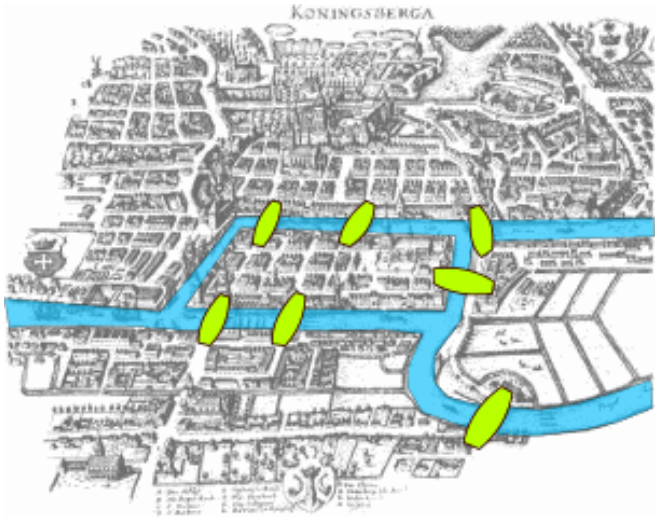


GRAPH-THEORETIC MODELS

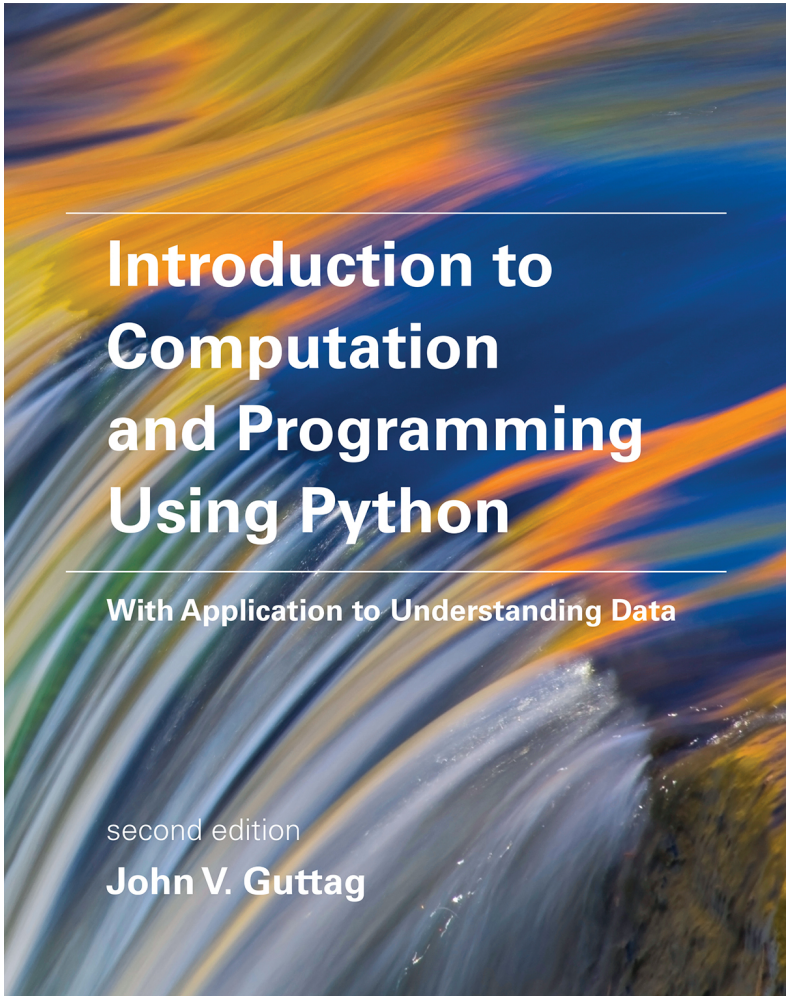
(download slides and .py files to follow along)

Eric Grimson

MIT Department Of Electrical Engineering and Computer Science

Relevant Reading

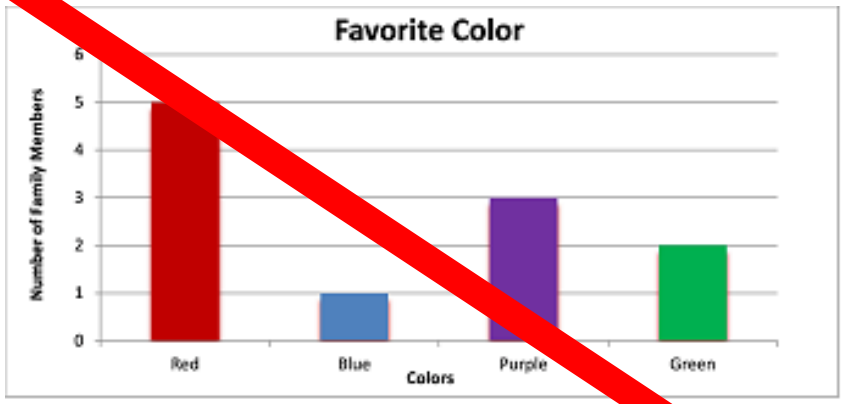
- Today
  - Section 12.2
- Next time
  - Section 15.1-15.4.1
  - Section 15.5



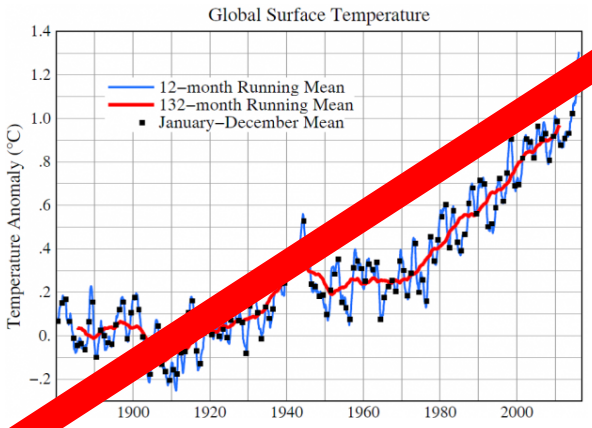
Computational Models

- Programs that help us understand real world settings and solve instances of practical problems
  - Framework on which to build computational thinking
  - Provides a computational *(in silico)* complement to physical *(in vitro* or *in vivo)* experiments and mathematical models
- 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
  - A decision tree can help find a good solution to such problems
  - Can try different metrics to optimize within decision tree structure
- Now want to look at broader class of models – **graphs**
  - Nice way to formulate many problems, mapping from problem description to rigorous computational structure to algorithms
  - Lead to powerful solutions to optimization problems

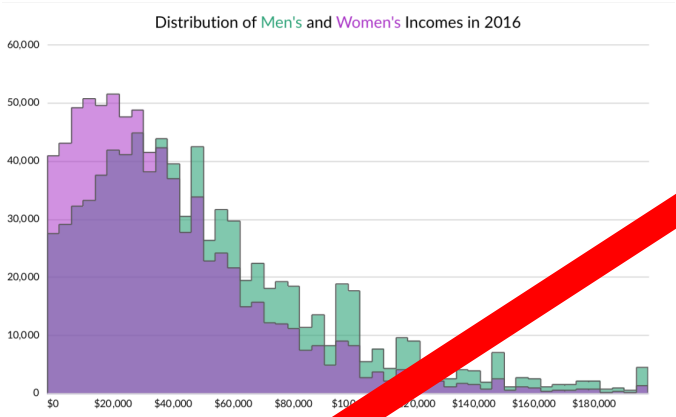
What is a Graph?



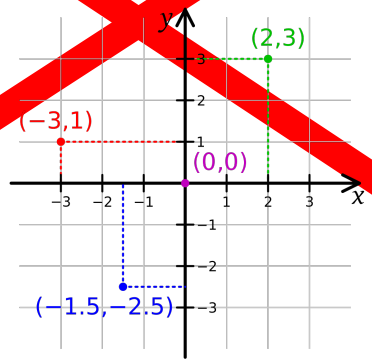
bar



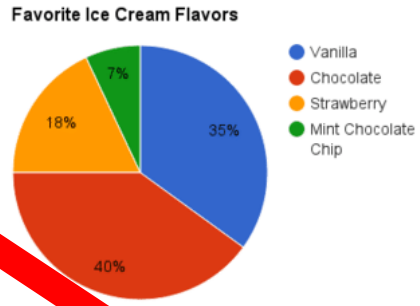
line



histogram



Cartesian

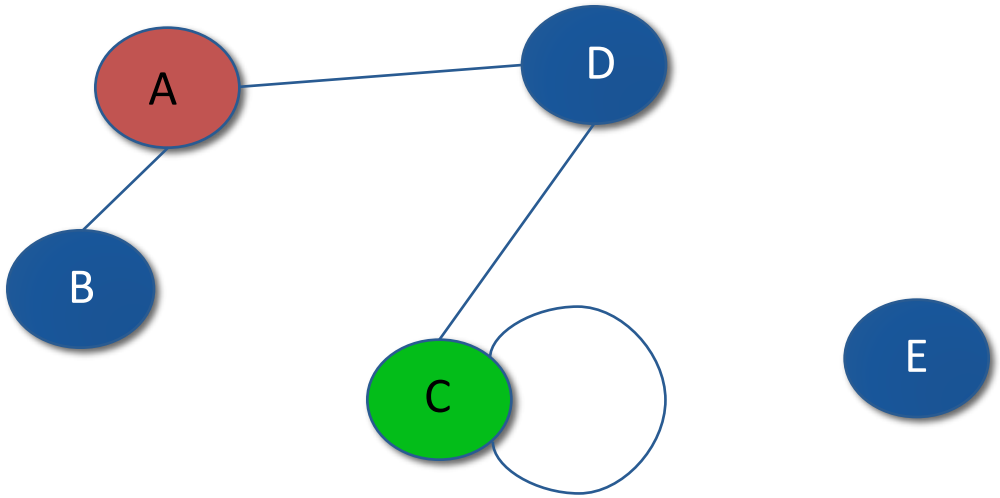


pie

These are all visual presentations of information; we want a structure that supports computation

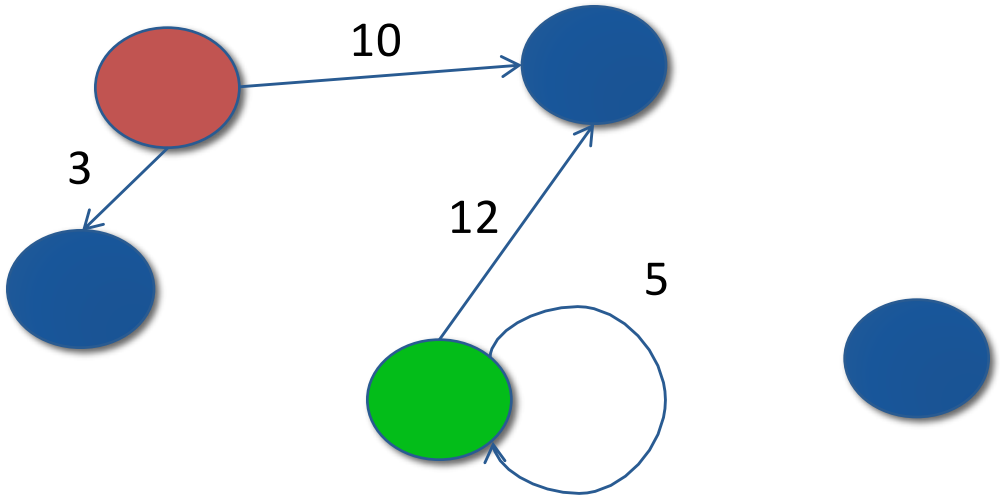
What is a Graph?

- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each connecting a pair of nodes
  - Undirected (graph)



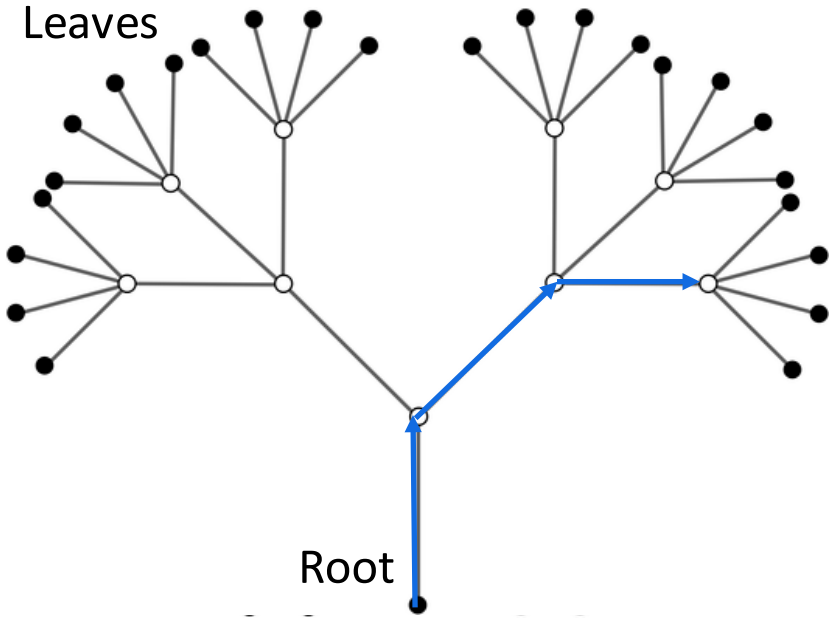
What is a Graph?

