

Graph-theoretic Models

Eric Grimson

MIT Department Of Electrical Engineering and
Computer Science

오늘 강의 관련 읽을거리

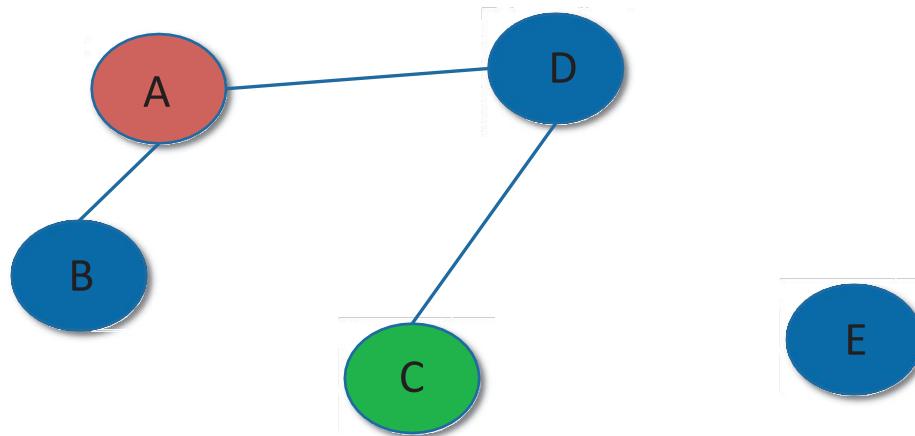
- Section 12.2

계산 모델

- 세상을 이해하고 실용적인 문제를 푸는 데 도움을 주는 프로그램
- 무엇을 먹을 것인지 결정하는 비공식적인 문제를 어떻게 최적화 문제로 연결해 생각하는지, 그리고 이를 푸는 프로그램을 어떻게 짜는지 지난 시간에 보았음
- 이번 시간에는 계산 모델의 일종인 그래프를 볼 예정

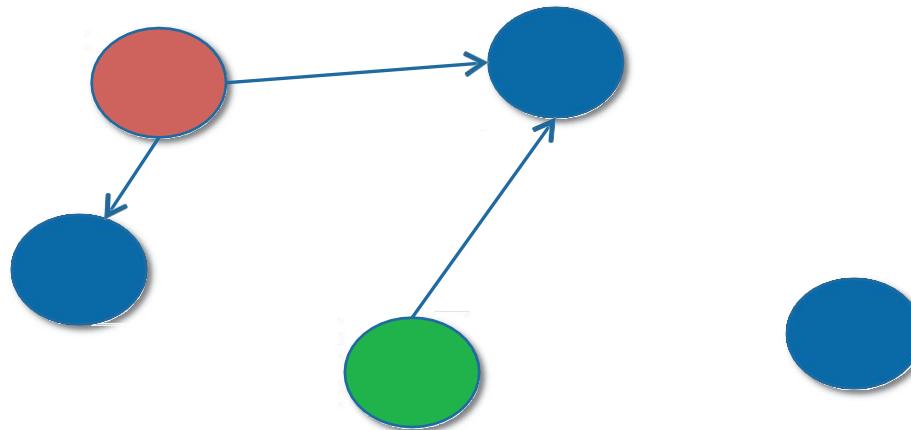
그래프란?

- 노드의 집합(꼭짓점)
 - 연관된 정보를 담을 수 있음
- 변의 집합(모서리) 노드의 쌍으로 구성
 - 무방향(그래프)
 - 유향(유향 그래프)
 - 시작점(부모)과 도착점(자식)
 - 무가중치 혹은 가중치



그래프란?

- 노드의 집합(꼭짓점)
 - 연관된 정보를 담을 수 있음
- 변의 집합(모서리) 노드의 쌍으로 구성
 - 무방향(그래프)
 - 유향(유향 그래프)
 - 시작점(부모)과 도착점(자식)
 - 무가중치 혹은 가중치

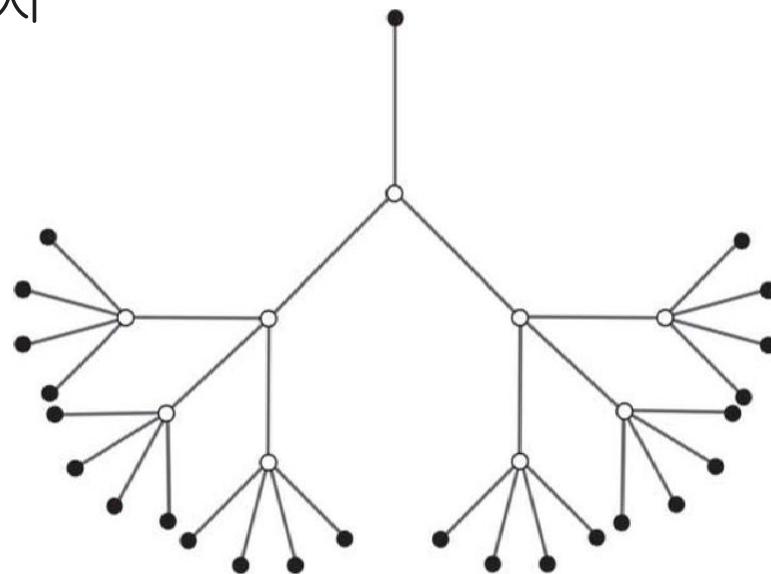


왜 그래프?

