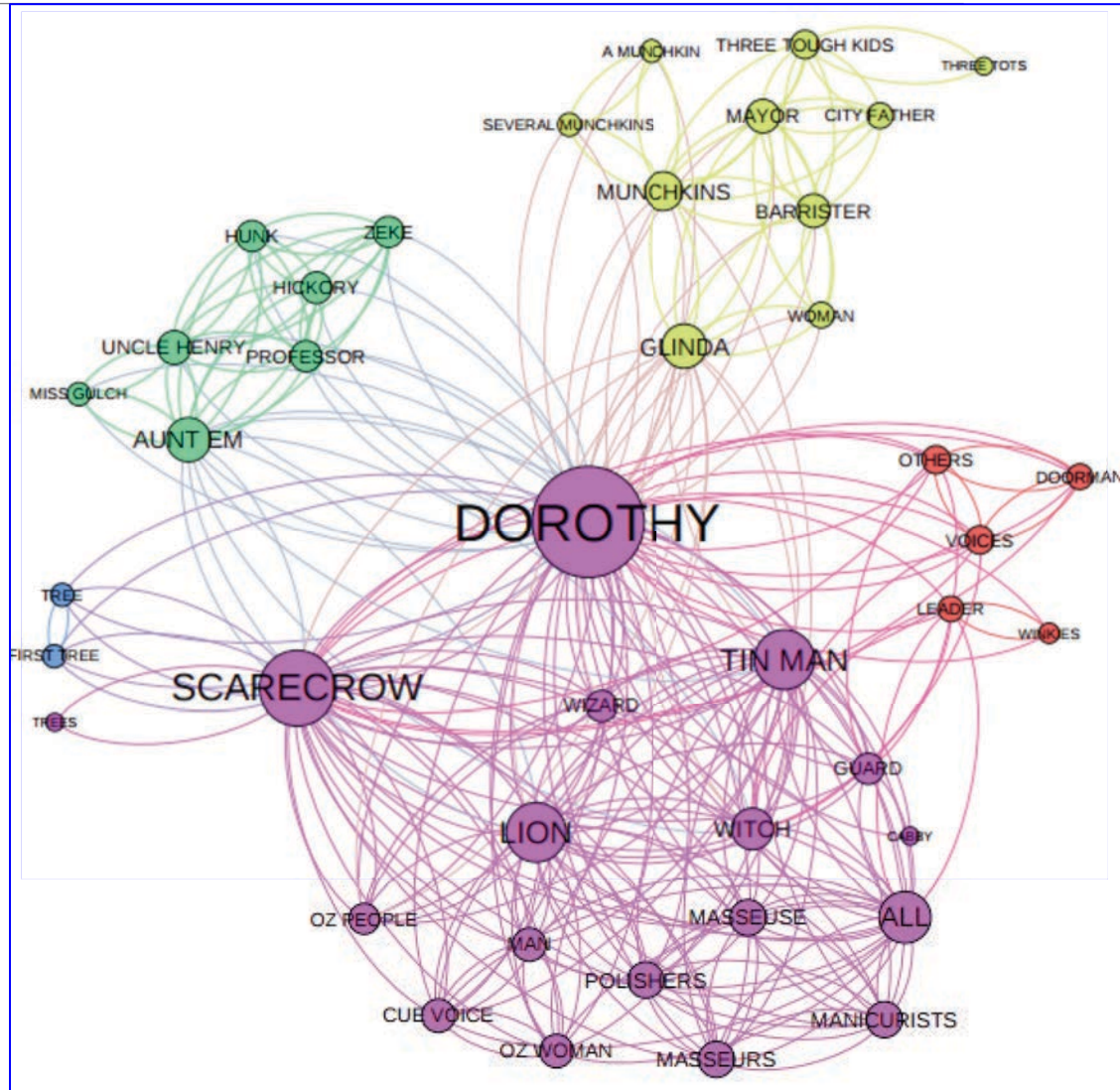
- Understanding a social network
 - Diffusion of misinformation
 - Homophily (clusters of people with similar characteristics)
 - Bridgers (people who connect different groups)
 - Etc.



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

Analyzing Texts

- *Wizard of Oz* (screen play):
 - 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

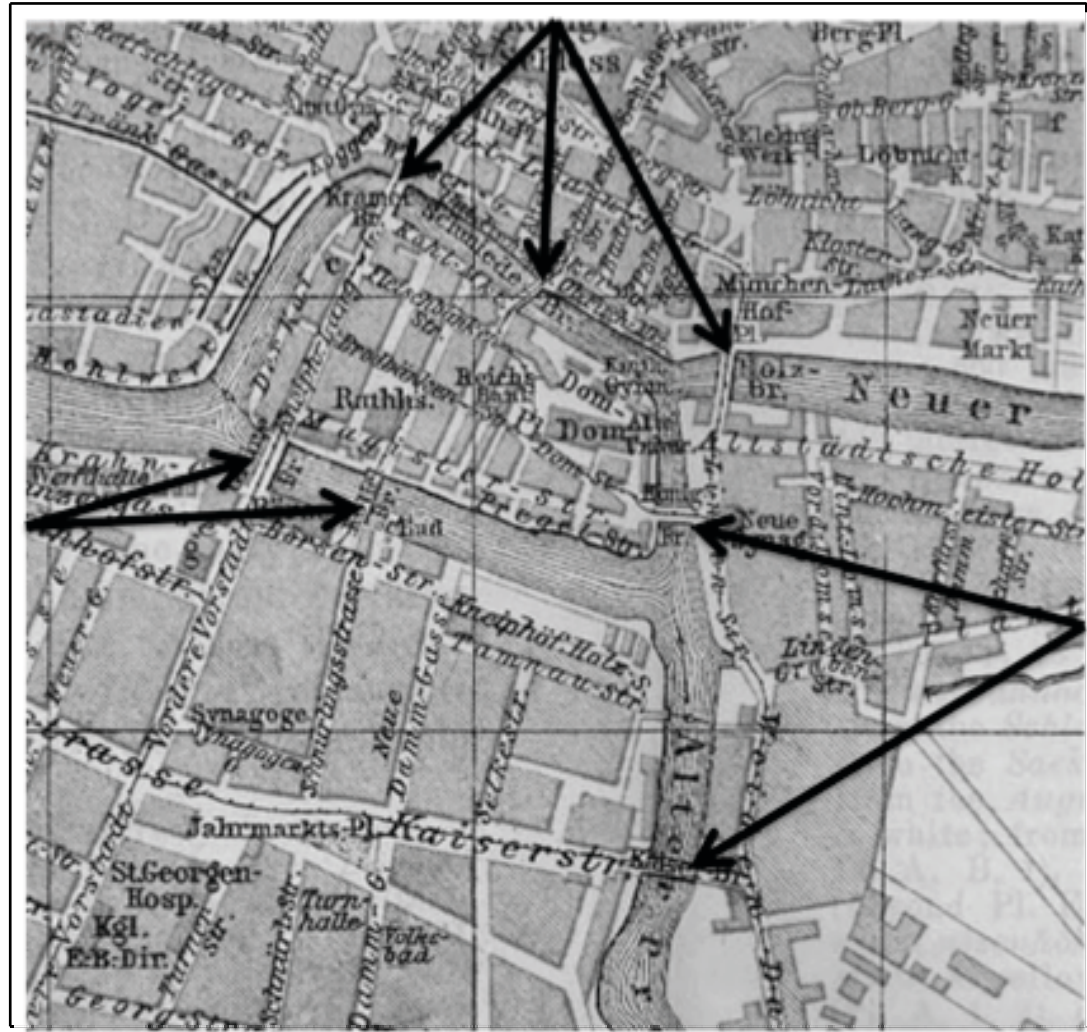
- Model road system using a digraph
 - Nodes: points where roads end or meet
 - Edges: weighted connections between points
 - Expected time between source and destination nodes
 - Distance between source and destination nodes
- Solve a graph optimization problem
 - Shortest weighted path between my house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