- 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

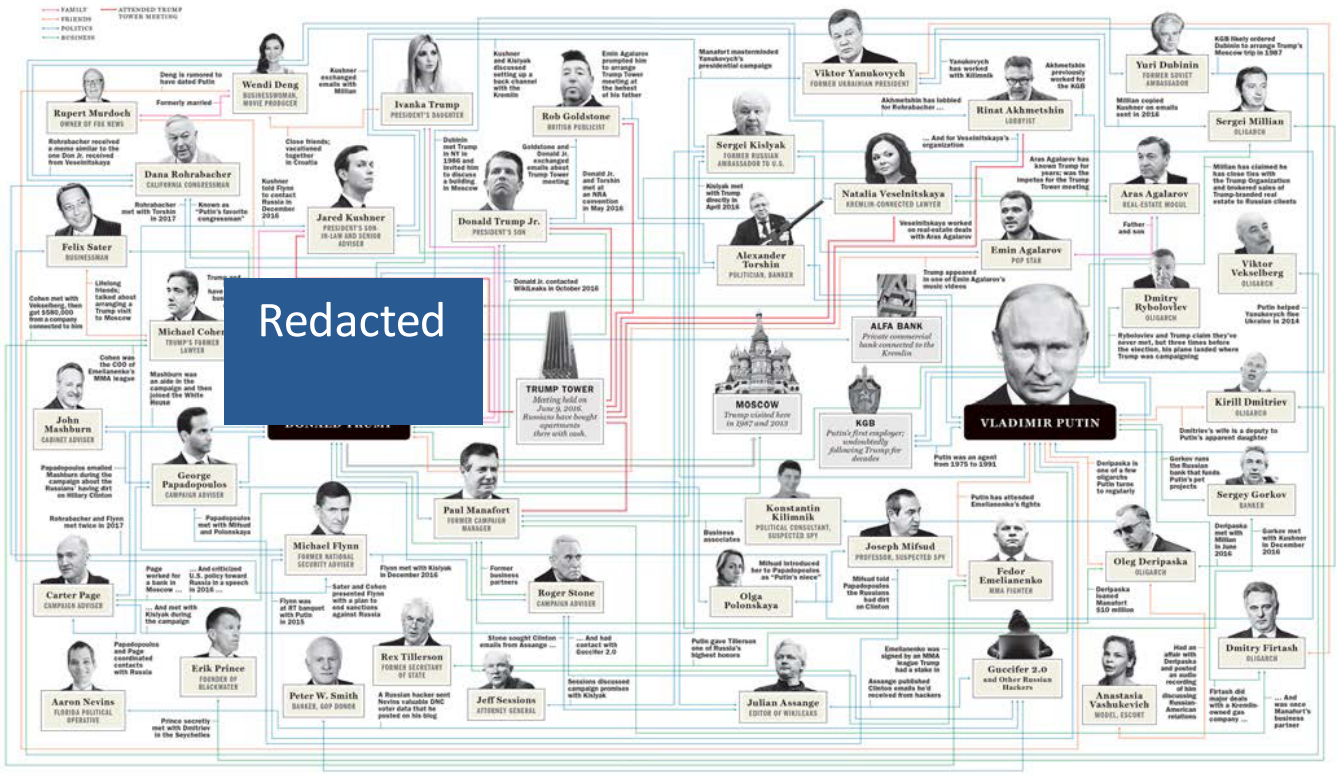
- A special kind of directed graph in which any pair of nodes is connected by a single path from the node closer to the root to the node further from the root
  - Recall the search trees we used to solve knapsack problem



An Australian tree?

Why Graphs?

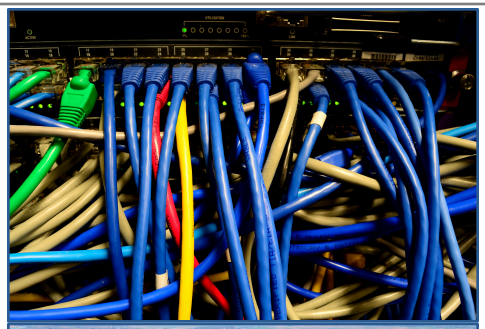
- Capture and reason about relationships among entities
  - Routes between Boston and San Jose
  - How the atoms in a molecule are related to one another
  - Ancestral relationships (family trees)
  - Business/social/political connections
  - …

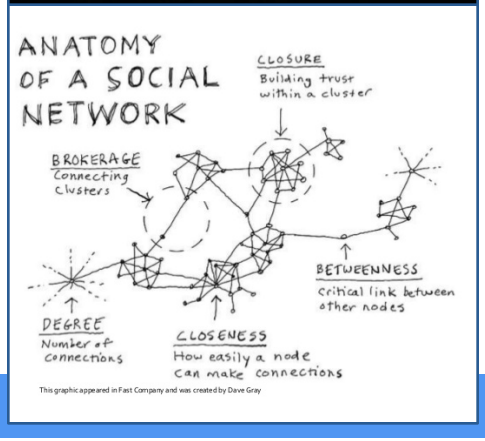


New York Magazine

Graphs model a wide range of systems

- Computer networks
  - How can I efficiently route information from one node to another?
- Transportation networks
  - How can I efficiently get to a particular destination?
- Electrical network
  - How can I efficiently transmit electricity between source and sink?
- Social networks
  - How can I understand diffusion of misinformation, identify clusters of people with similar characteristics



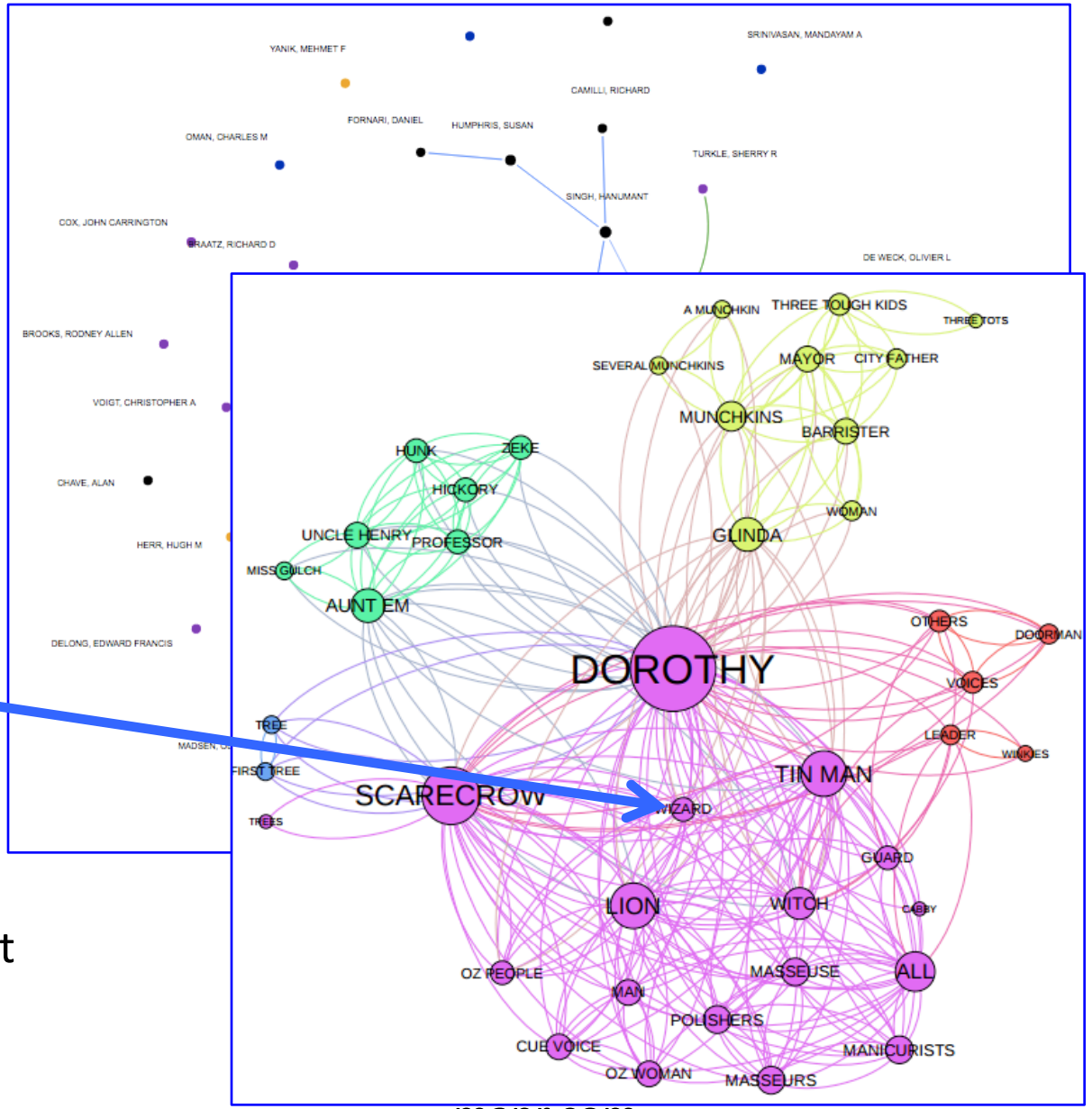


The first three examples all ask about finding an efficient path between two nodes; the last example suggests that there can be other questions

Graphs can answer more than path problems

- Finding connections between faculty members' research
  - Size of node is number of articles
  - Edges are co-authored papers

- Analyzing text information, e.g., "Wizard of Oz":
  - size of node reflects number of scenes in which character shares dialogue
  - edges represent interactions
  - color of clusters reflects natural interactions with each other but not others



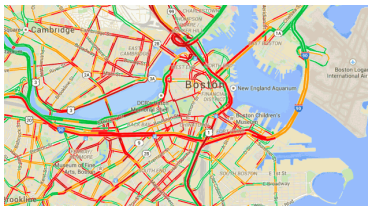
mapr.com

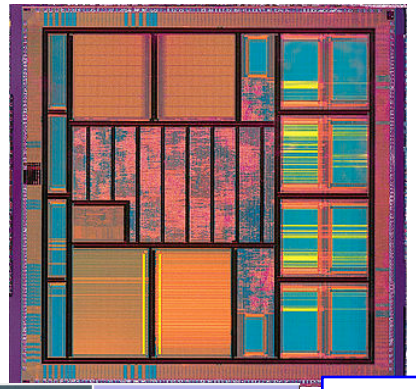
Why Graphs Are So Useful

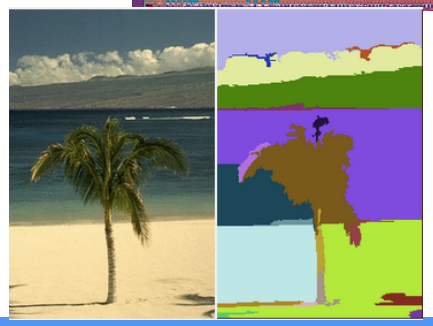
- Not only do graphs capture relationships in connected networks of items, they support **inference** on those structures

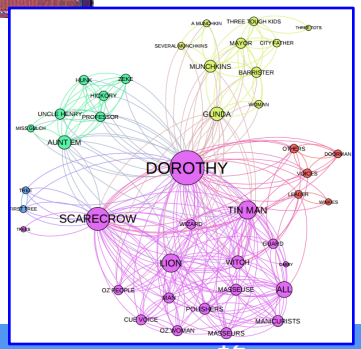
- Find sequences of links between elements (aka the **path problem**)
- Finding least expensive path between elements (aka the **shortest path problem**)
- Partitioning graph into subgraphs with minimal connections between them (aka **graph partition problem** or **graph clique problem**)
- Finding the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