- 개체들 간의 유용한 관계를 담기 위해서
 - 파리와 런던 간 철도(레일)
 - 한 문자 내의 원자들이 서로 어떻게 연관되어 있는가
 - 조상 관계

트리: 중요하고 특별한 경우

- 꼭짓점 쌍이 하나의 경로로 연결되어 있는 유향
그래프의 특별한 경우
 - 냅색 문제를 풀 때 사용했던 탐색 트리를
떠올려보자



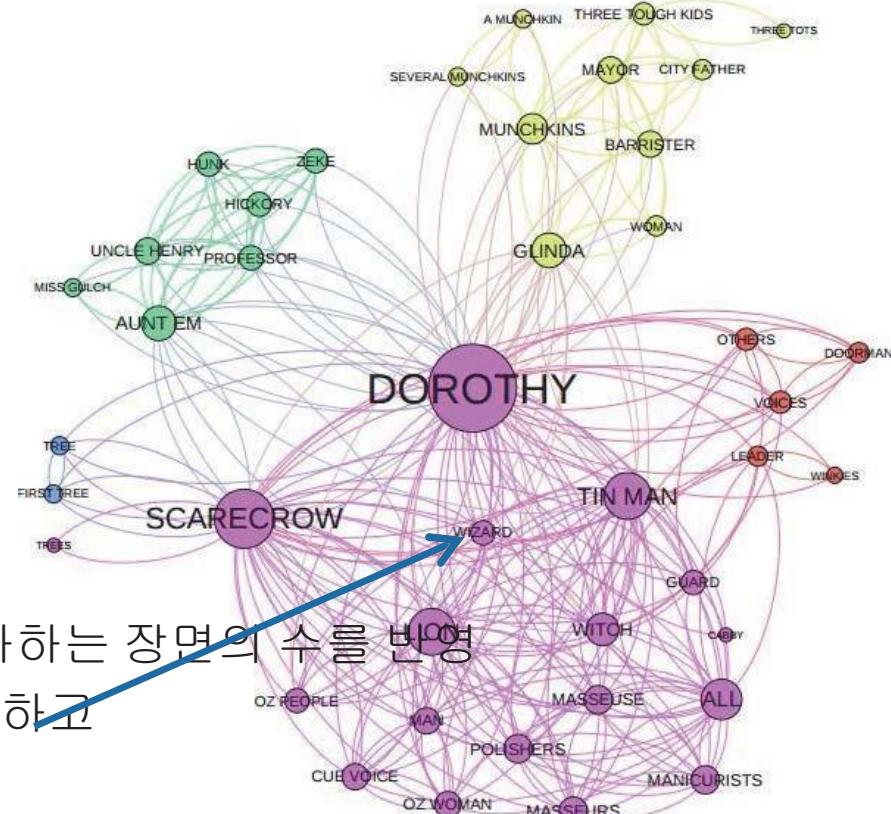
왜 그래프가 유용한가

- 세상은 관계 기반의 네트워크로 가득참

- 컴퓨터 네트워크
- 교통 네트워크
- 금융 네트워크
- 상하수 네트워크
- 정치 네트워크
- 범죄 네트워크
- 사회 네트워크
- 기타

“오즈의 마법사” 분석 :

- 노드의 크기는 캐릭터가 대사하는 장면의 수를 반영
- 군집의 색은 군집 안에서는 하고
밖에서는 하지 않는
자연스러운 상호작용을 반영



Wizard of Oz dialogue map © Mapr.com. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

왜 그래프가 유용한가

- 그래프가 네트워크 내 요소들 간의 관계만 담고 있는 것이 아니라, 구조를 통한 추론도 도와줌
 - 요소 간 경로의 배열 찾기 - A에서 B로 가는 경로가 있는가
 - 요소 간 최소 비용 경로 찾기 (최단 경로 문제)
 - 그래프를 연결된 요소들의 집합으로 분할하기 (그래프 분할 문제)
 - 연결된 요소들의 집합으로 분할하는 가장 효율적인 방법 찾기 (최소 컷 최대 유량 문제)