*Leads to different
optimization
functions*



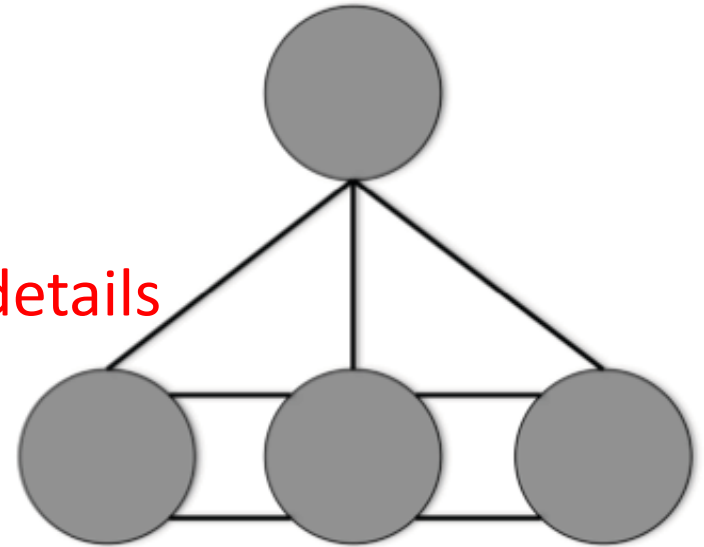
First Reported Use of Graph Theory

- 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 of 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

Classes Node and Edge

```
class Node(str):  
    pass
```

Why?

```
class Edge(object):  
    def __init__(self, src, dest, weight = 1):  
        """Assumes src and dest are nodes"""  
        self._src = src  
        self._dest = dest  
        self._weight = weight  
    def getSource(self):  
        return self._src  
    def getDestination(self):  
        return self._dest  
    def getWeight(self):  
        return self._weight  
    def __str__(self):  
        return self.src + '->(' + self.getWeight() + ')\'  
            + self.dest
```

Common Representations of Digraphs

- Digraph is a directed graph
 - Edges pass in one direction only
- Adjacency matrix
 - Rows: source nodes
 - Columns: destination nodes
 - $\text{Cell}[s, d] = 1$ if there is an edge from s to d
 $= 0$ otherwise
 - Note that in digraph, matrix is **not** symmetric
 - Uses $O(|\text{nodes}|^2)$ memory
- **Adjacency list**
 - Associate with each node a list of destination nodes
 - Use $O(|\text{edges}|)$ memory, therefore good for sparse graphs

Class WeightedDigraph, part 1

```
class WeightedDigraph(object):
    """edges is a dict mapping each node to a list of
    its children and weight of edge"""
    def __init__(self, nodes):
        self._edges = {v: [] for v in nodes}
    def addNode(self, node):
        if node in self._edges:
            raise ValueError('Duplicate node')
        else:
            self._edges[node] = []
    def addEdge(self, edge):
        """edge is an 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, edge.getWeight()))
```

Nodes are used as keys in dictionary

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

A directed weighted edge

Class Weighted Digraph, part 2

```
def childrenOf(self, node):
    return [e[0] for e in self._edges[node]]
def hasNode(self, node):
    return node in self._edges
def getAllNodes(self):
    return(list(self._edges.keys()))
def __str__(self):
    vals = []
    for src in self._edges:
        entry = src + ':'
        for edge in self._edges[src]:
            entry += edge[0] + '(' + str(edge[1]) + '), '
        if entry[-2:] != ':': #there was at least one edge
            vals.append(entry[:-2])
        else:
            vals.append(entry[:-1])
    vals.sort(key = lambda x: x.split(':')[0])
    result = ''
    for v in vals:
        result += v + '\n'
    return result[:-1] #omit final newline
```

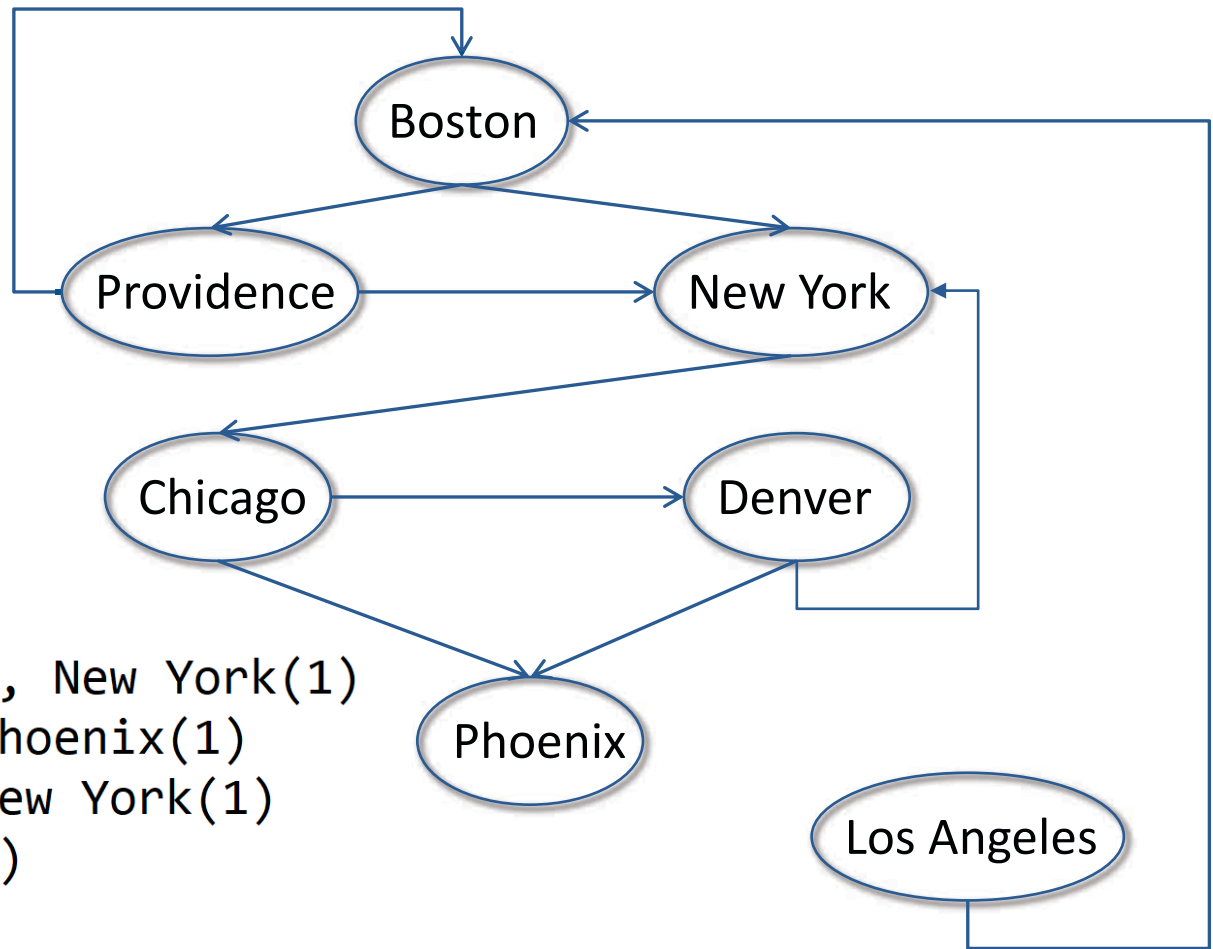
What is going on here?