You'll see these problems in 6.042, 6.046, and other classes

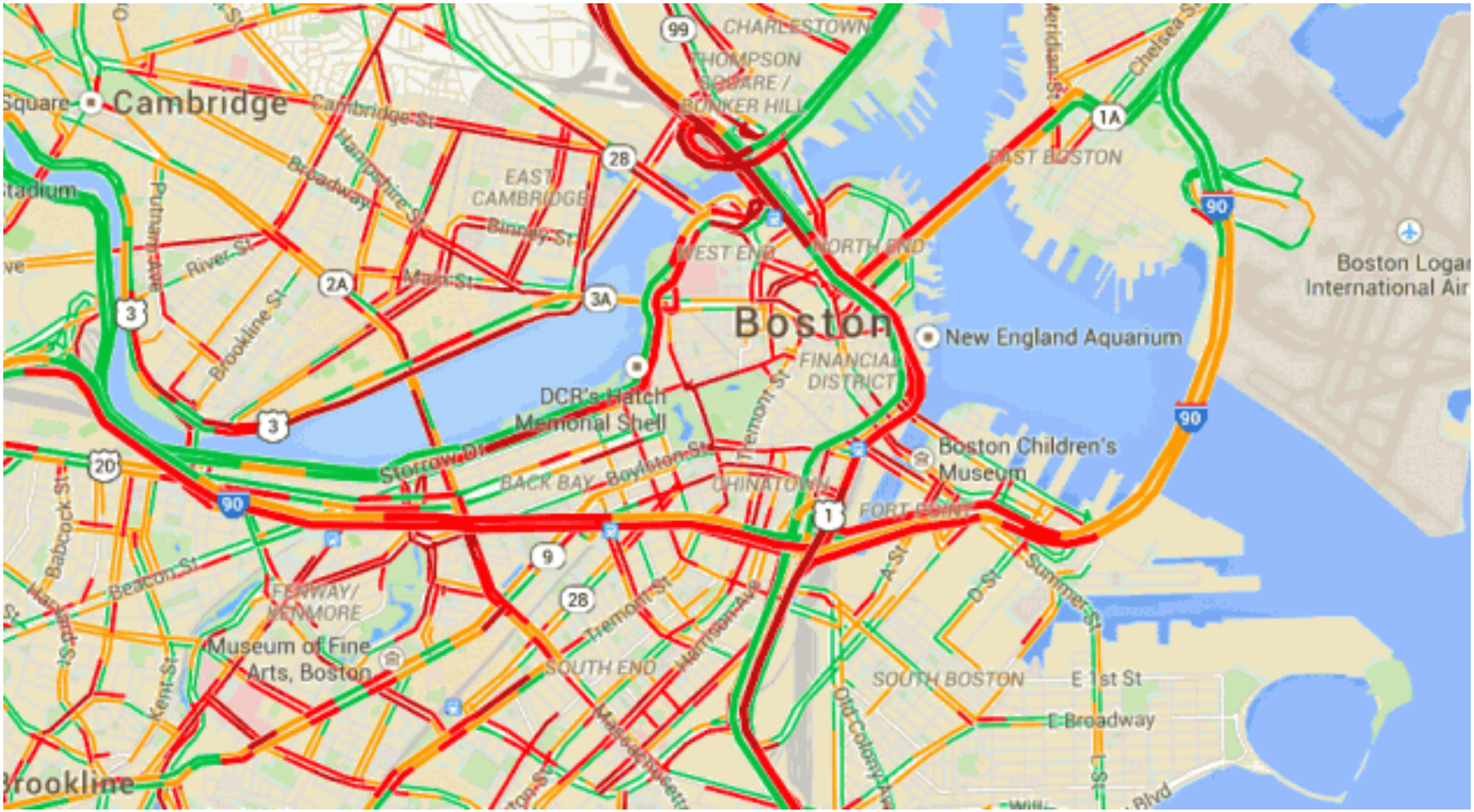








Graph Theory Saves Me Time Every Day



Some path problems are easier than others

Getting Chancellor Grimson to his Office

- Model road system using a digraph
  - Nodes: points where roads end or meet (intersections)
  - Edges: weighted connections between points (streets)
    - Expected time between nodes for each edge
    - Distance between nodes
    - Maximum minimum speed between nodes

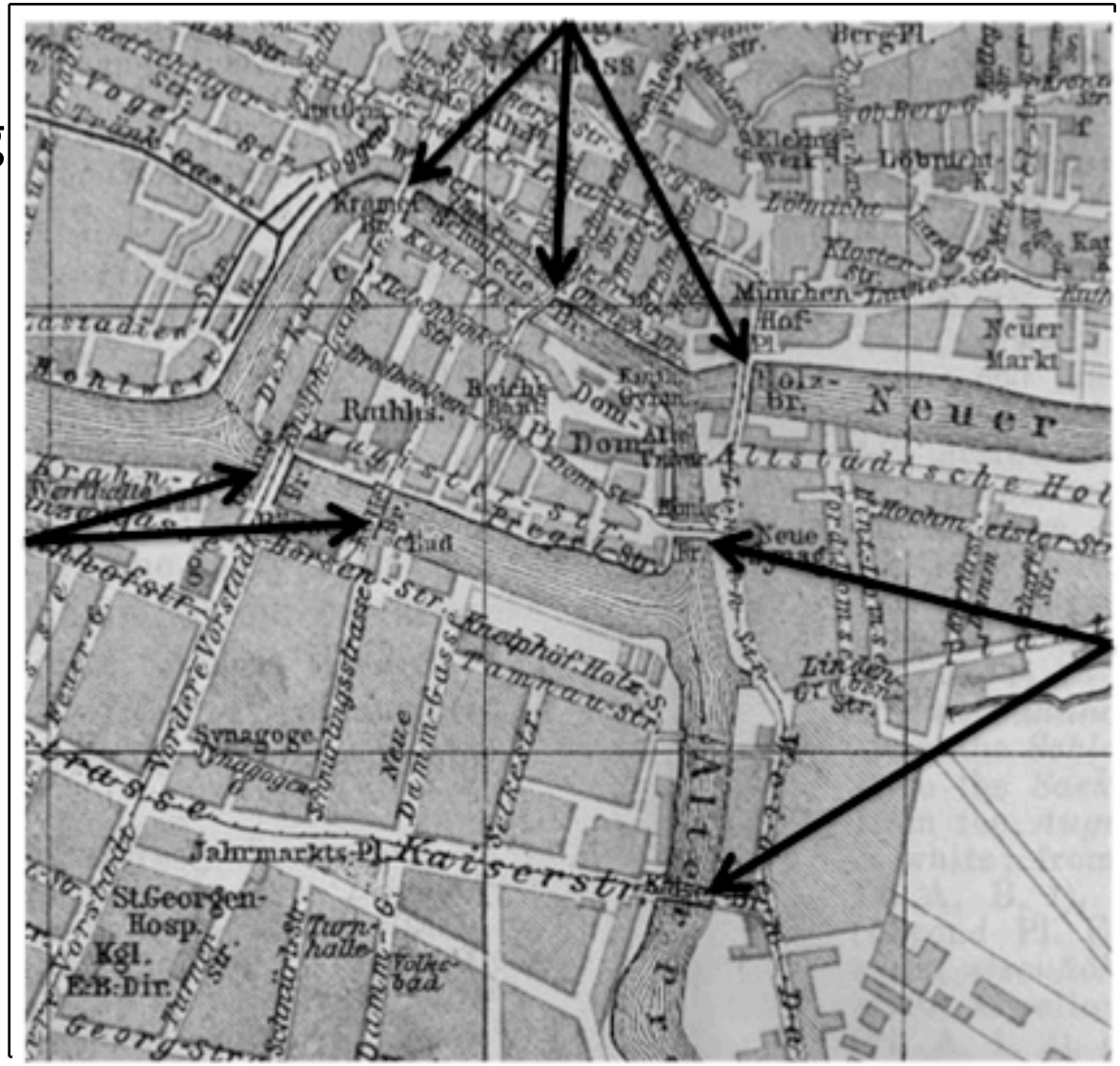
Leads to different optimization functions

Poll:
Which optimization do you prefer

- Solve a graph optimization problem
  - Shortest weighted path between my house and my office

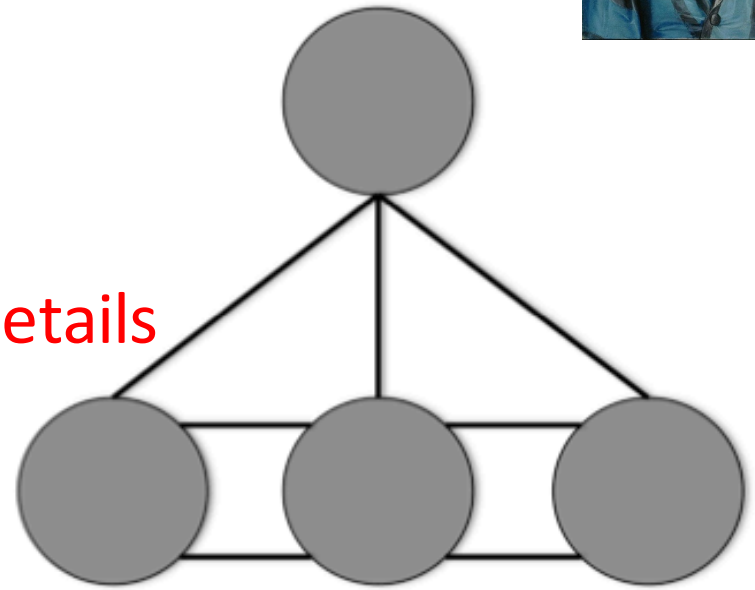
First Reported Use of Graph Theory

- Bridges of Königsberg problem (1735)
  - Known today as Kaliningrad
  - Two islands plus two mainland portions of city connected by 7 bridges
- Is it 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?

Poll:
Do you think that such a path exists?

What's Interesting About This

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

Implementing and using graphs

- Building graphs
  - Nodes
  - Edges
  - Stitching together to make graphs
- Using graphs
  - Searching for paths between nodes
  - Searching for optimal paths between nodes

Node & edge classes

```
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
```

Setting attributes

Accessing attributes

This creates a directed edge; to create an undirected edge, simply add another from dest to src

Common Representations of Digraphs

- Digraph is a directed graph
  - Edges pass in one direction only
  - Need to represent that collection of edges
- 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
  - Uses O(|nodes|**2) memory

destination

source

	A	B	C	D
A				1
B	1			
C		1		
D	1			1

A: [D]
B: [A]
C: [B]
D: [A, D]

- Adjacency list
  - Associate with each node a list of destination nodes that can be reached by one edge
  - Uses O(|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()))
```

Create from a set of nodes; initially no edges

Nodes represented as keys in dictionary

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

A directed weighted edge

Get list of destinations for source node from dict

Add new destination and edge weight to list

Class WeightedDigraph, 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
```

List of reachable nodes

Nodes reachable by a directed edge from a specific node

Using properties of dictionary to represent nodes

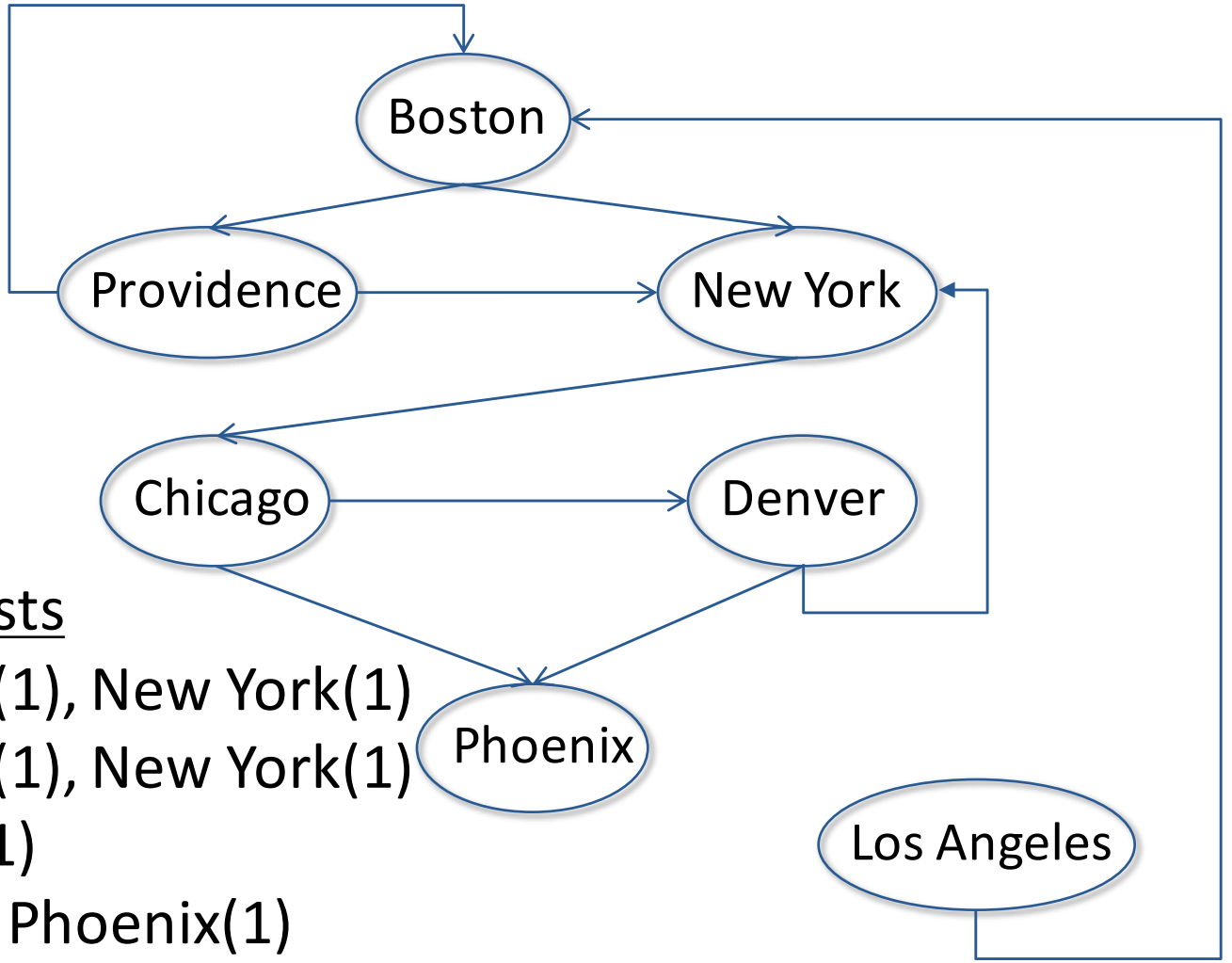
Build the Graph

```
def buildCityGraph():
    g = WeightedDigraph (('Boston', 'Providence', 'New York', 'Chicago',
                          'Denver', 'Phoenix', 'Los Angeles')) #Create 7 nodes
    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)
```