그래프 이론은 매일 제 시간을 절약해줍니다



Map image © source unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

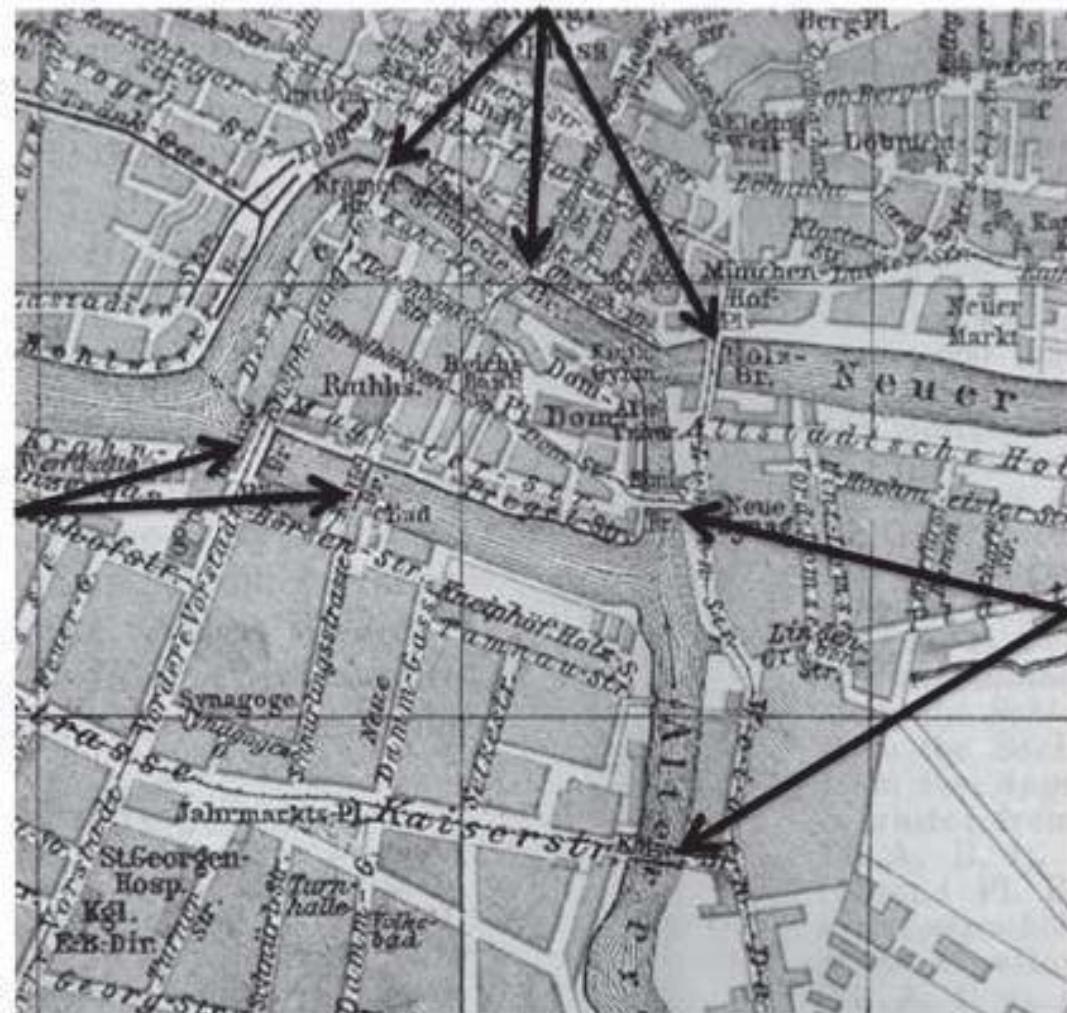
제가 오피스로 가는 길

- 유향 그래프를 사용한 도로망 모델링
 - 꼭짓점: 길이 끝나거나 교차하는 지점
 - 변: 점들을 연결하는 선
 - 각 변은 가중치를 가짐
 - 그 변을 따라 시작점에서 도착점까지 걸리는 예상 소요 시간
 - 시작점에서 도착점까지 거리
 - 시작점에서 도착점까지 평균 속도
- 그래프 최적화 문제 풀기
 - 제 집에서 오피스까지 최소 가중치 경로



최초 그래프 이론 사용 사례

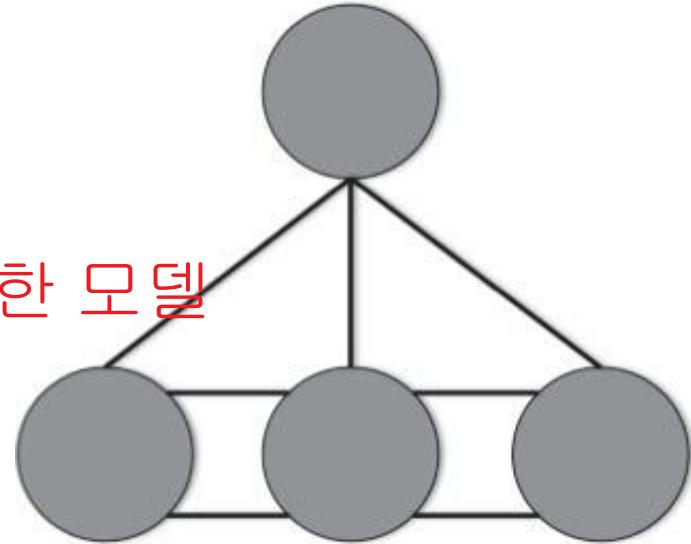
- 쾨니히스베르크 다리 (1735)
- 7개의 다리를 딱 한번씩만 걸어서 모두 횡단할 수 있을까?