Built and Print a Graph

```
def buildCityGraph():
    """Generate and return an example graph"""
    g = WeightedDigraph(('Boston', 'Providence', 'New York', 'Chicago',
                        'Denver', 'Phoenix', 'Los Angeles'))
    g.addEdge(Edge('Boston', 'Providence'))
    g.addEdge(Edge('Boston', 'New York'))
    g.addEdge(Edge('Providence', 'Boston'))
    g.addEdge(Edge('Providence', 'New York'))
    g.addEdge(Edge('New York', 'Chicago'))
    g.addEdge(Edge('Chicago', 'Denver'))
    g.addEdge(Edge('Chicago', 'Phoenix'))
    g.addEdge(Edge('Denver', 'Phoenix'))
    g.addEdge(Edge('Denver', 'New York'))
    g.addEdge(Edge('Los Angeles', 'Boston'))
    return g

g = buildCityGraph()
print('The city graph:')
print(g)
```

The Graph



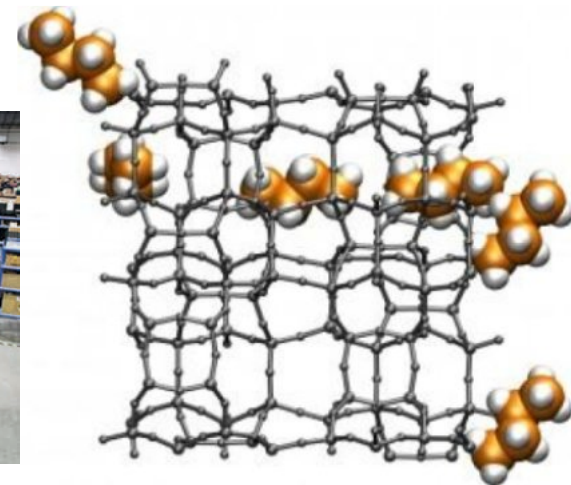
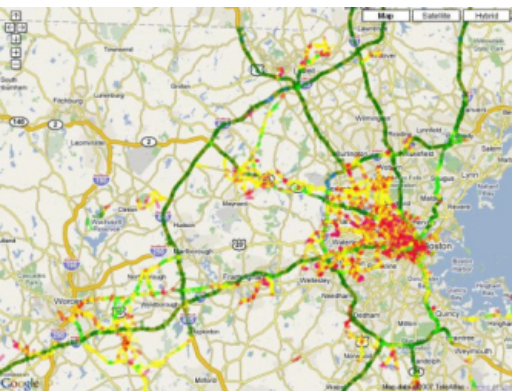
Boston: Providence(1), New York(1)
Chicago: Denver(1), Phoenix(1)
Denver: Phoenix(1), New York(1)
Los Angeles: Boston(1)
New York: Chicago(1)
Phoenix:
Providence: Boston(1), New York(1)

A Classic Graph Optimization Problem

- Shortest (unweighted) path from n_1 to n_2
 - Shortest sequence of edges such that
 - Source node of first edge is n_1
 - Destination of last edge is n_2
 - For edges, e_1 and e_2 , in the sequence, if e_2 follows e_1 in the sequence, the source of e_2 is the destination of e_1
- 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
- ...




Finding the Shortest Path

- Algorithm 1, **breadth-first search** (BFS)
- Algorithm 2, **depth-first search** (DFS)
- Algorithm 3, **Dijkstra's algorithm**

All use *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 (but not necessarily the shortest) from source to destination

Breadth First Search

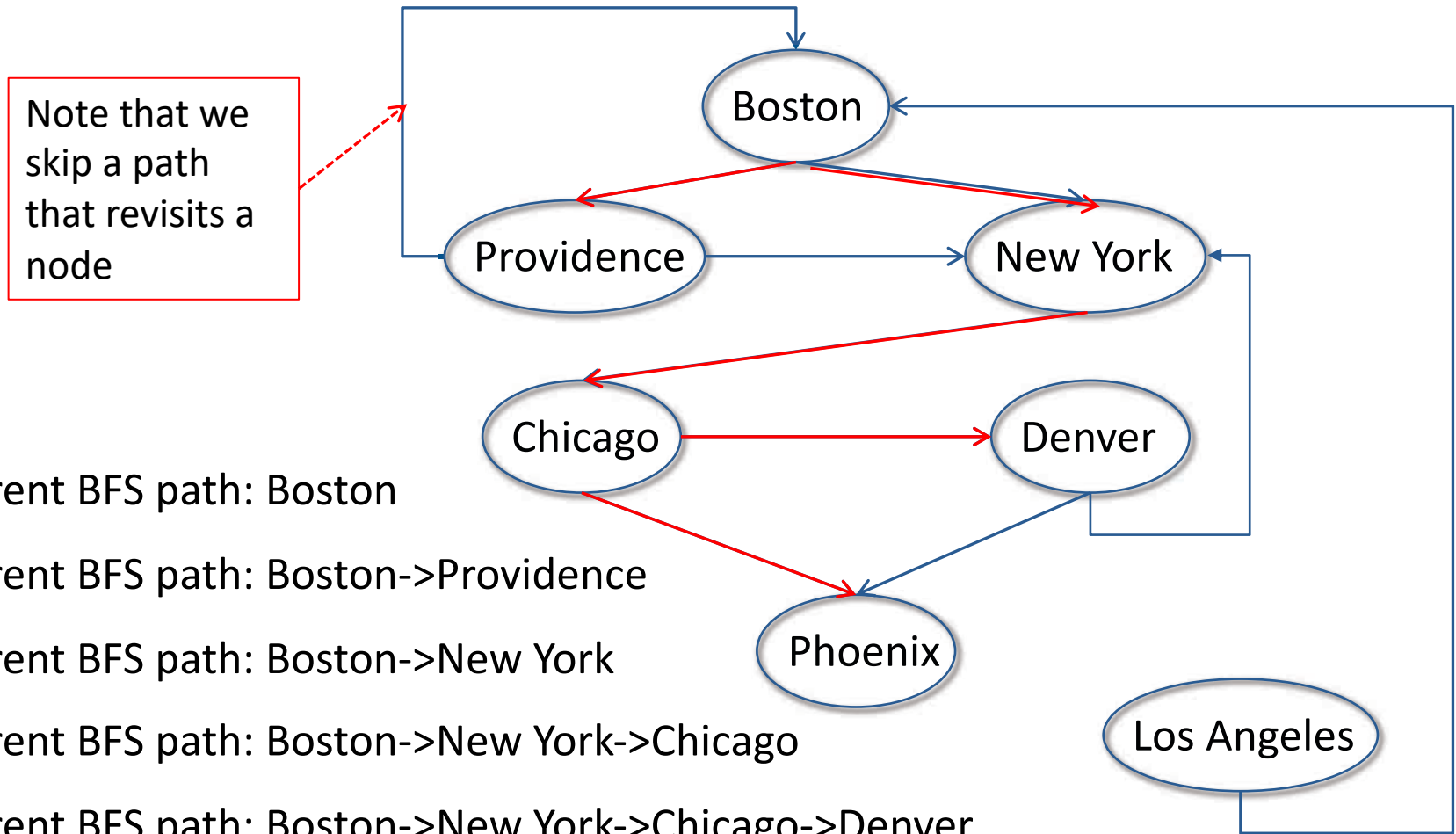
- 
- Start at an initial node
 - Consider all the edges that leave that node, in some order
 - Follow the first edge, and check to see if at goal node
 - If so, stop
 - If not, try the next edge from the current node **that has not yet been examined**
 - Continue until out of options
 - When run out 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

```
def bfs(graph, start, end, toPrint = False):
    """Assumes graph is a Digraph; start and end are nodes
    Returns a shortest path (number of hops) from start to end
    in graph"""
    visited = {} #no vertices visited thus far
    pathQueue = [[start]] #FIFO ← First-in-first-out
    visited[start] = True
    while len(pathQueue) != 0:
        tmpPath = pathQueue.pop(0)
        if toPrint:
            print('Current BFS path:', printPath(tmpPath))
        lastNode = tmpPath[-1]
        if lastNode == end:
            return tmpPath ← Why is it ok to stop?
        for nextNode in graph.childrenOf(lastNode):
            if nextNode not in visited: #avoid visiting again
                newPath = tmpPath + [nextNode] #Note copy of list
                visited[nextNode] = True
                pathQueue.append(newPath)
    return None
```

