

# 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

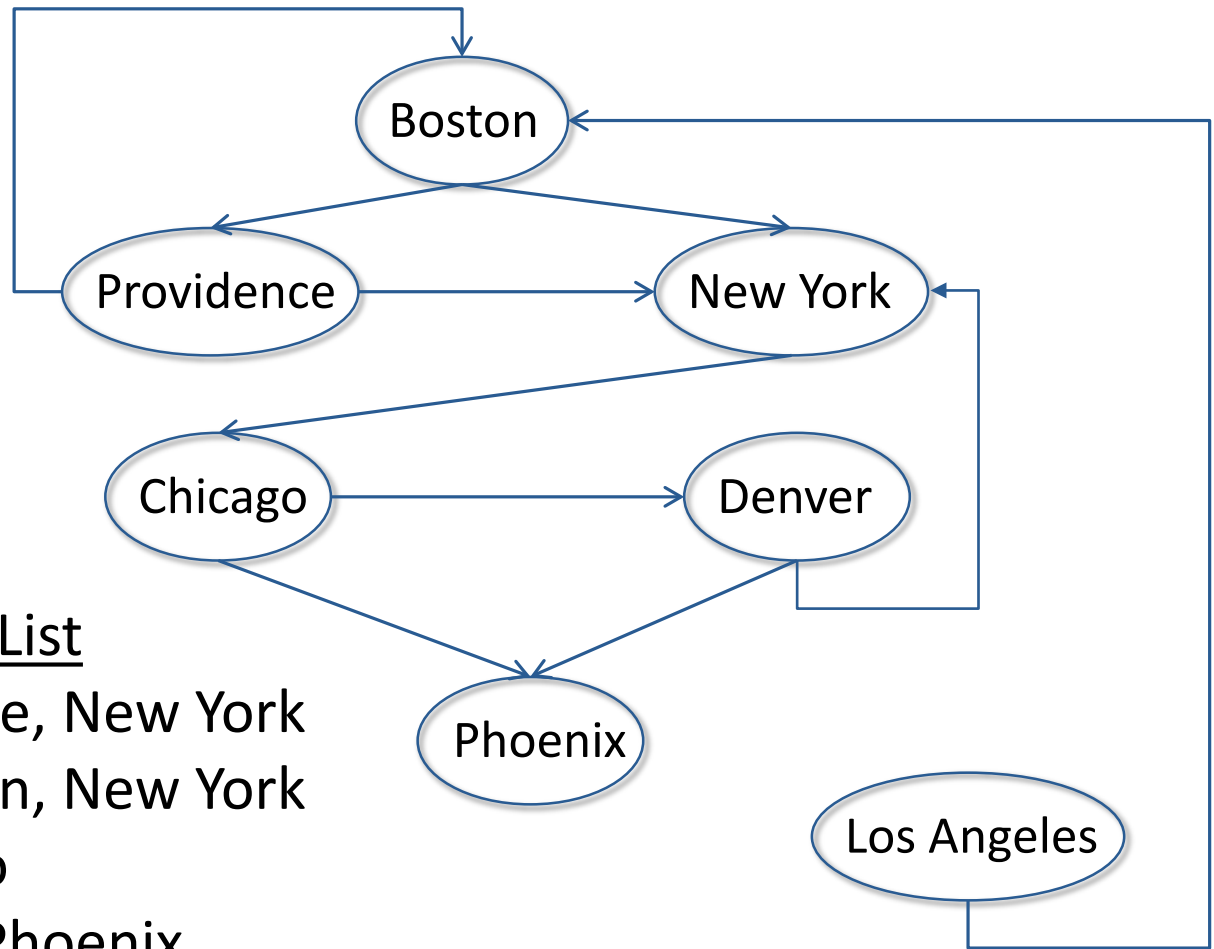
# Test DFS

---

```
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

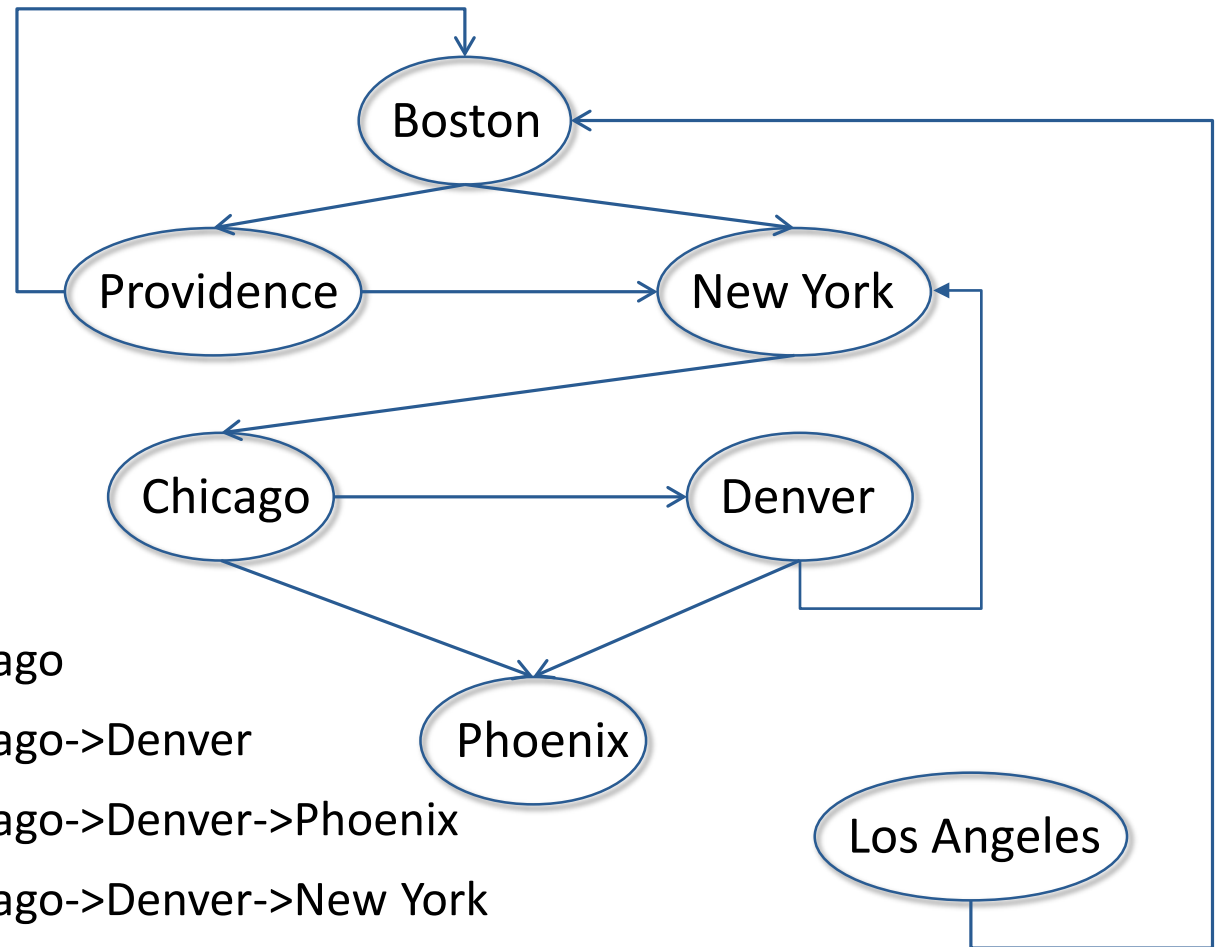
New York: Chicago

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