Map image © source unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

레온하르트 오일러의 모델

- 각 섬을 꼭짓점으로
- 각 다리를 무방향 변으로
- 무의미한 정보는 버리고 추상화한 모델
 - 섬의 크기
 - 다리의 길이
- 각 변을 딱 한 번씩만 포함하는 경로가 있는가?
 - 없다!



그래프 구현과 사용

- 그래프 만들기
 - 꼭짓점
 - 변
 - 그래프를 만들기 위해 연결
- 그래프 사용
 - 꼭짓점 간 경로 탐색
 - 꼭짓점 간 최적 경로 탐색

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() + '->'\n             + self.dest.getName()
```

유향 그래프의 일반적인 표현

- 유향 그래프란 방향이 있는 그래프
 - 변은 오직 한 방향으로만 갈 수 있음
- 인접 행렬
 - 행: 시작점
 - 열: 도착점
 - $\text{Cell}[s, d] = 1$, s에서 d로가는 변이 존재하는 경우
 $= 0$, 그렇지 않은 경우
 - 유향 그래프에선, 행렬은 대칭 행렬이 아님
- 인접 리스트
 - 각 꼭짓점별 도착점 리스트와 연관

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 represented as keys in dictionary

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

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)
```

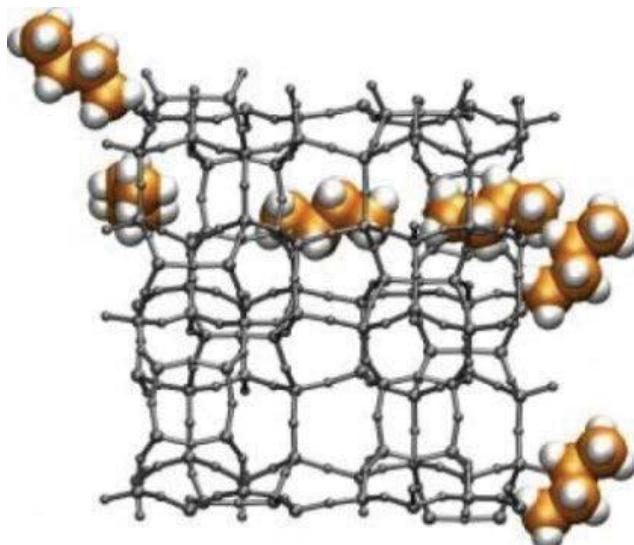
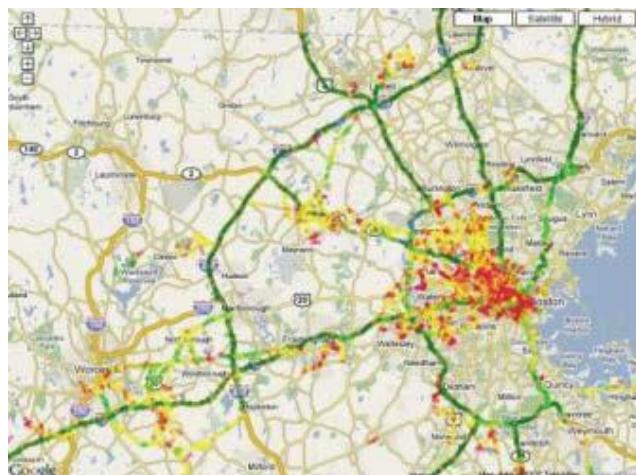
- 그래프는 변과 연관된 방향성을 가지지 않음
 - 변은 양 방향을 모두 허용
- 왜 그래프가 유향 그래프의 하위 클래스인가?
- 대체 규칙을 기억하는가?
 - 클라이언트 코드가 상위 타입 인스턴스를 사용해도 잘 작동한다면, 하위 타입 인스턴스를 상위 타입 인스턴스로 대체해도 잘 작동해야 함
- 유향 그래프를 사용해서 작동하는 프로그램이면 그래프를 사용해도 잘 작동함 (하지만 반대는 아님)

전형적인 그래프 최적화 문제

- n_1 에서 n_2 로 가는 최단 경로
 - 다음을 만족하는 변의 최단 배열
 - 첫 번째 변의 시작점을 n_1
 - 마지막 변의 도착점을 n_2
 - 변 e_1, e_2 에 대해서, 배열에서 e_2 가 e_1 다음에 나온다면 e_2 의 시작점은 e_1 의 도착점
- 최소 가중치 경로
 - 경로 내 변의 가중치의 합을 최소화

최단 경로 문제 몇 가지

- 한 도시에서 다른 도시로 가는 경로 찾기
- 통신망 네트워크 설계
- 화학적 미궁에서 분자의 경로 찾기
- ...