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

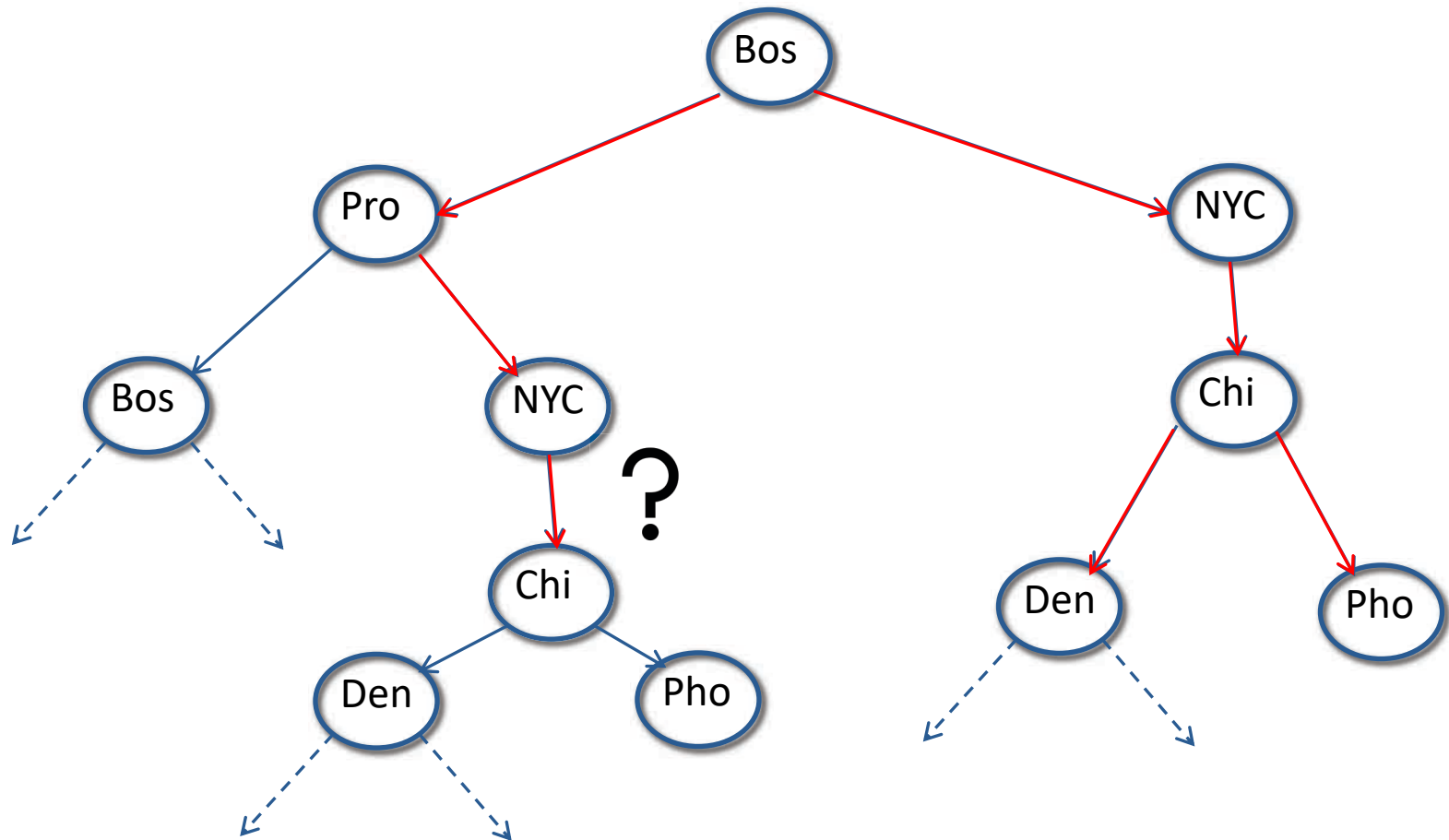
Current BFS path: Boston->New York

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

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

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

Visualizing as a tree search



Depth First Search

*Like brute force solution
to knapsack problem*

- 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 so, check if shorter than shortest already found
- 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



Implementation Like BFS, Except

- Uses a **LIFO** (often called a **stack**) instead of a **FIFO** queue
- Finds multiple paths, not just one



FIFO

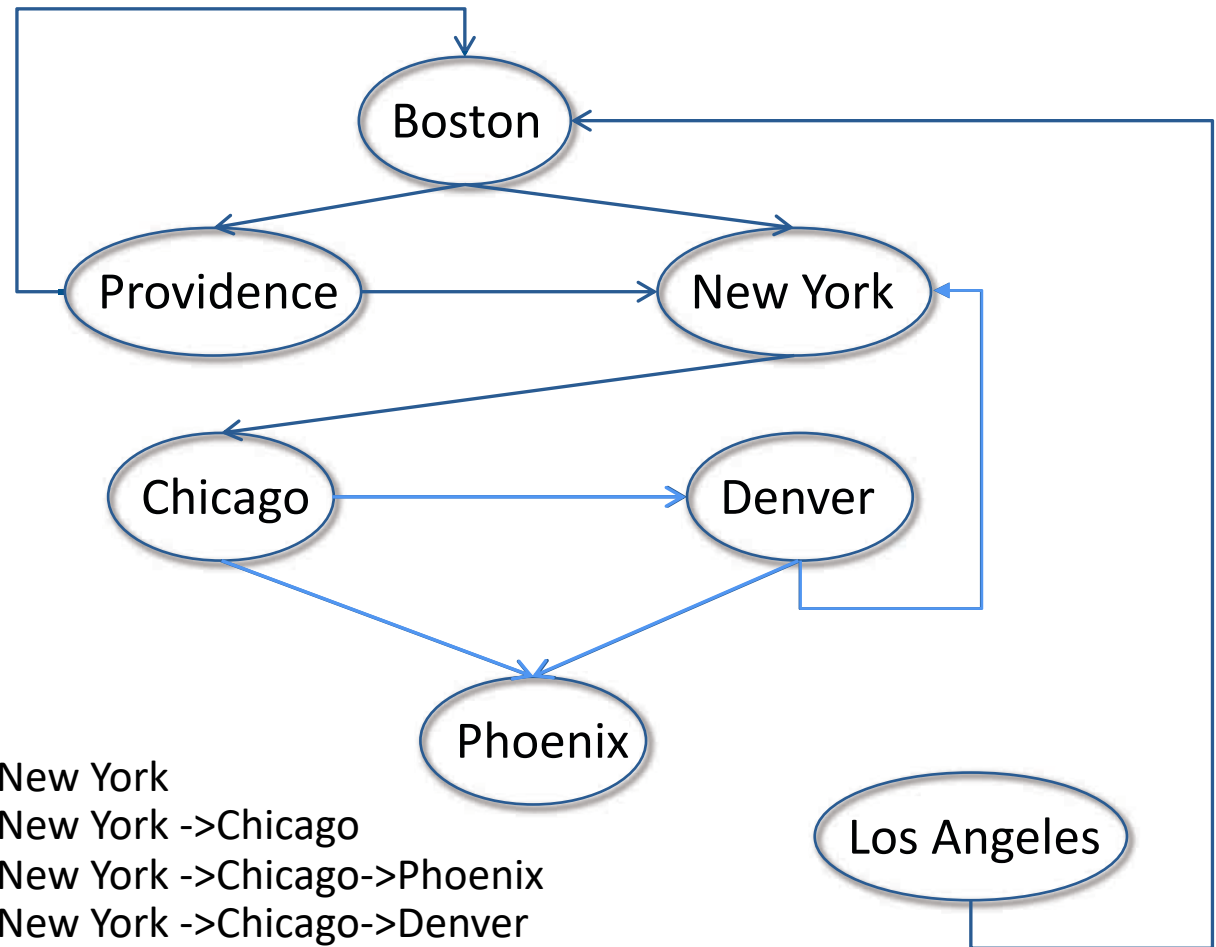


LIFO

Depth-first Search

```
def dfs(graph, start, end, toPrint = False):
    """Assumes graph is a Digraph; start and end are nodes
    Returns a shortest path from start to end in graph"""
    bestPath = None
    initPath = [start]
    pathQueue = [initPath] #LIFO
    while len(pathQueue) != 0:
        #Get and remove newest element in pathQueue
        tmpPath = pathQueue.pop(-1)
        if toPrint:
            print('Current DFS path:', printPath(tmpPath))
        lastNode = tmpPath[-1]
        if lastNode == end:
            if toPrint:
                print('Path found')
            if bestPath == None or len(tmpPath) < len(bestPath):
                bestPath = tmpPath
            continue
        if bestPath != None and len(tmpPath) >= len(bestPath):
            continue
        for nextNode in graph.childrenOf(lastNode):
            if nextNode not in tmpPath:
                newPath = tmpPath + [nextNode]
                pathQueue.append(newPath)
    return bestPath
```