Since just using nodes as names, can use strings rather than creating node instances

An Example



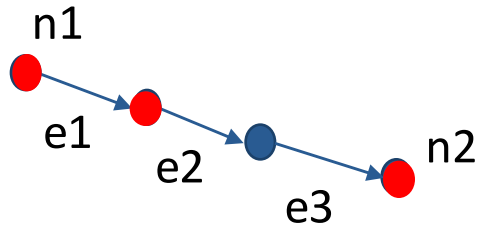
All nodes in graph

Adjacency Lists

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

A Classic Graph Optimization Problem

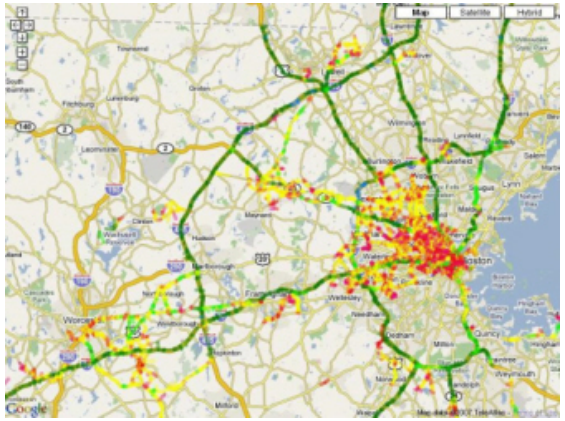
- Shortest (unweighted) path from n1 to n2
  - Shortest sequence of edges such that
    - Source node of first edge is n1
    - Destination of last edge is n2
    - For edges, e1 and e2, if e2 directly follows e1 in the sequence, the source of e2 is the destination of e1

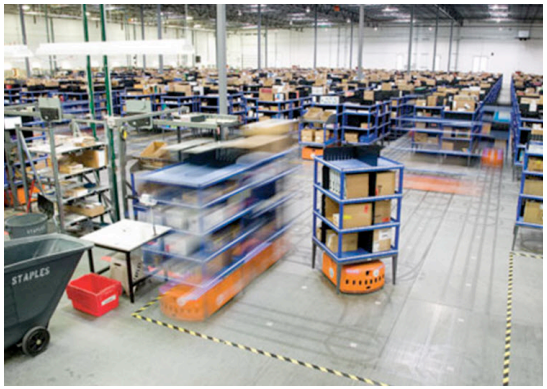


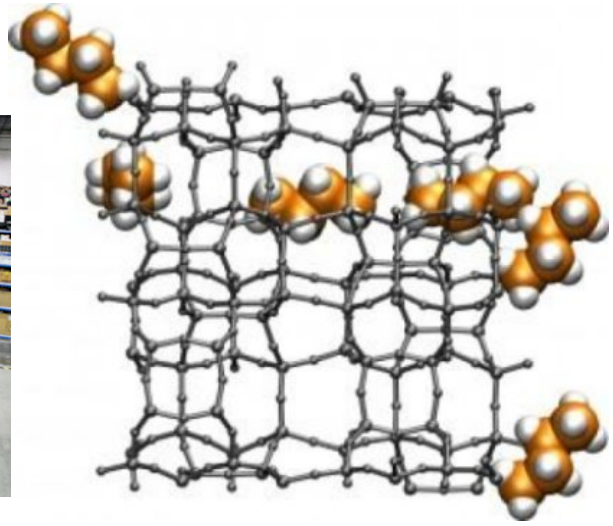
- Shortest weighted path
  - Minimize the sum of the weights of the edges in the path
- For this lecture, we are mainly going to focus on unweighted paths

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

Another example of **recursive thinking**!

Breadth First Search

- Start at an initial node (call it the current node)
- Consider all the edges that leave that node, in some order
- Follow the first edge
  - 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 either find goal node, or run out of options
  - When run out of edge options for current node, 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

Algorithm 1: Breadth-first Search (BFS)

```
def bfs(graph, start, end, toPrint = False):
    visited = {}
    pathQueue = [[start]]
    visited[start] = True
    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 visited:
                newPath = tmpPath + [nextNode]
                visited[nextNode] = True
                pathQueue.append(newPath)
    return None
```

Set of nodes already seen

Note: a list of paths to explore, each of which is a list of nodes

First in, first out

Next path in queue

Why can we stop here?

Add path in queue

Poll:
If we skip marking visited nodes, and just add paths to queue, does this still work?

Because we are using a FIFO queue, will explore all paths with n hops before any path with more than n hops; if don't find a path, return None

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

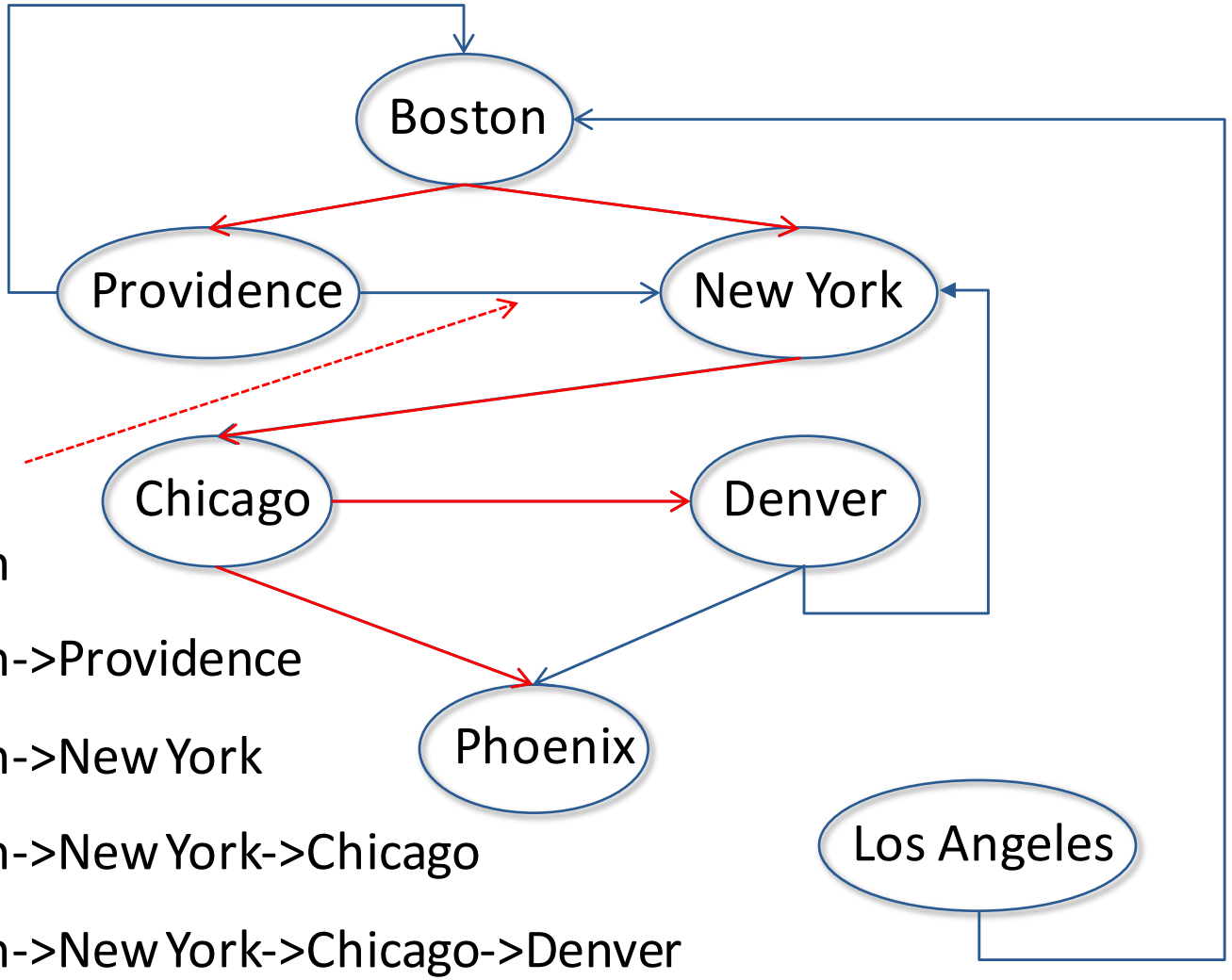
Shortest path from Boston to Phoenix is Boston->New York->Chicago->Phoenix

can stop as soon as find a successful path, since guaranteed to be shortest

Output (Boston to Pheonix)

Note that we skip a path that revisits a node, creating a loop

Note that we can skip a path that reaches an already visited node, as can't be shorter



Current BFS path: Boston

Current BFS path: Boston->Providence

Current BFS path: Boston->New York

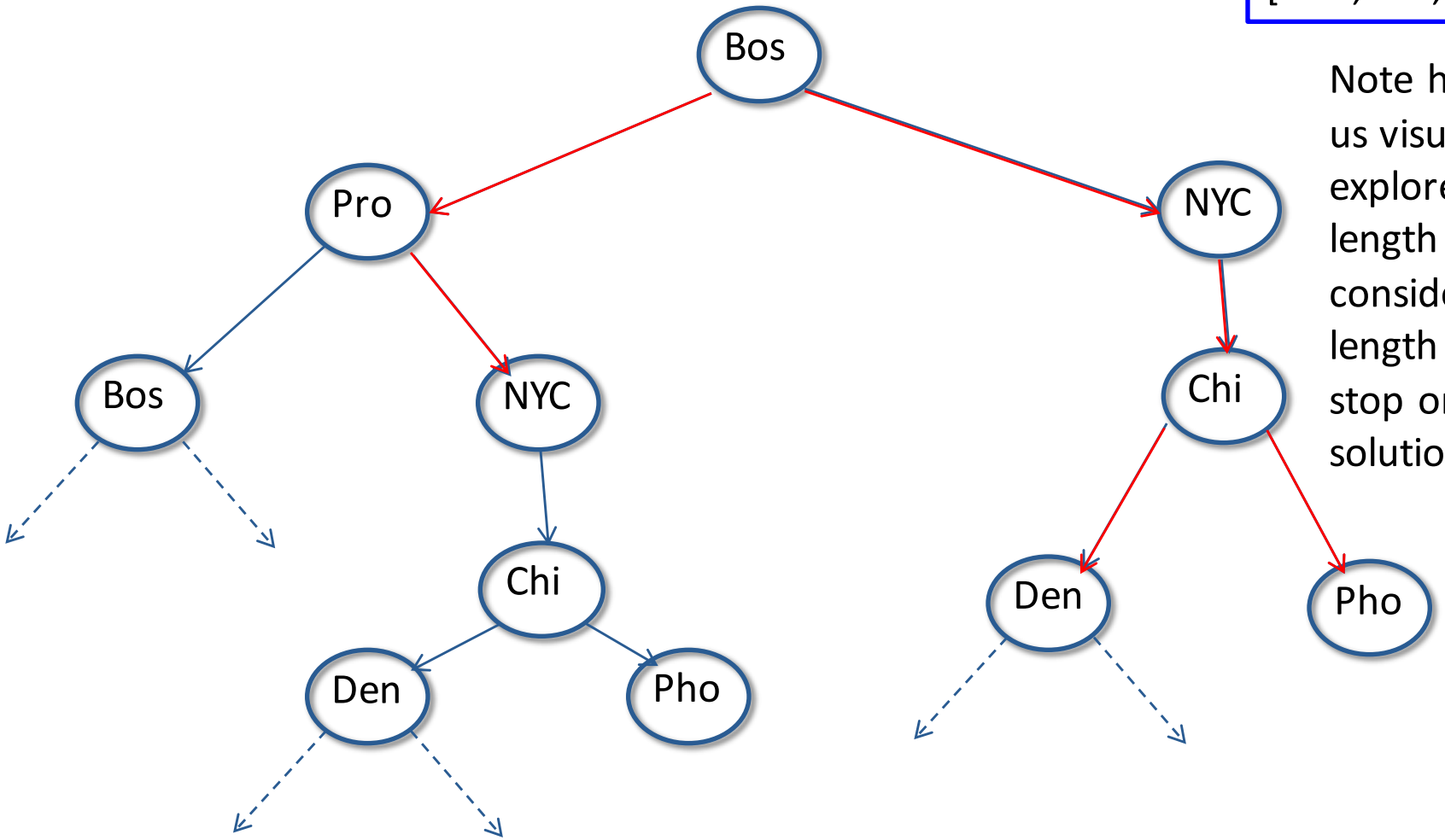
Current BFS path: Boston->New York->Chicago