Images © sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

예시

인접 리스트

Boston: Providence, New York

Providence: Boston, New York

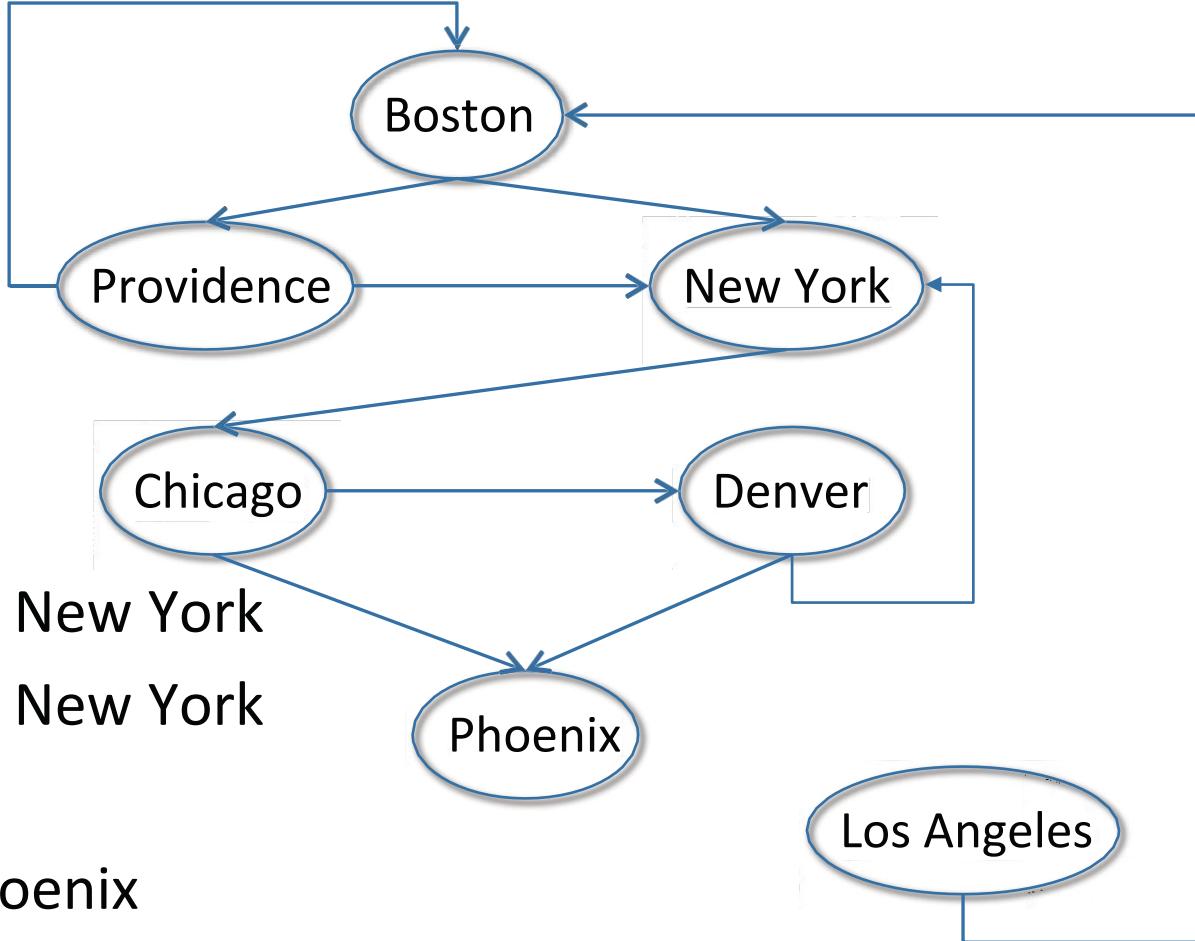
New York: Chicago

Chicago: Denver, Phoenix

Denver: Phoenix, New York

Los Angeles: Boston

Phoenix:



그래프 만들기

```
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')))
```

최단 경로 찾기

- 알고리즘 1, 깊이 우선 탐색(DFS)
- 탐색 트리의 왼쪽 우선 깊이 우선 방법과 비슷 (강의 2)
- 가장 큰 차이점은 그래프는 순환할 수 있다는 것,
따라서 무한 루프를 피하기 위해 지나간 경로를
기억해야 함

분할 정복을 사용한다는 것을 알아두세요 :
시작점에서 중간점까지 경로를 찾고, 중간점에서 도착점까지
경로를 찾을 수 있다면, 두 경로의 결합은 시작점에서 도착점까지
전체 경로임