Output (Boston to Phoenix)



Current DFS path: Boston

Current DFS path: Boston->New York

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

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

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

Current DFS path: Boston->Providence

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

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

BFS and DFS

- BFS fast for unweighted graphs
 - But early stopping does not work for weighted graphs
- DFS easily modified to deal with weighted graphs
 - But slow
- Gets us to Dijkstra's algorithm

But First, a Five Minute Break

Dijkstra

Dahl & Nygaard (Classes)

Hoare (Quick Sort)



Dijkstra's Algorithm

- Generalization of breadth-first search that does not require edges to have equal weights
- Uses a priority queue instead of a FIFO

Three Key Data Structures and Basic Idea

- **unvisited**: list of nodes that have not yet been visited.
 - Initially contains all nodes in graph
- **distanceTo**: a dict mapping each node to the minimum distance found so far of that node from the start node
 - Initially zero for start node and infinity for all others
- **predecessor**: a dict mapping each node to a predecessor node on the shortest path found so far from start to that node
 - Initially None for all nodes
- Visit nodes in increasing order of distance from start (as in BFS), updating *distanceTo* and *predecessor*
- When all nodes visited, construct shortest path from *predecessor* by working backwards from end node

Outline of Algorithm

- For current node (initially start node), chose a node with shortest distance from current node to visit first
 - This is the metric defining priority queue
- Check each of its neighbors
 - Calculate distance from starting node to neighbor
- Will show implementation that assumes all edges have equal weights
 - Almost trivial to adapt to unequal weights
 - But need to leave something for you to think about

Initialization

```
def Dijkstra(graph, start, end, toPrint = False):  
    """  
    graph: an unweighted digraph  
    start: a node in graph  
    end: a node in graph  
    returns a list representing shortest path from start to end,  
           and None if no path exists"""  
    #Easily modified to deal with non-negative weighted edges  
  
    # Mark all nodes unvisited and store them.  
    # Set the distance to zero for our initial node  
    # and to infinity for other nodes.  
    unvisited = graph.getAllNodes()  
    distanceTo = {node: float('inf') for node in graph.getAllNodes()}  
    distanceTo[start] = 0  
    # Mark all nodes as not having found a predecessor node on path  
    #from start  
    predecessor = {node: None for node in graph.getAllNodes()}
```

Main Loop

`while` unvisited: *While unvisited not empty*

*# Select the unvisited node with the smallest distance from
start, it's current node now.*

`current = min(unvisited, key=lambda node: distanceTo[node])`

`if` toPrint: *#for pedagogical purposes*

`...`

Stop, if the smallest distance

among the unvisited nodes is infinity.

`if` distanceTo[current] == `float('inf')`:

`break`

Find unvisited neighbors for the current node

and calculate their distances from start through the

current node.

`for` neighbour `in` graph.childrenOf(current):

`alternativePathDist = distanceTo[current] + 1`

Compare the newly calculated distance to the assigned.

Save the smaller distance and update predecessor.

`if` alternativePathDist < distanceTo[neighbour]:

`distanceTo[neighbour] = alternativePathDist`

`predecessor[neighbour] = current`

Remove the current node from the unvisited set.

`unvisited.remove(current)`

Ordering used by min

Counting hops

Build Path from predecessors

#Attempt to be build a path working backwards from end

```
path = []
```

```
current = end
```

```
while predecessor[current] != None:
```

```
    path.insert(0, current)
```

```
    current = predecessor[current]
```

```
if path != []:
```

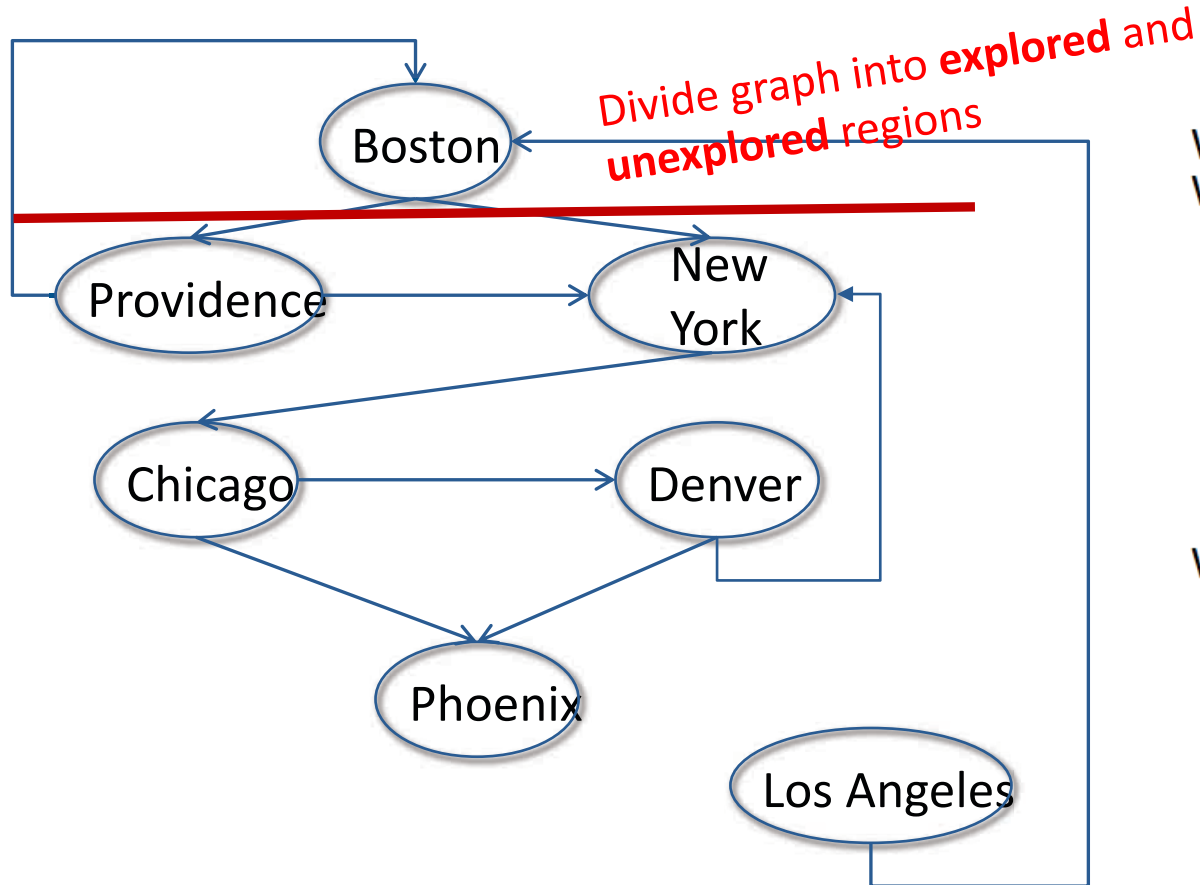
```
    path.insert(0, current)
```

```
else:
```

```
    return None
```