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

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

Visualizing as a tree search

[BOS]
[BOS,PRO]
[BOS,NYC]
[BOS,PRO,NYC] #no loops
[BOS,NYC,CHI] #shorter
[BOS,NYC,CHI,DEN] path
[BOS,NYC,CHI,PHO]
[BOS,NYC,CHI,DEN,PHO]



Note how tree helps us visualize that we explore all paths of length n before considering paths of length n+1; so can stop once we find a solution

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 seen and save
- If not at goal node, repeat the process from new node
- Continue until either find goal node, or run out of options
  - When run out of edge options, backtrack to previous node, repeating this process
  - When run out of node options, backtrack to the previous node and try the next edge, repeating this process

Implementation like BFS, except

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

FIFO

LIFO

Algorithm 2: Depth-first Search (DFS)

```
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
```

Different from BFS, taking **last** path in queue

Need to check if better than current best path

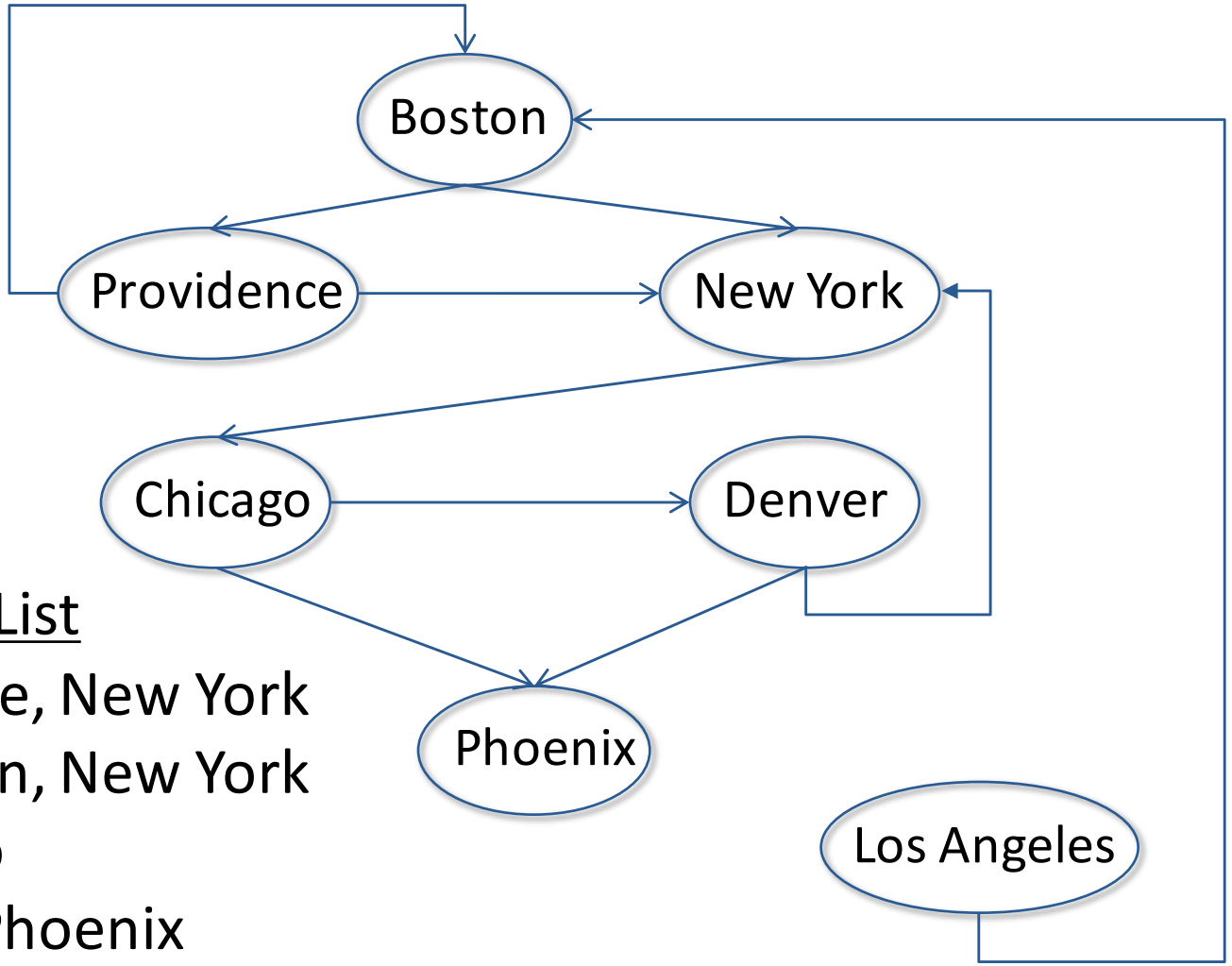
Skip to next iteration of loops

Only keep looking if not too long

But still adding new paths at end of queue

Poll:
If we don't check that nextNode already in path before adding new paths to queue, will this still work?

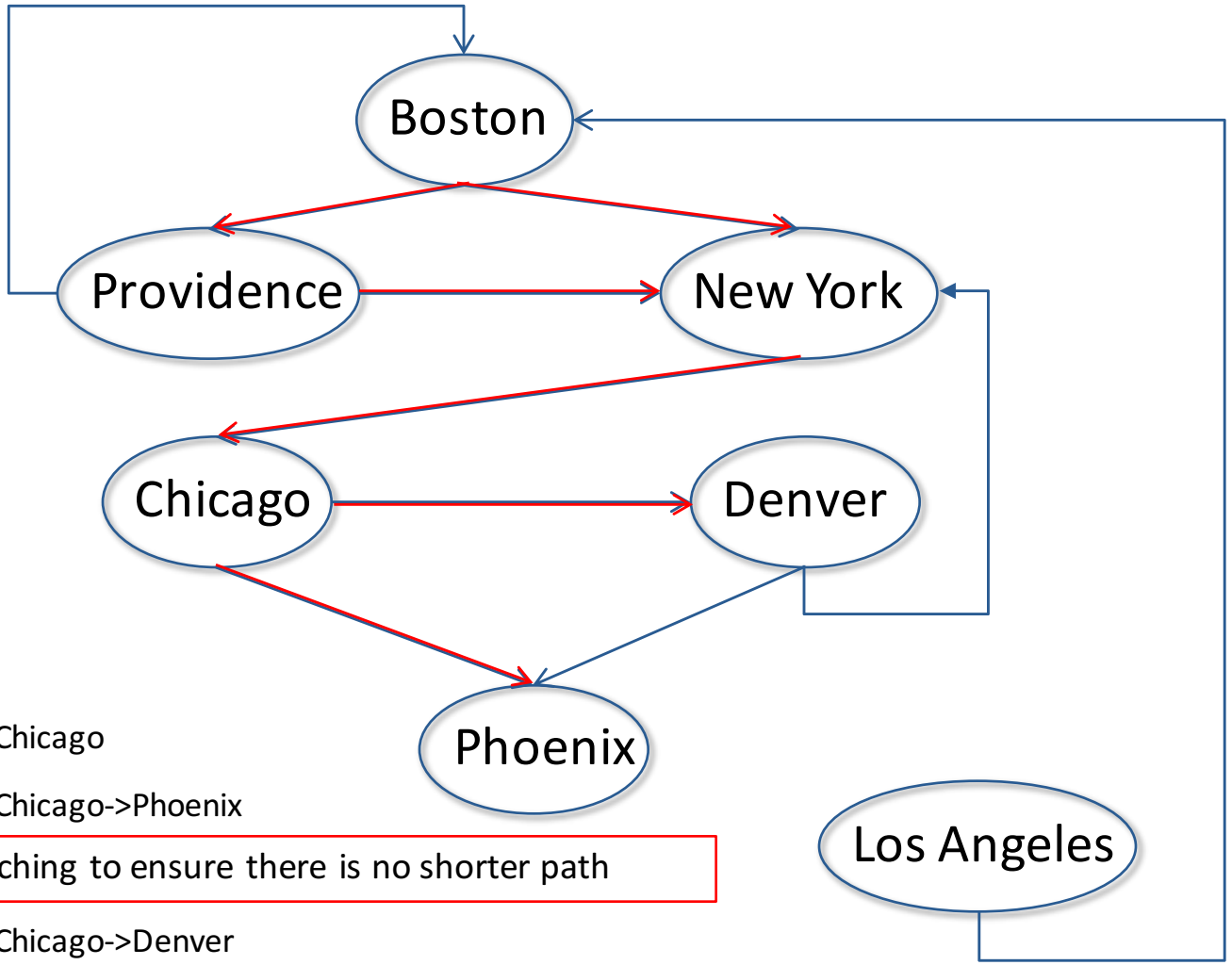
An Example



Adjacency List

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

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

Path found

Need to keep searching to ensure there is no shorter path

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

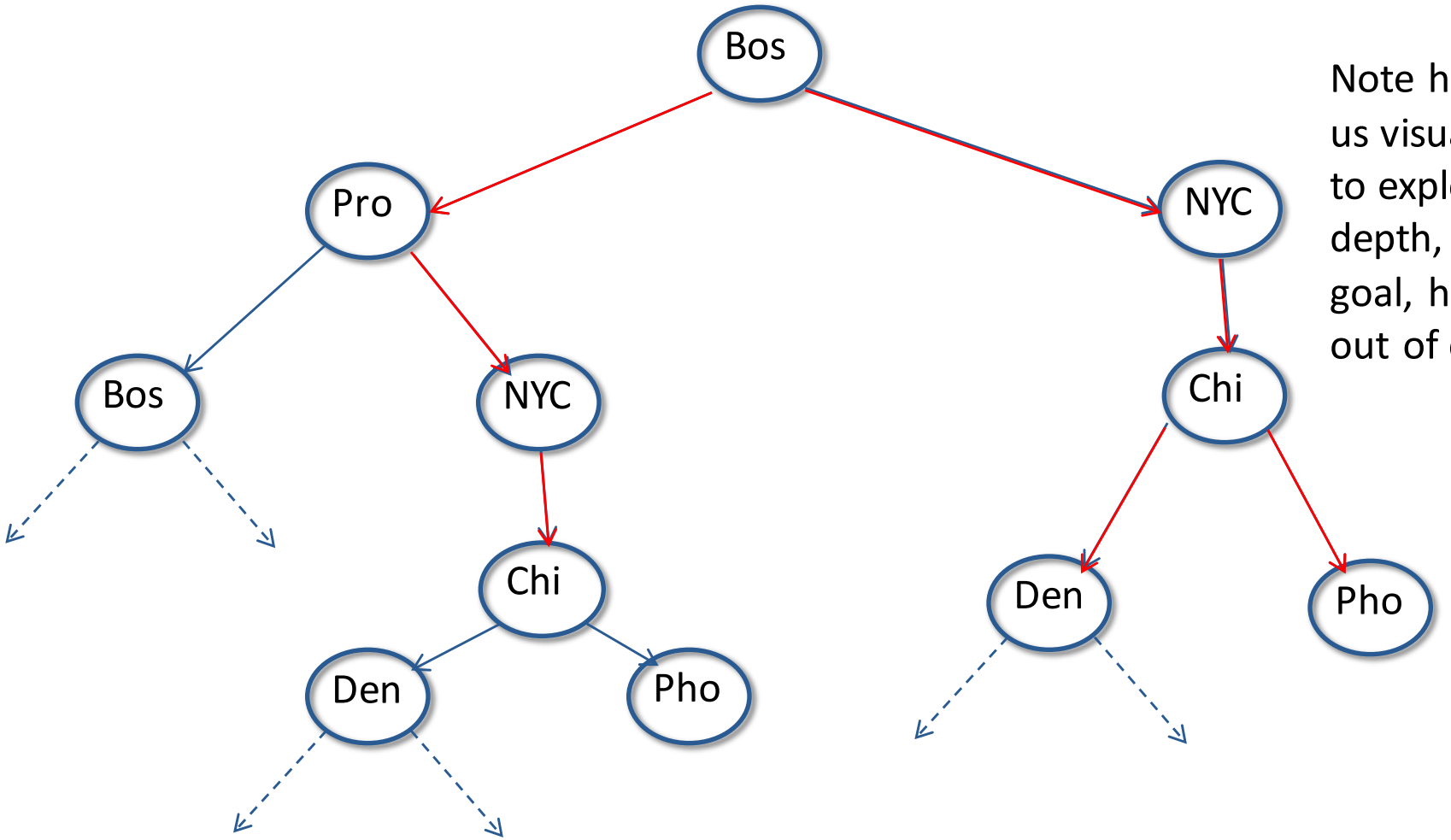
Current DFS path: Boston->Providence

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

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

Visualizing as a tree search

[]
[BOS,PRO]
[BOS,NYC,CHI,DEN]