깊이 우선 탐색

- 시작점에서 시작
- 특정한 순서로 시작점에서 나가는 모든 변을 고려
 - 첫 번째 변을 따라가고, 도착점인지 확인
 - 아니라면, 새로운 꼭짓점에서 과정을 반복
 - 도착점을 찾거나, 선택지가 없어질 때까지 반복
 - 선택지가 없어지면, 이전 꼭짓점으로 돌아가서 다음 변에 대해 과정을 반복



깊이 우선 탐색 (DFS)

```
def DFS(graph, start, end, path, shortest, toPrint = False):  
    [start]  
    if toPrint:  
        print('Current DFS path:', printPath(path))  
        return path  
    for node in graph.childrenOf(start):  
        if start == end:  
            ... returning to this point in the  
            recursion to try next node  
        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
```



```
def shortestPath(graph, start, end, toPrint = False):  
    end, [], None, toPrint)
```

DFS는 wrapper function에서
호출 shortestPath

적절한 재귀 사용

적절한 추상화 제공

DFS 테스트

```
def testSP(source, destination):      g = buildCityGraph(DiGraph)
    sp = shortestPath(g, g.getNode(source), g.getNode(destination)
                      toPrint = True)
    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')
```

예시

인접 리스트

Boston: Providence, New York

Providence: Boston, New York

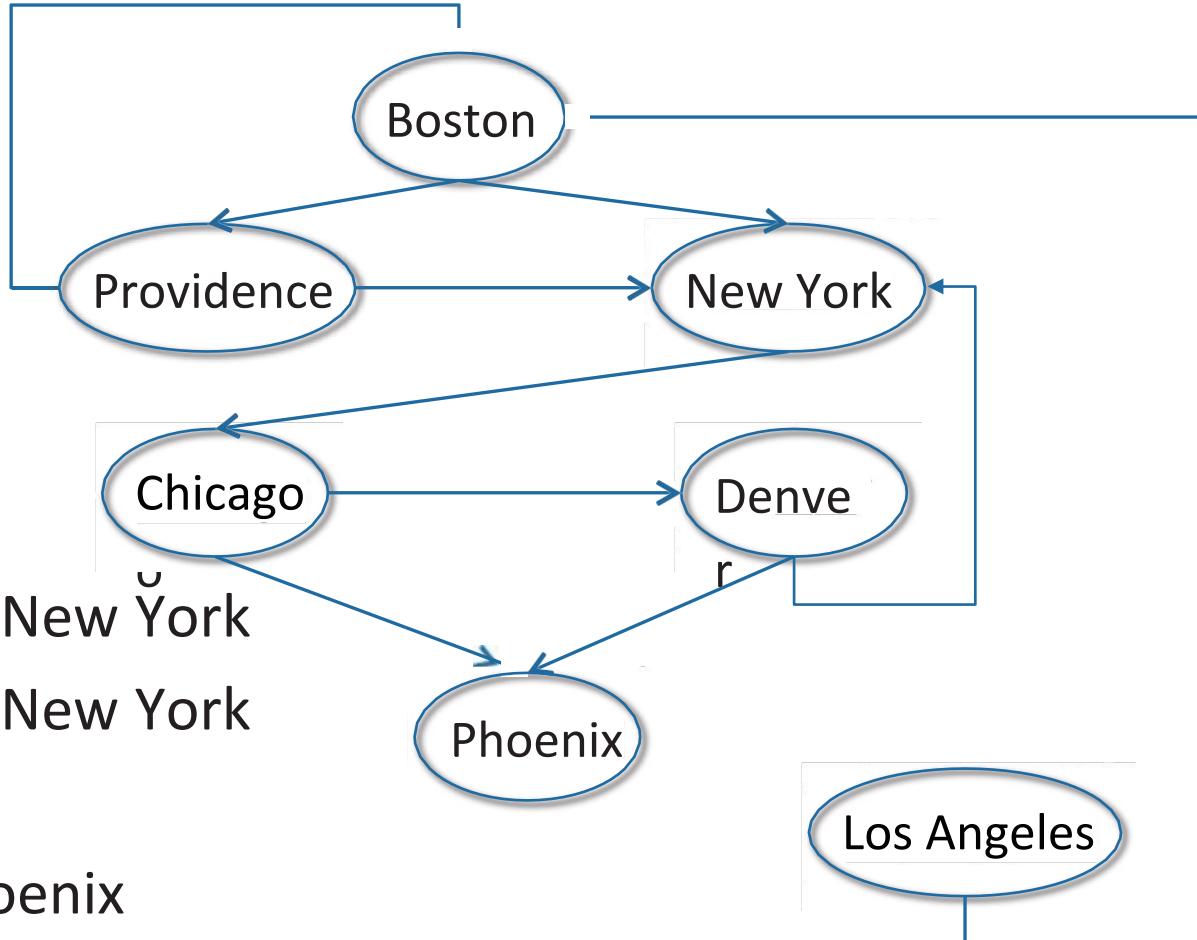
New York: Chicago

Chicago: Denver, Phoenix

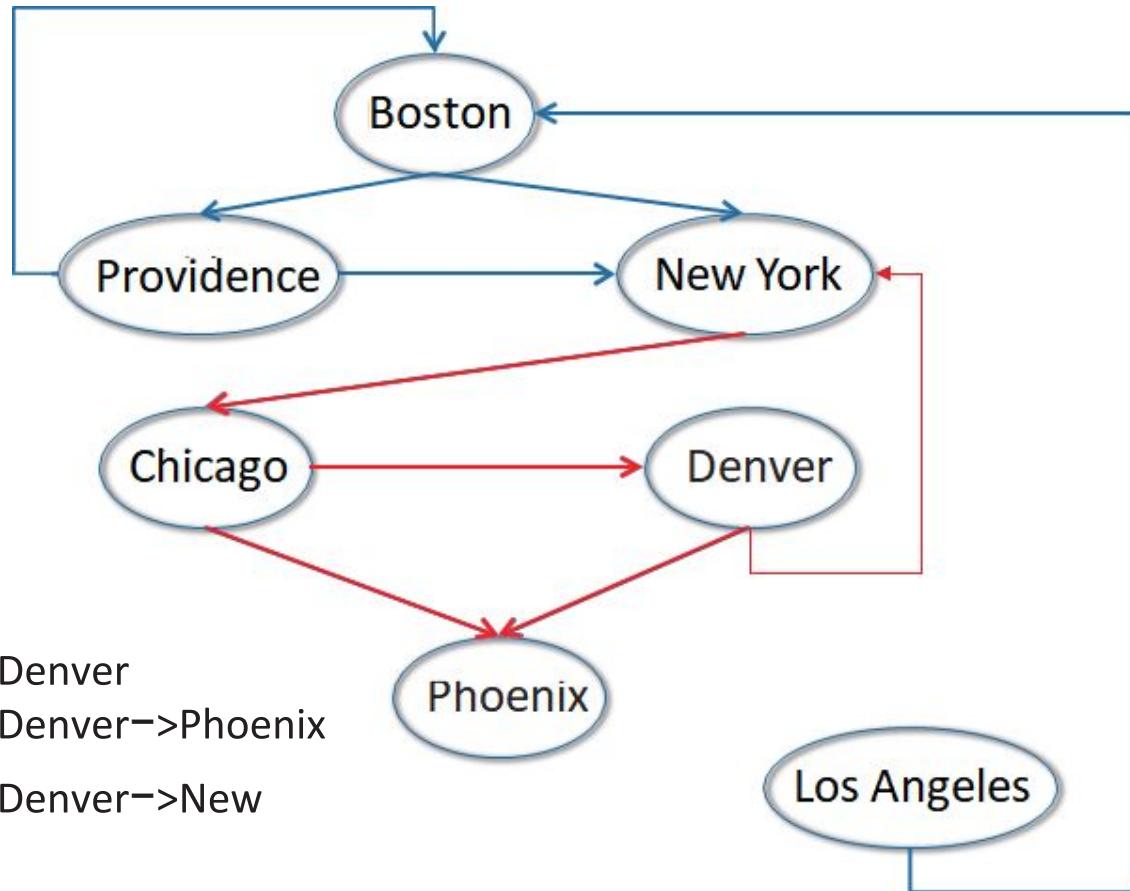
Denver: Phoenix, New York

Los Angeles: Boston

Phoenix:



출력 (Chicago에서 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

출력 (Boston에서 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->Providence->New York->Chicago->Phoenix **Found a shorter path**

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

Current DFS path: Boston->New York->Chicago->Phoenix **Found a shorter path**

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

너비 우선 탐색

- 
- 시작점에서 시작
 - 특정한 순서로 시작점에서 나가는 모든 변을 고려
 - 첫 번째 변을 따라가고, 도착점인지 확인
 - 아니라면, 현재 꼭짓점에서 다음 변을 시행
 - 도착점을 찾거나, 선택지가 없어질 때까지 반복
 - 변의 선택지가 없어지면, 시작점에서의 거리가 같은 다음 꼭짓점으로 가서 과정을 반복
 - 꼭짓점의 선택지가 없어지면, 그래프의 다음 레벨로 가서(시작점에서 한 단계 이동한 모든 꼭짓점) 반복

알고리즘 2: 너비 우선 탐색 (BFS)

```
def BFS(graph, start, end, toPrint = False):    initPath = [start]
    pathQueue = [initPath]        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
```

?