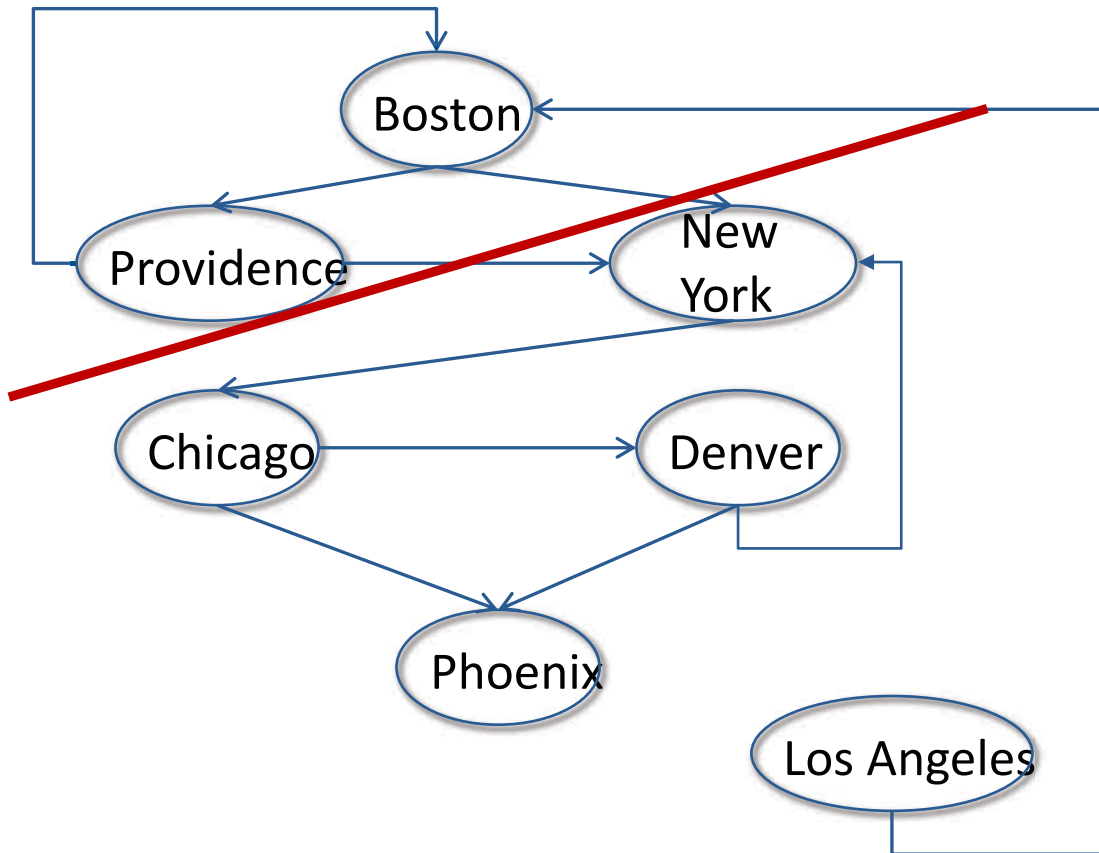
```
return path
```

Example



Value of current: Boston
Value of distanceTo:
Boston: 0
Providence: inf
New York: inf
Chicago: inf
Denver: inf
Phoenix: inf
Los Angeles: inf
Value of predecessor:
Boston: None
Providence: None
New York: None
Chicago: None
Denver: None
Phoenix: None
Los Angeles: None

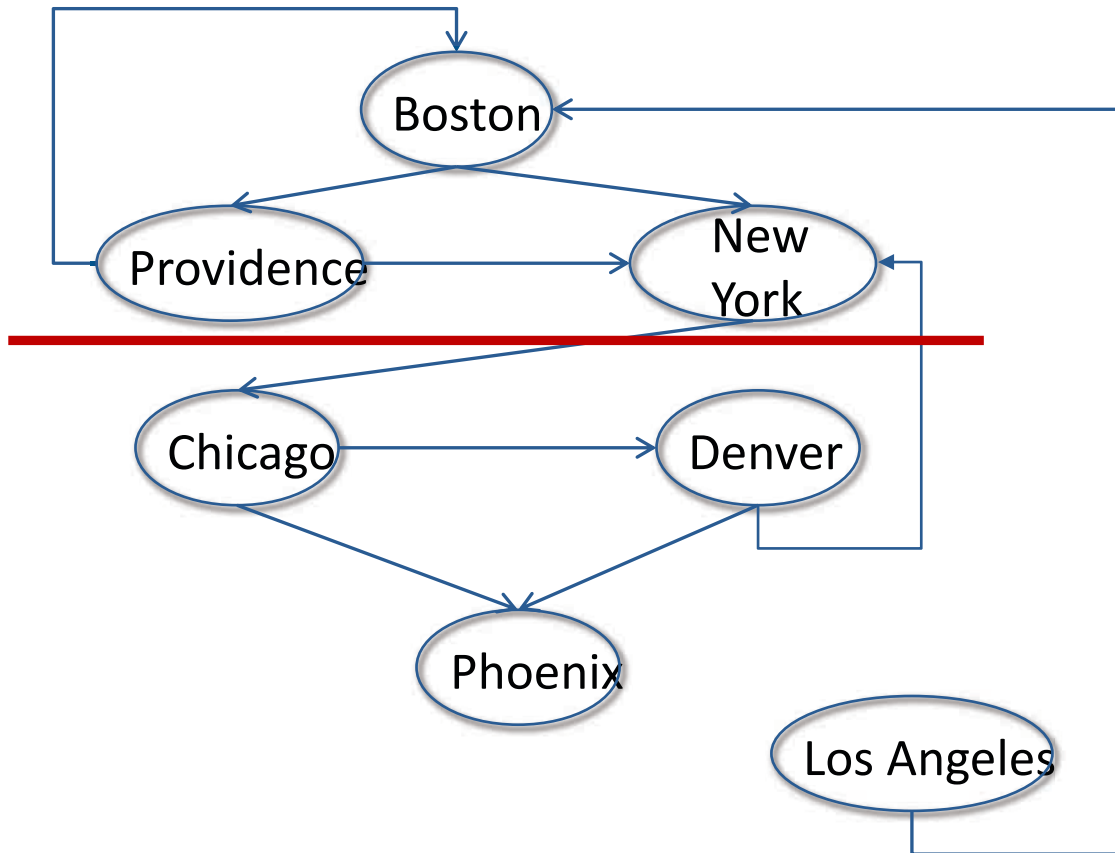
Move Graph Cut



Value of current: Providence
Value of distanceTo:
Boston: 0
Providence: 1
New York: 1
Chicago: inf
Denver: inf
Phoenix: inf
Los Angeles: inf
Value of predecessor:
Boston: None
Providence: Boston
New York: Boston
Chicago: None
Denver: None
Phoenix: None
Los Angeles: None

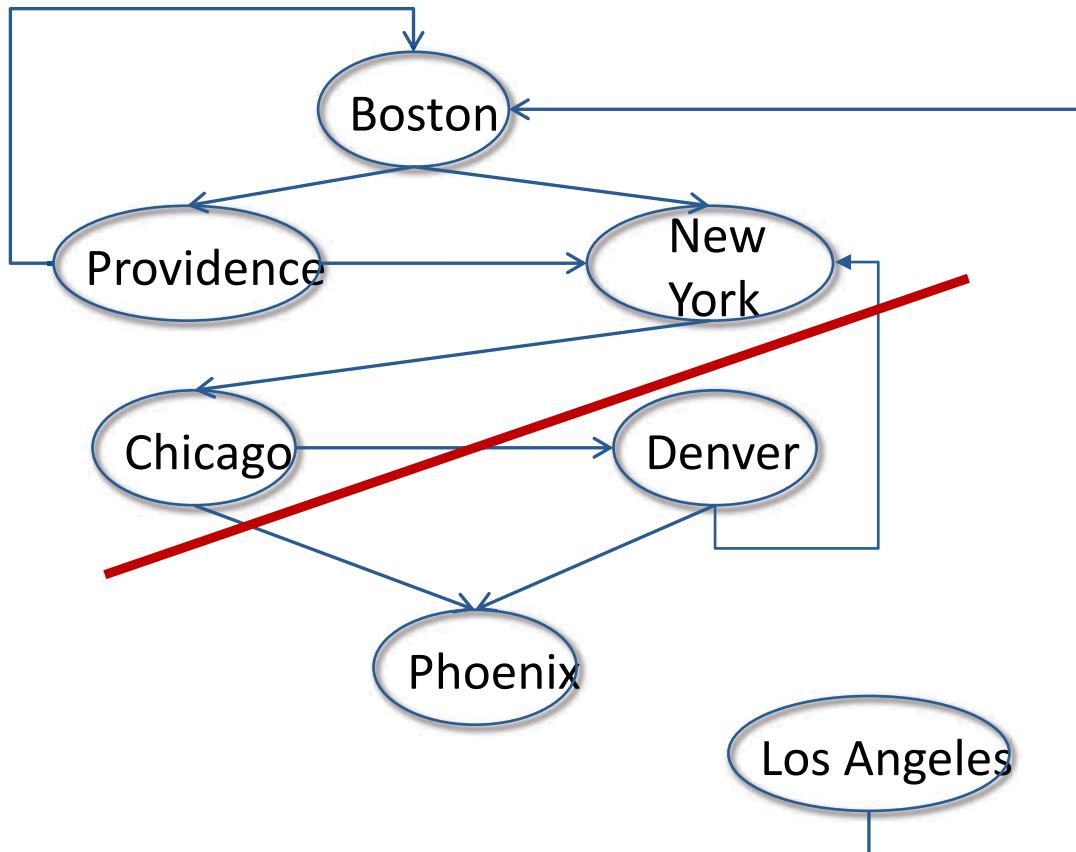
distanceTo and predecessor
guaranteed not to change for
nodes above the cut