Note how we remove from end of queue, not front

Note how tree helps us visualize that we try to explore one path in depth, until we find goal, hit a loop, or run out of options

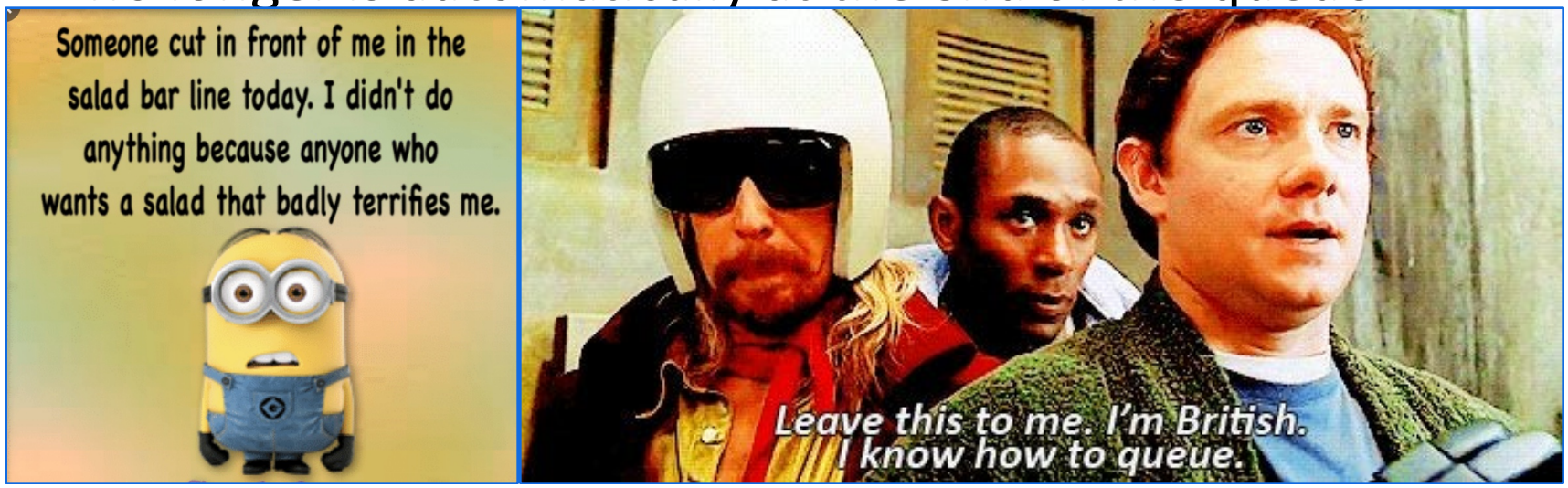
What About a Weighted Shortest Path

- Want to minimize the sum of the weights of the edges, not just the number of edges
- DFS can be easily modified to do this, if we assume that weights are non-negative numbers
  - But slow
- BFS is fast for unweighted graphs, but cannot work for weighted graphs, since shortest weighted path may have more than the minimum number of hops
- Gets us to Dijkstra's algorithm

Let's take a short break

Algorithm 3: Dijkstra's Algorithm

- Generalization of breadth-first search that does not require edges to have equal weight
- Uses a **priority** queue instead of a FIFO queue
  - Priority queue uses some numerical measure to insert new items in queue based on ordering (typically smallest first)
  - Still takes items from the front of the queue, but insertion no longer is automatically at the end of the queue



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 for a path from start node to that node
  - Initially zero for start node and infinity for all others
- predecessor: a dict mapping each node to previous 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 using *predecessor* by working backwards from end node

Outline of Algorithm

- For current node, choose an unvisited node with shortest distance from start node, (initially this is the start node)
  - This is the metric defining the priority queue
- Check each neighbor of current node (those reachable in one step from current node)
  - Calculate distance from starting node to each neighbor, using path passing through current node
  - Keep shortest distance path for each neighbor (and update *predecessor* and *distanceTo* if needed)
- Repeat for all nodes, until all are visited
- Will show implementation that assumes all edges have equal weights
  - Almost trivial to adapt to unequal weights

Initialization

```
def Dijkstra(graph, start, end, toPrint = False):
    """
    graph: an unweighted (all edges have weight 1) 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 not empty

```
while unvisited:
    # 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 pedagocical 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 #hops as d

        # Compare the newly calculated distance to the assigned.
        # Save the smaller distance and update predecssor.
        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

How would this change for weighted edges?

Counting hops

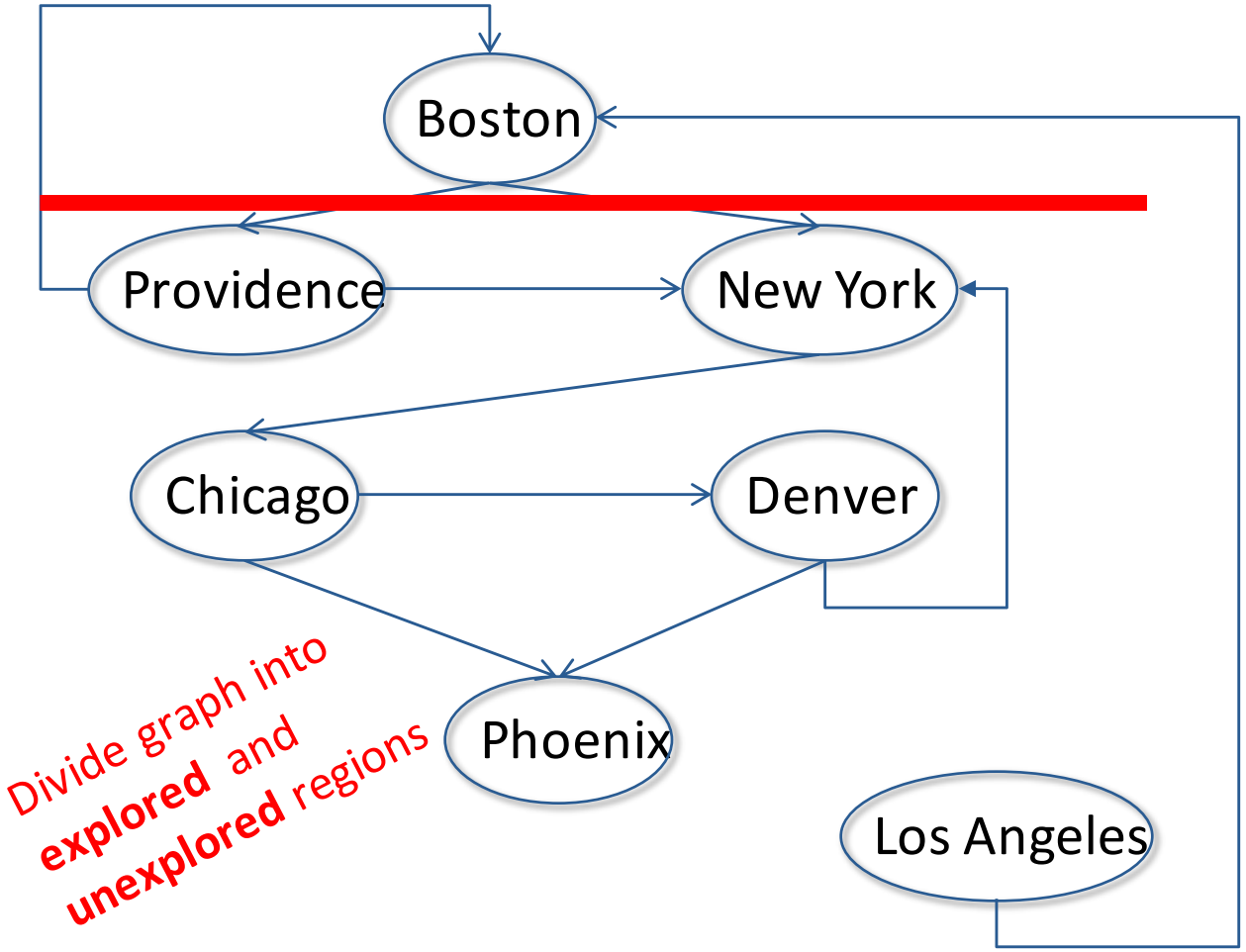
Need to check for shorter path, update information

Build Path from predecessor

```
#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
```

Back chaining to reconstruct path from end to start

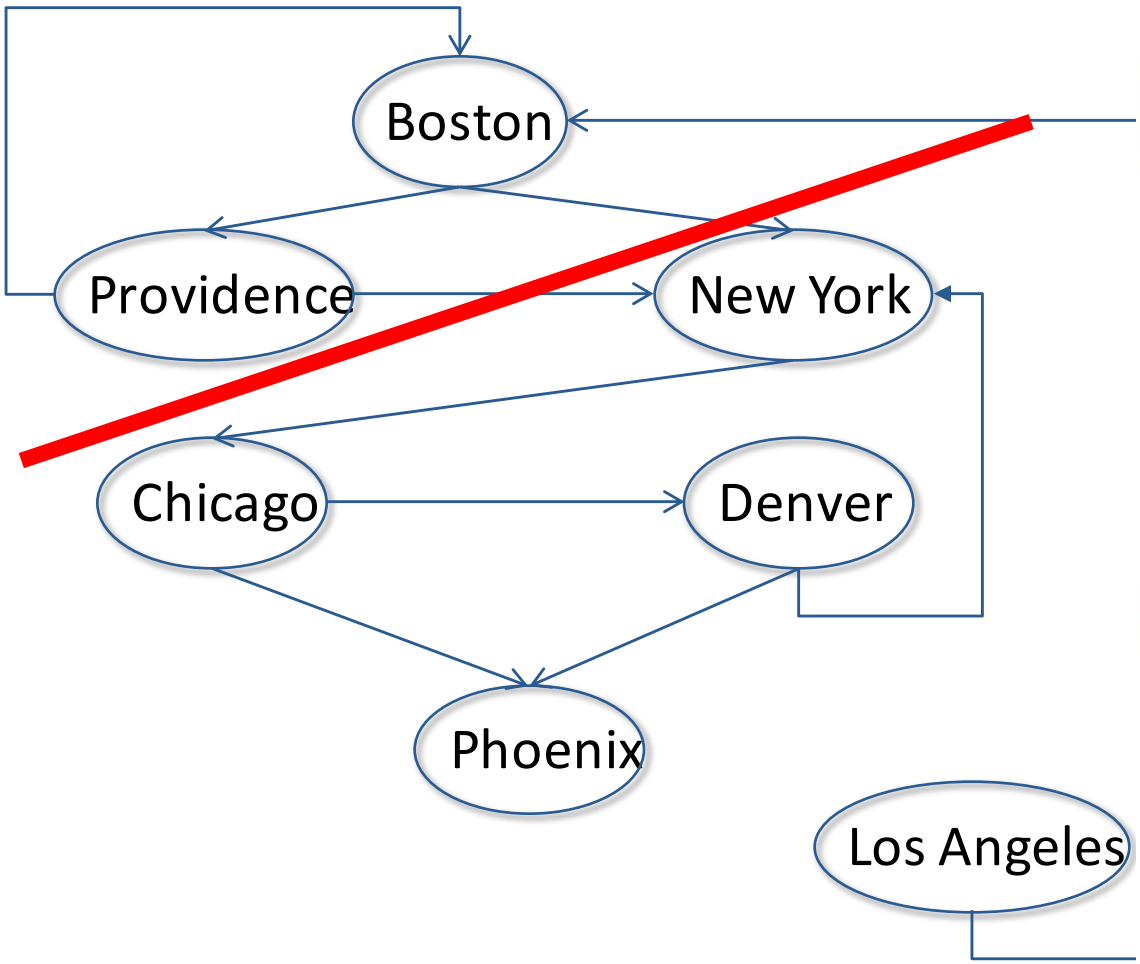
Example (Boston to Chicago)




```
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
```