n 루프 이상의 경로를 탐색하기 전에
n 루프의 모든 경로를 탐색

출력 (Boston에서 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

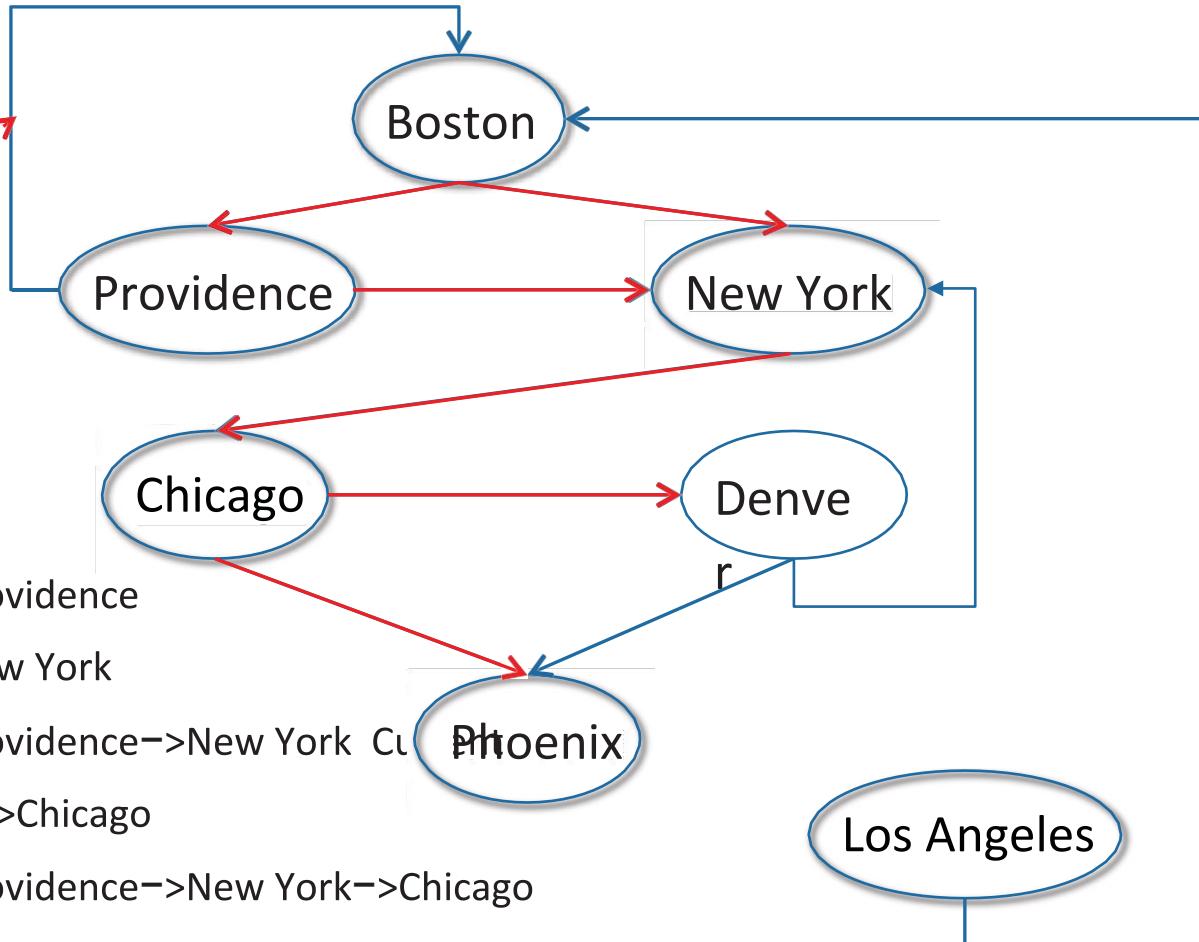
Current BFS path: Boston->Providence->New York->Chicago Current BFS path:
Boston->New York->Chicago->Denver

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

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

출력 (Boston에서 Pheonix)

꼭짓점을
재방문하는 경로는
넘어가는 걸
알아두세요



Current BFS path: Boston

Current BFS path: Boston->Providence

Current BFS path: Boston->New York

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

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

최소 가중치 경로는 어떨까

- 변의 수가 아닌, 변의 가중치의 합을 최소화하려 함
- DFS 는 이를 위해 쉽게 수정 가능
- BFS 는 불가능, 왜냐하면 최소 가중치 경로는 최소 루프 수보다 더 많은 루프를 가질 수 있기 때문

복습

- 그래프는 멋짐
 - 많은 것들의 모델을 만드는데 좋은 방법
 - 물건들 간의 관계를 담고 있음
 - 많은 중요한 문제들은 우리가 이미 해를 구하는 방법을 알고 있는 그래프 최적화 문제로 나타낼 수 있음
- 깊이 우선 탐색과 너비 우선 탐색은 중요한 알고리즘
 - 많은 문제를 푸는데 쓰일 수 있음

[MIT OpenCourseWare](#)

<https://ocw.mit.edu>

6.0002 Introduction to Computational Thinking and Data Science

Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.