Move Graph Cut



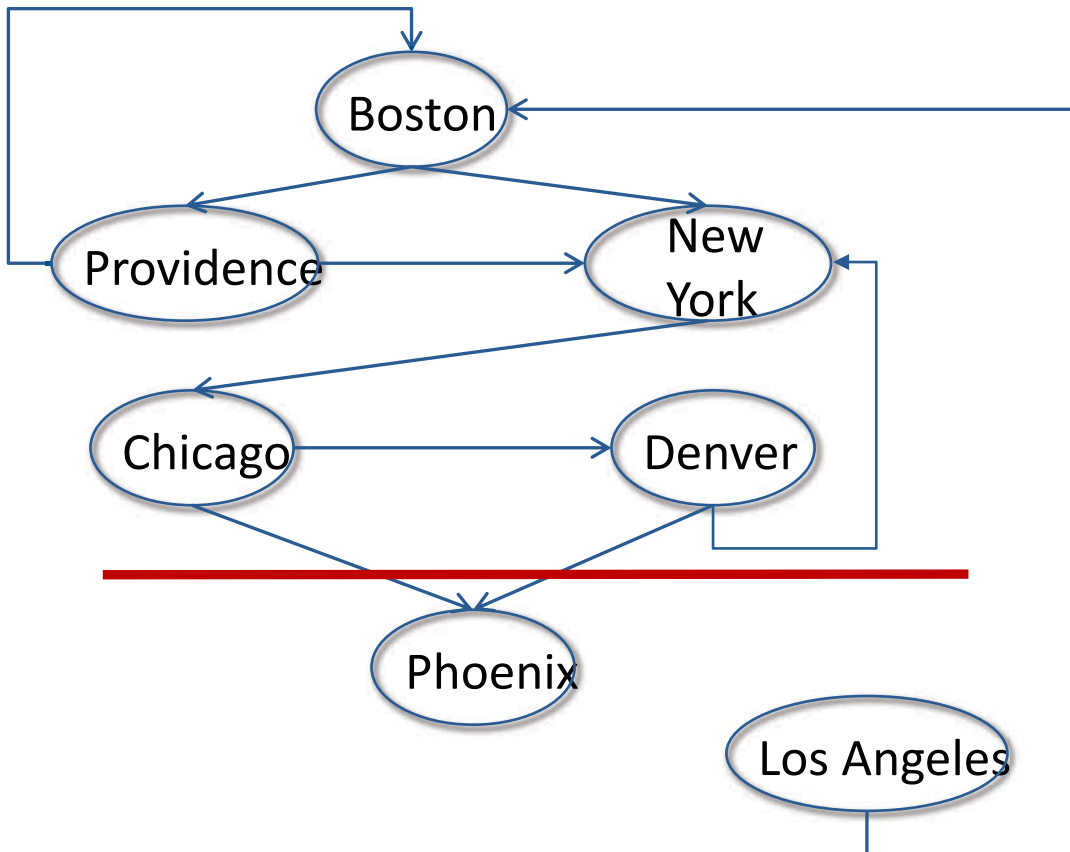
Value of current: New York
Value of distanceTo:
Boston: 0
Providence: 1
New York: 1
Chicago: inf
Denver: inf
Phoenix: inf
Los Angeles: inf
Value of predecessor:
Boston: None
Providence: Boston
New York: Boston
Chicago: None
Denver: None
Phoenix: None
Los Angeles: None

Move Graph Cut



Value of current: Chicago
Value of distanceTo:
Boston: 0
Providence: 1
New York: 1
Chicago: 2
Denver: inf
Phoenix: inf
Los Angeles: inf
Value of predecessor:
Boston: None
Providence: Boston
New York: Boston
Chicago: New York
Denver: None
Phoenix: None
Los Angeles: None

Move Graph Cut



Value of current: Denver

Value of distanceTo:

Boston: 0

Providence: 1

New York: 1

Chicago: 2

Denver: 3

Phoenix: 3

Los Angeles: inf

Value of predecessor:

Boston: None

Providence: Boston

New York: Boston

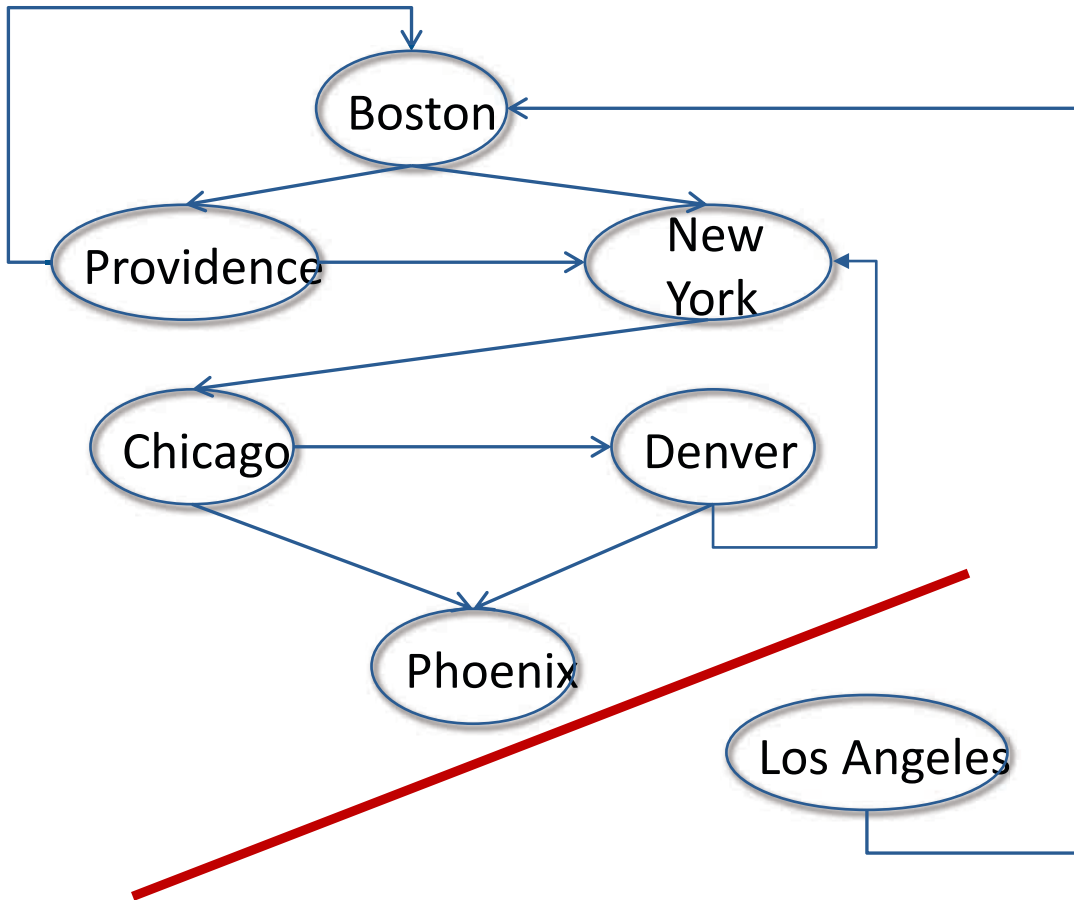
Chicago: New York

Denver: Chicago

Phoenix: Chicago

Los Angeles: None

Move Graph Cut



Value of current: Phoenix

Value of distanceTo:

Boston: 0

Providence: 1

New York: 1

Chicago: 2

Denver: 3

Phoenix: 3

Los Angeles: inf

Value of predecessor:

Boston: None

Providence: Boston

New York: Boston

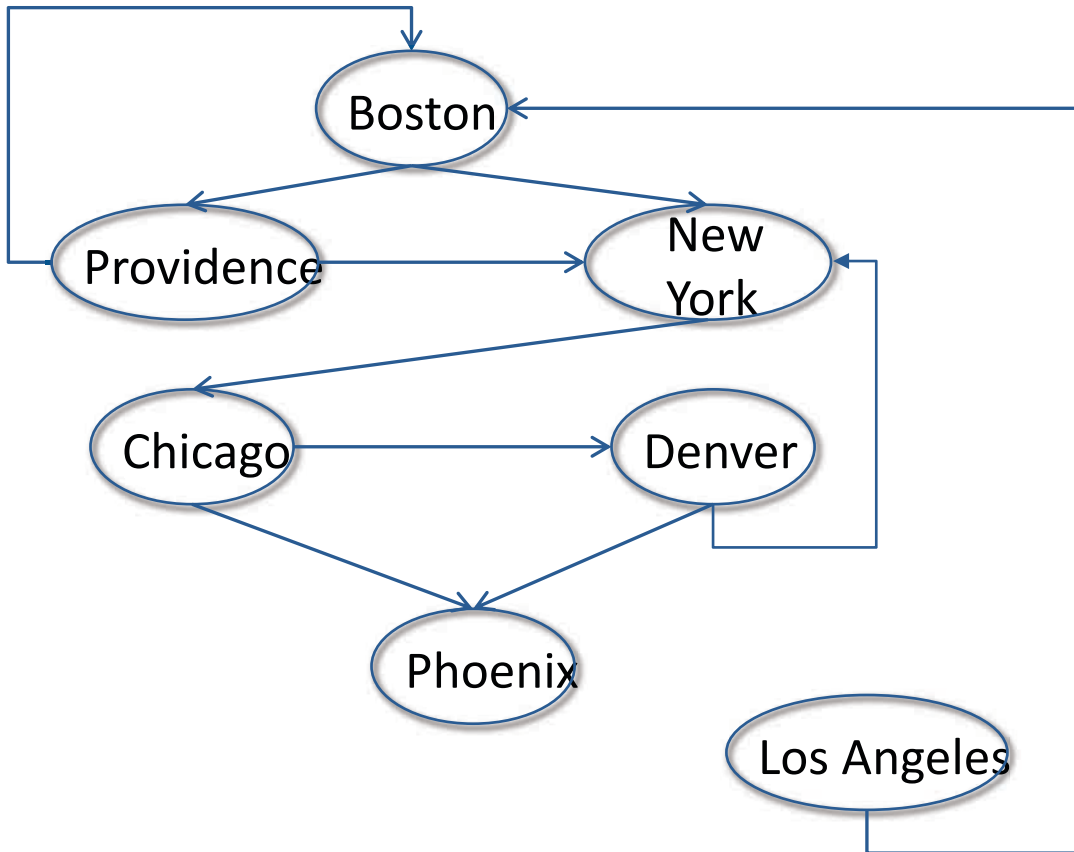
Chicago: New York

Denver: Chicago

Phoenix: Chicago

Los Angeles: None

Move Graph Cut



Value of current: Los Angeles

Value of distanceTo:

Boston: 0

Providence: 1

New York: 1

Chicago: 2

Denver: 3

Phoenix: 3

Los Angeles: inf

Value of predecessor:

Boston: None

Providence: Boston

New York: Boston

Chicago: New York

Denver: Chicago

Phoenix: Chicago

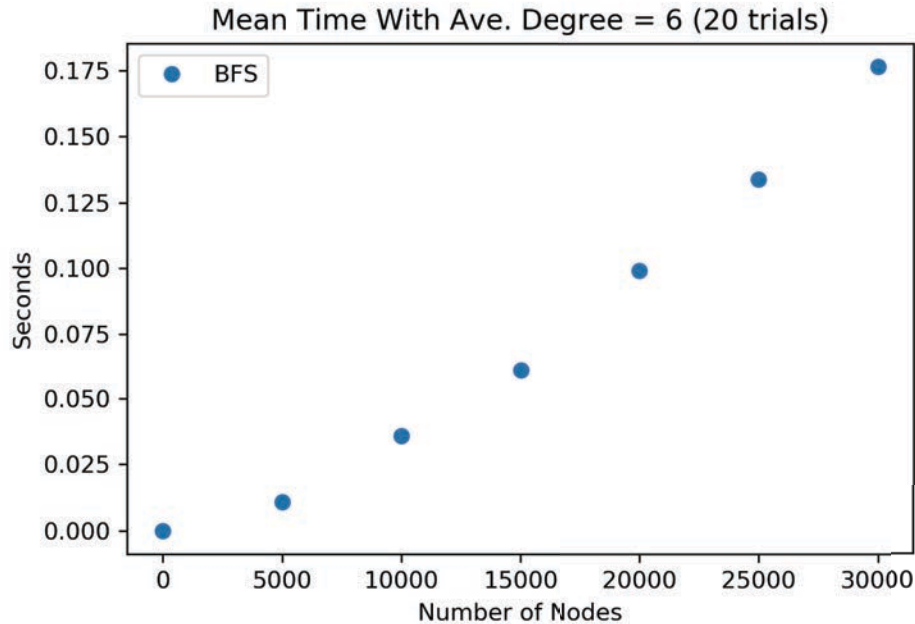
Los Angeles: None

Boston

New York

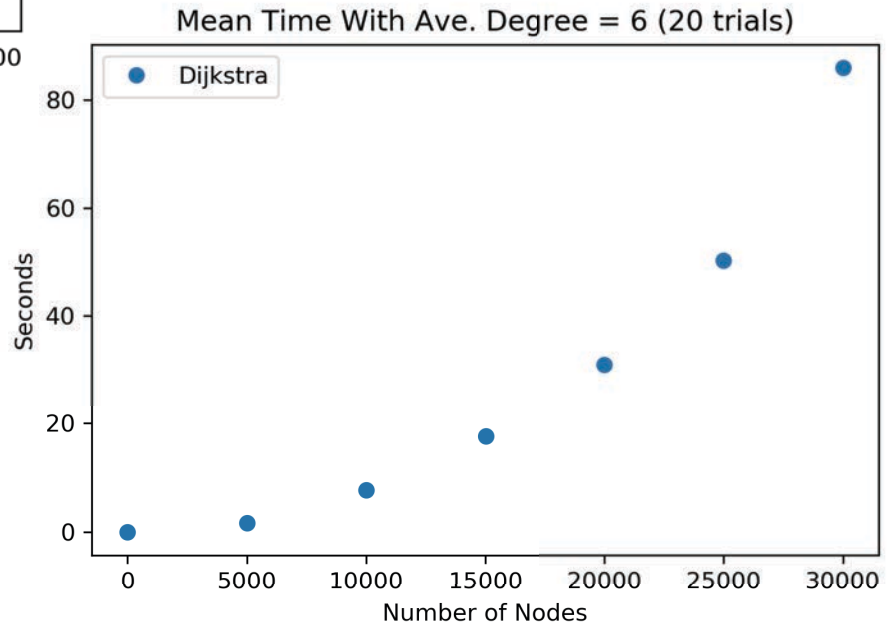
Chicago

Test on Some Large Graphs



$$O(|E| + |V|)$$

$$O(|E| + |V| \log |V|)$$



So, Why Bother with Dijkstra's Algorithm

- Suppose edges have non-negative weights?
 - DFS and BFS have to explore all paths
 - Dijkstra's (suitably modified) does not
- All-nodes shortest path

All Nodes Shortest Path

- Notice that *end* doesn't come into play until last step of algorithm
- *predecessor* can be used to **quickly** find a path from start to **any** node in graph
- If algorithm run using each node as start, and result stored, can quickly find shortest path between any pair of nodes
 - Need not start from scratch for each starting node

Summarizing

- 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
- Dijkstra's algorithm better for finding shortest path in large graphs with (non-negative) weighted edges
 - Many variants optimized for specific applications
 - Especially useful for multiple tasks