For simplicity, just looking for path from Boston to Chicago in this example

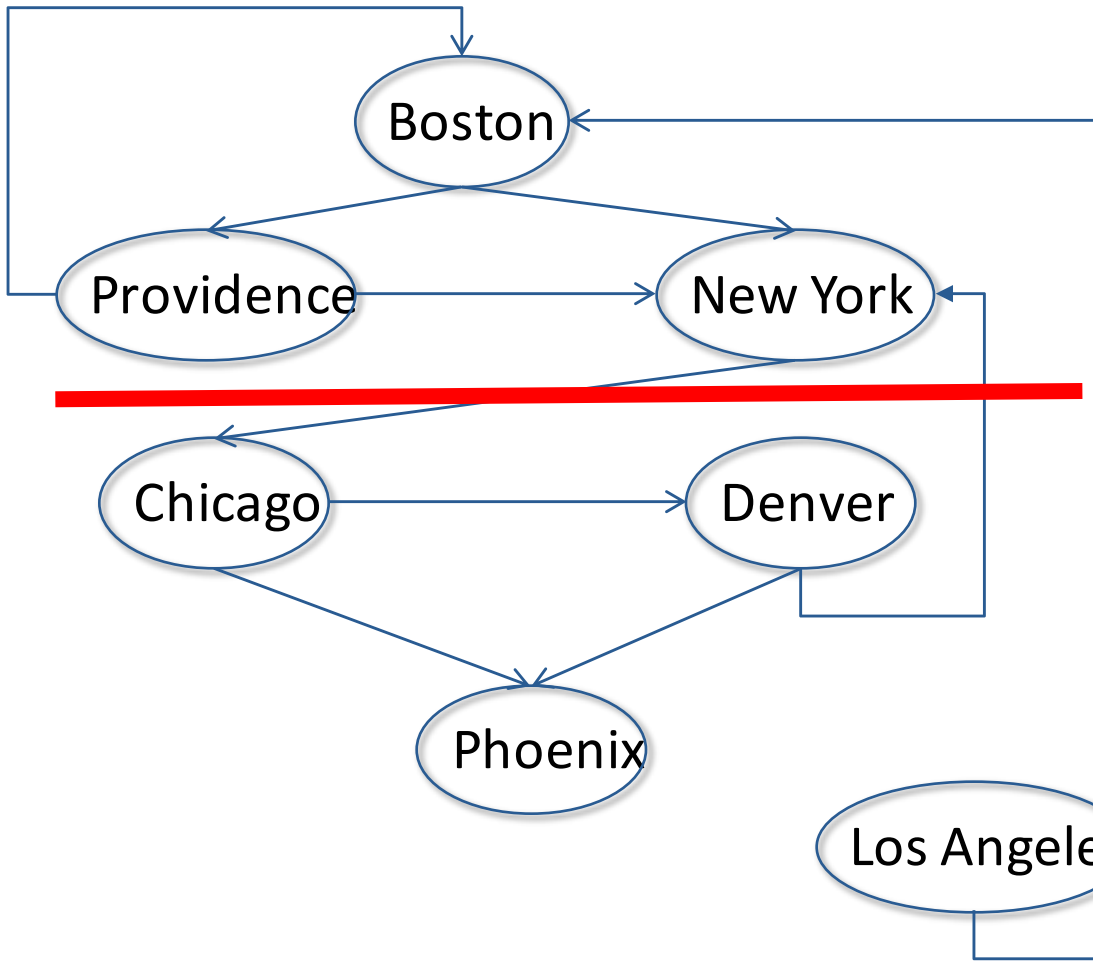
Move Graph Cut




```
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
```

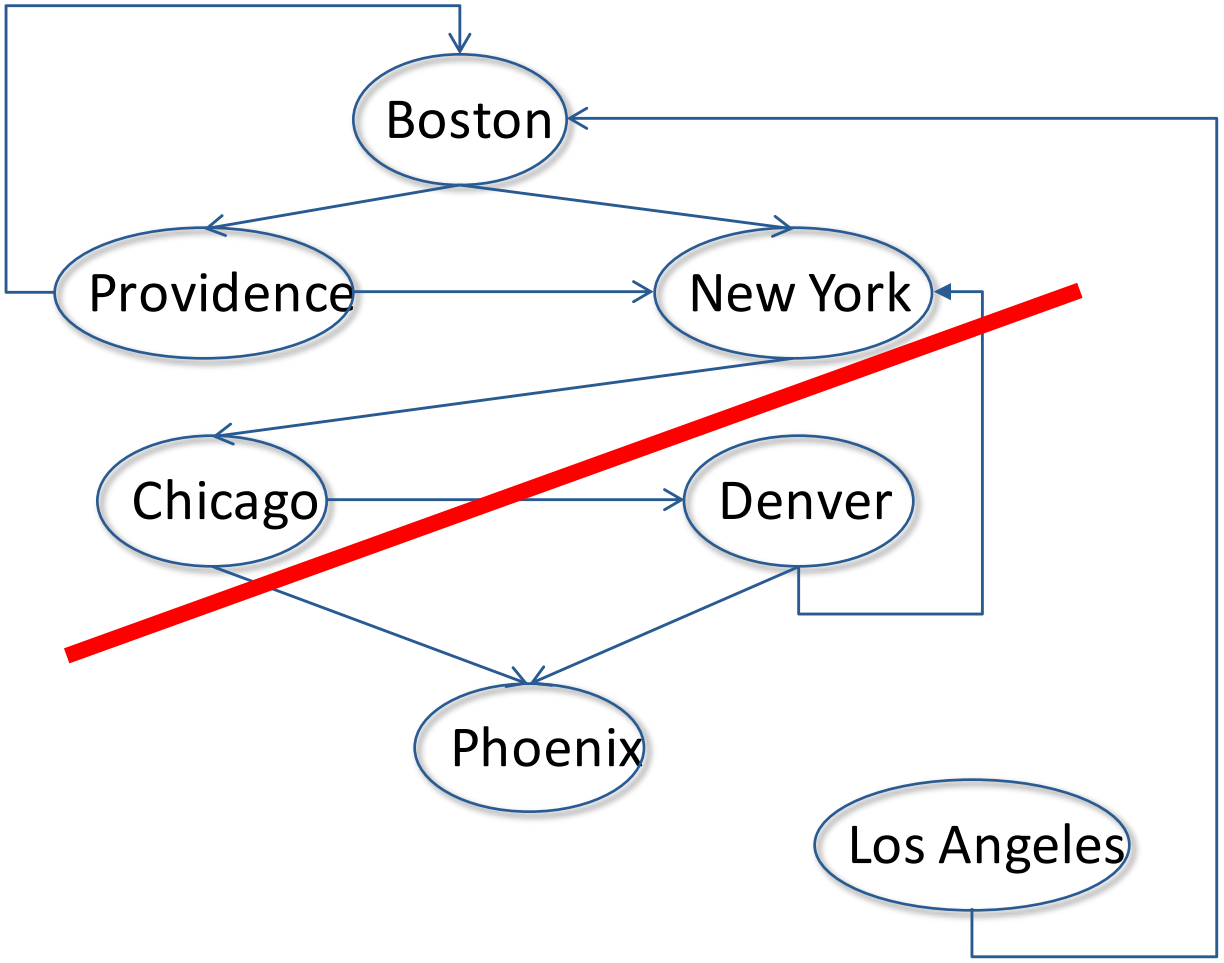
distanceTo and predecessor are guaranteed not to change for nodes above the cut

Move Graph Cut



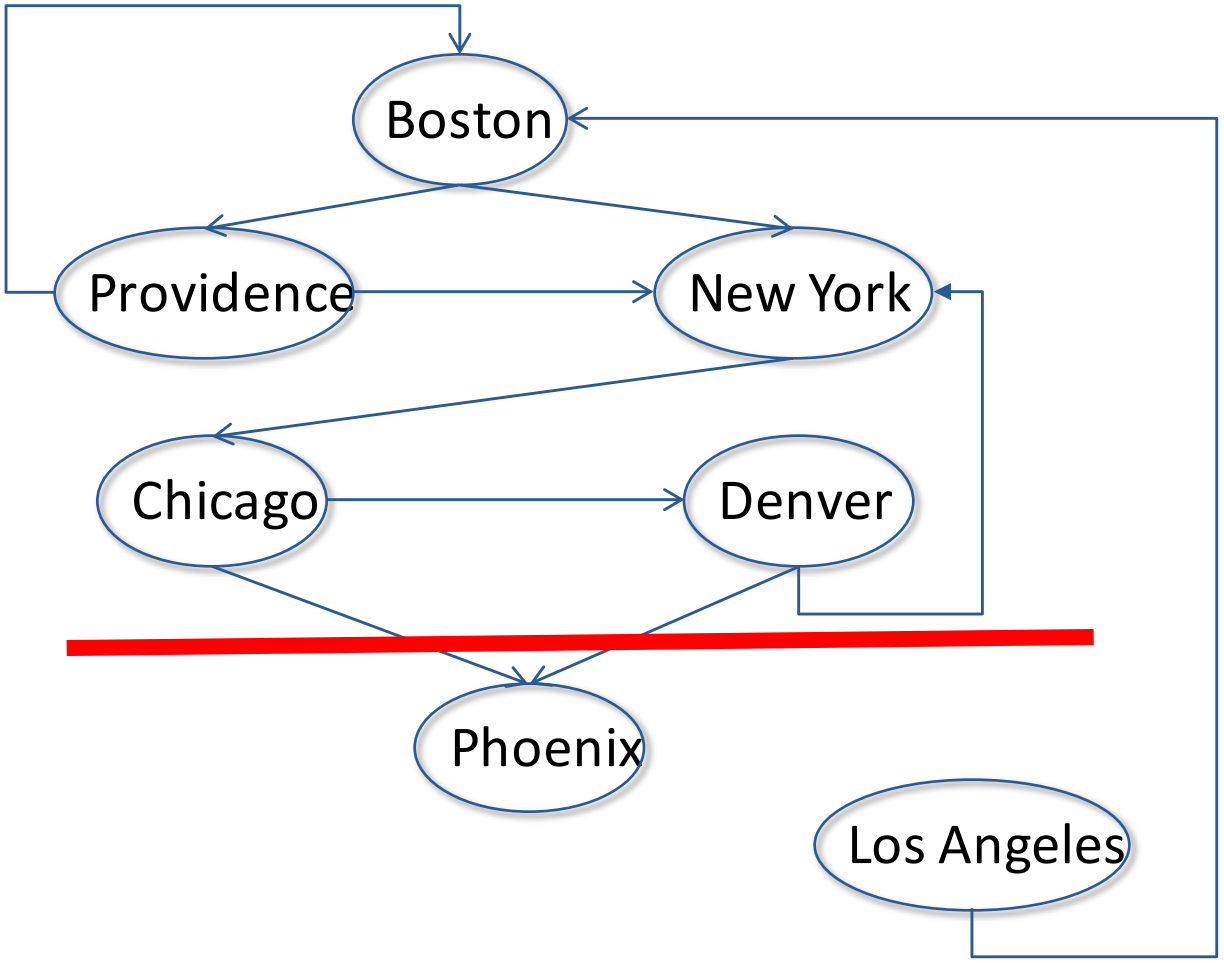

```
Value of current: New York
Value of distanceTo:
  Boston: 0
  Providence: 1
  New York: 1
  Chicago: inf
  Denver: inf
  Phoenix: inf
  Los Angeles: inf
Value of predecessor:
  Boston: None
  Providence: Boston
  New York: Boston
  Chicago: None
  Denver: None
  Phoenix: None
  Los Angeles: None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

Move Graph Cut



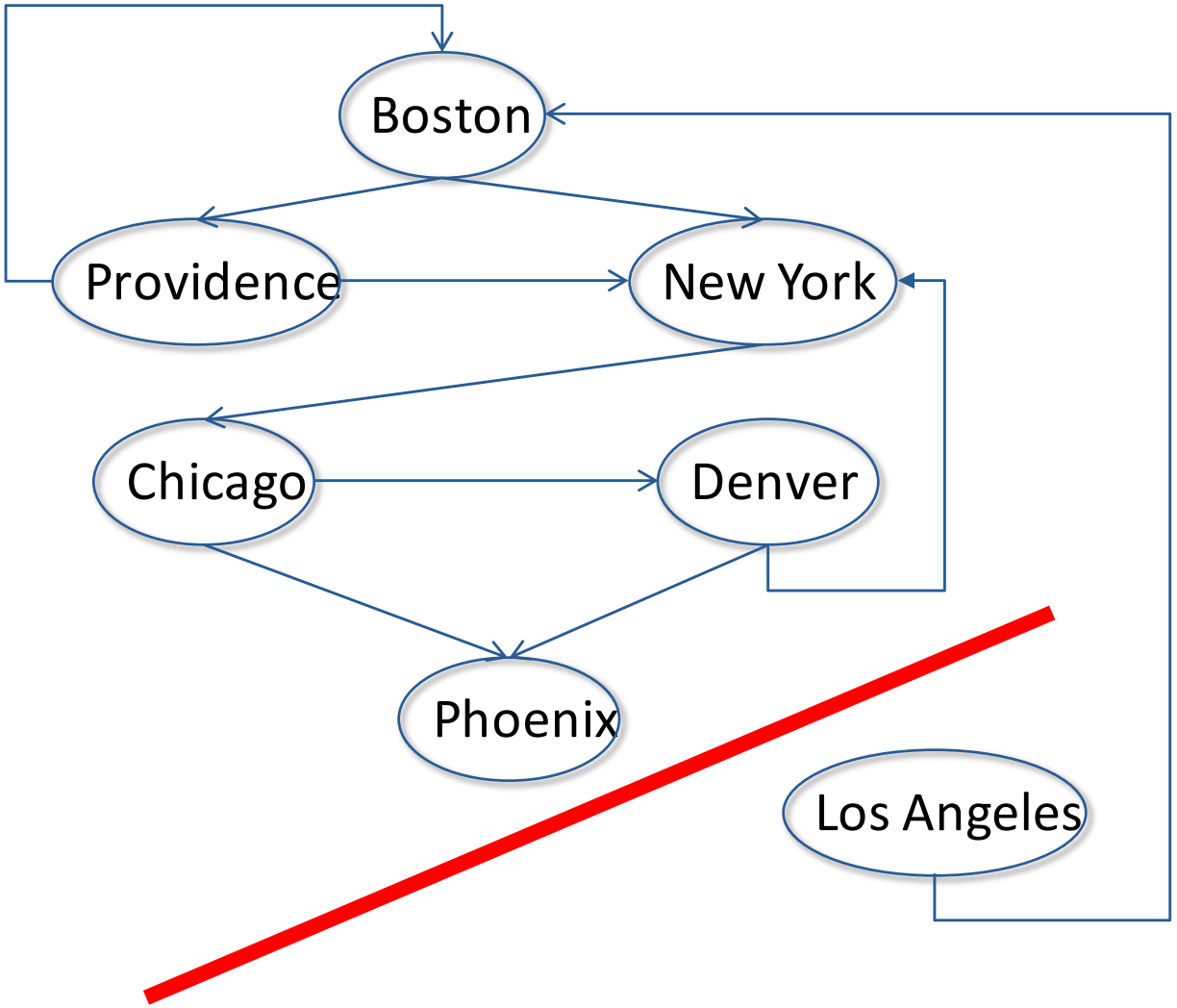

```
Value of current: Chicago
Value of distanceTo:
  Boston: 0
  Providence: 1
  New York: 1
  Chicago: 2
  Denver: inf
  Phoenix: inf
  Los Angeles: inf
Value of predecessor:
  Boston: None
  Providence: Boston
  New York: Boston
  Chicago: New York
  Denver: None
  Phoenix: None
  Los Angeles: None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

Move Graph Cut



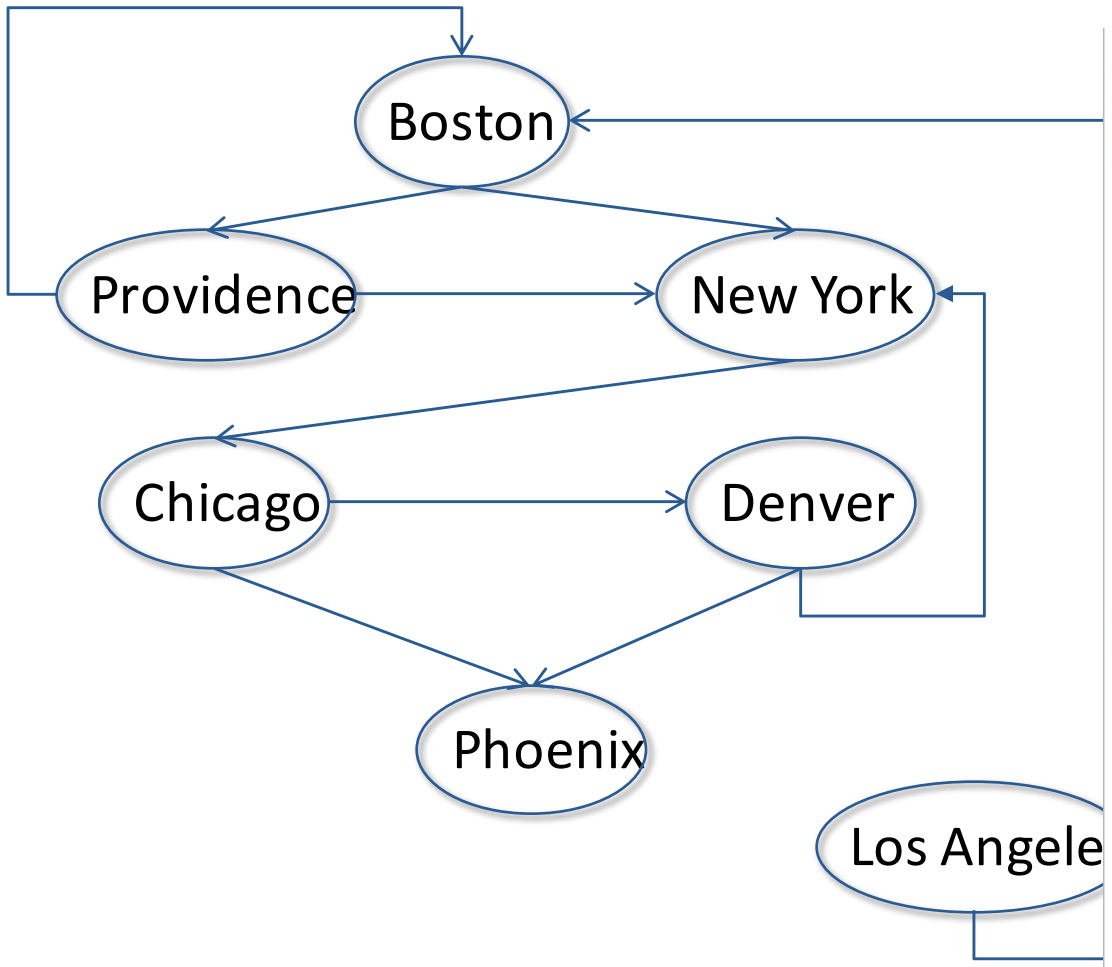

```
Value of current: Denver
Value of distanceTo:
  Boston: 0
  Providence: 1
  New York: 1
  Chicago: 2
  Denver: 3
  Phoenix: 3
  Los Angeles: inf
Value of predecessor:
  Boston: None
  Providence: Boston
  New York: Boston
  Chicago: New York
  Denver: Chicago
  Phoenix: Chicago
  Los Angeles: None
```

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



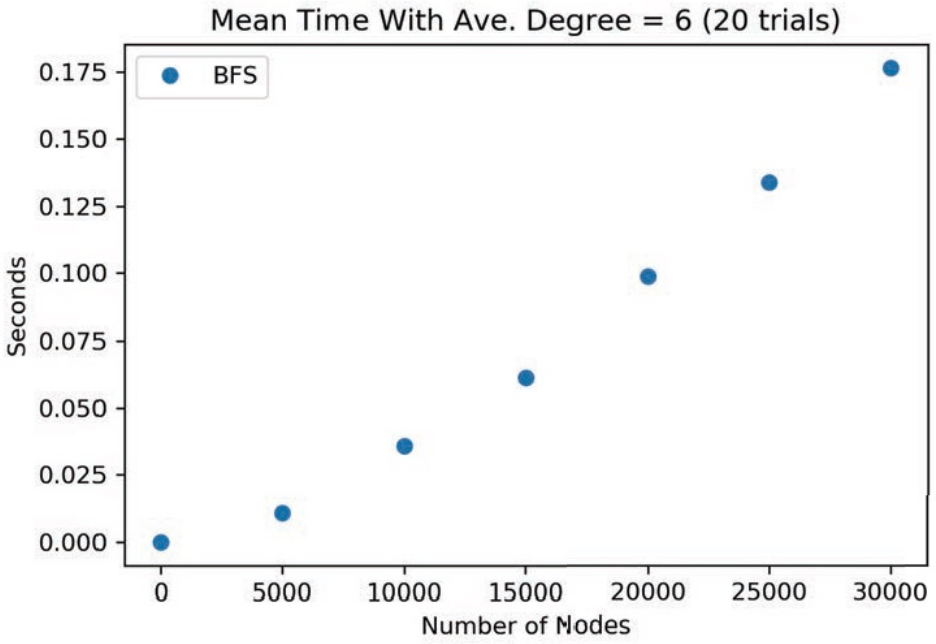

```
Value of current: Los Angeles
Value of distanceTo:
  Boston: 0
  Providence: 1
  New York: 1
  Chicago: 2
  Denver: 3
  Phoenix: 3
  Los Angeles: inf
Value of predecessor:
  Boston: None
  Providence: Boston
  New York: Boston
  Chicago: New York
  Denver: Chicago
  Phoenix: Chicago
  Los Angeles: None
```

```
Boston->New York->Chicago
```

Some Observations

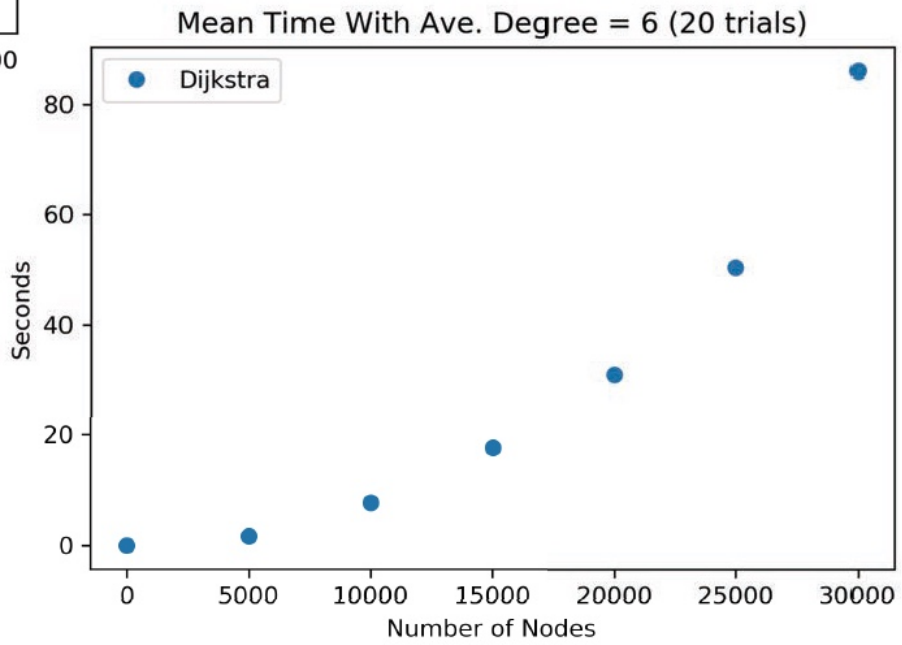
- BFS will stop once it finds a path
- DFS will stop once it has found a path and has explored all paths of shorter length
- Dijkstra visits all nodes in the graph, but not all paths
- So what does this suggest about efficiency of the algorithms, both in terms of complexity and in terms of practical application?

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 be modified to explore all paths
  - Dijkstra's algorithm (easily modified) does **not** need to explore all paths
    - loop over outbound edges, not just neighbors;
    - add weight of edge when considering alternative paths, not just count of hops
- Dijkstra's algorithm also easily solves the "all nodes shortest path problem"

Build a graph with weighted edges

```
def buildCityGraph_weighted():
    """Generate and return an example graph"""
    g = WeightedDigraph(('Boston', 'Providence', 'New York','Chicago',
                         'Denver', 'Phoenix', 'Los Angeles'))
    g.addEdge(Edge('Boston', 'Providence', 50))
    g.addEdge(Edge('Boston', 'New York', 190))
    g.addEdge(Edge('Providence', 'Boston', 50))
    g.addEdge(Edge('Providence', 'New York',180))
    g.addEdge(Edge('New York', 'Chicago', 790))
    g.addEdge(Edge('Chicago', 'Denver', 1000))
    g.addEdge(Edge('Chicago', 'Phoenix', 1750))
    g.addEdge(Edge('Denver', 'Phoenix', 820))
    g.addEdge(Edge('Denver', 'New York', 1780))
    g.addEdge(Edge('Los Angeles', 'Boston', 2990))
    return g

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

Dijkstra's Algorithm with Weights

```
while unvisited:
    # 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 pedagocical 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
    # current node.
    for edge in graph._edges[current]:
        alternativePathDist = distanceTo[current] + edge[1]
        neighbour = edge[0]
        # Compare the newly calculated distance to the assi
        # Save the smaller distance and update predecssor.
        if alternativePathDist < distanceTo[neighbour]:
            distanceTo[neighbour] = alternativePathDist
            predecessor[neighbour] = current

    # Remove the current node from the unvisited set.
    unvisited.remove(current)
```

Was looping over neighbours, which are nodes

Get neighbour

Add weight, not just 1

Example of Weighted Path

```
Value of current: Los Angeles
Value of distanceTo:
  Boston: 0
  Providence: 50
  New York: 190
  Chicago: 980
  Denver: 1980
  Phoenix: 2730
  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
```

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 is run using each node as start, and result is 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


That's all Folks!