파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

그래프의 너비우선탐색(Breath-First Search)

알고리즘을 학습하고 구현한다

Data Structures in Python
Chapter 9

- Graph Introduction
- **Graph Traversal – BFS**
- Graph Traversal – DFS
- Topological Sort of DAG

Agenda

- Graph Traversals
  - **BFS - Breadth First Search**
  - DFS - Depth First Search

- Reference:
  - Problem Solving with Algorithms and Data Structures
  - Wikipedia: Breadth-first search

Graph Traversals

- Important graph-processing operations include:
  - Finding the shortest path to a given vertex (source) in a graph
  - Finding all the items to which a given item is connected by paths

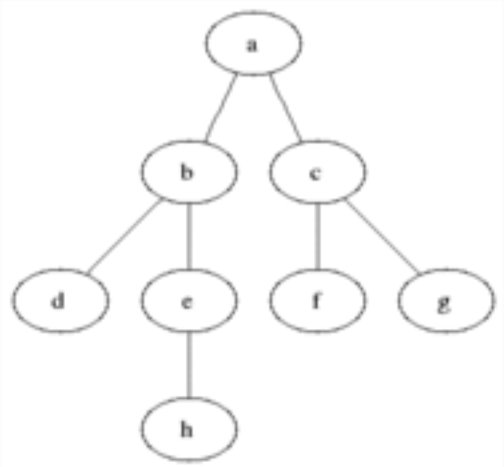
- **Breadth-First Search (BFS)**
  - **Idea:** Explore from a source in all possible directions, **layer by layer.**
  - It begins at the source vertex and explores its **neighbors first**.
  - Then, it explores their unexplored next neighbors, until it visits the target vertex or all.

- **Depth-First Search (DFS)**
  - **Idea:** Follow the first path you find as far as you can go.
  - Then, back up to last unexplored edge when you reach a dead end, then go as far you can.

BFS Algorithm

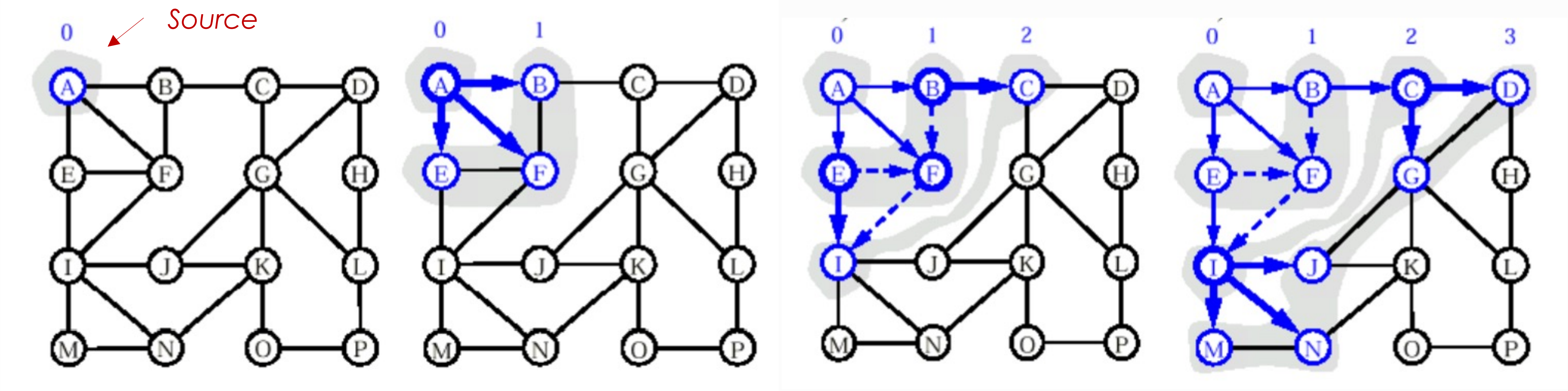
- It takes the current vertex (the source vertex in the beginning) and then add all its neighbors that we have not visited yet to a **queue**.
- Continue this with the next vertex in the queue (the "oldest" vertex).
- Set its distance to the distance of **the current vertex plus 1** (since all edges are weighted equally), with the distance to the source vertex being 0.
- This is repeated until there are no more vertices in the queue (all vertices are visited).

- It always **finds the shortest path** if there is more than one path between two vertices.

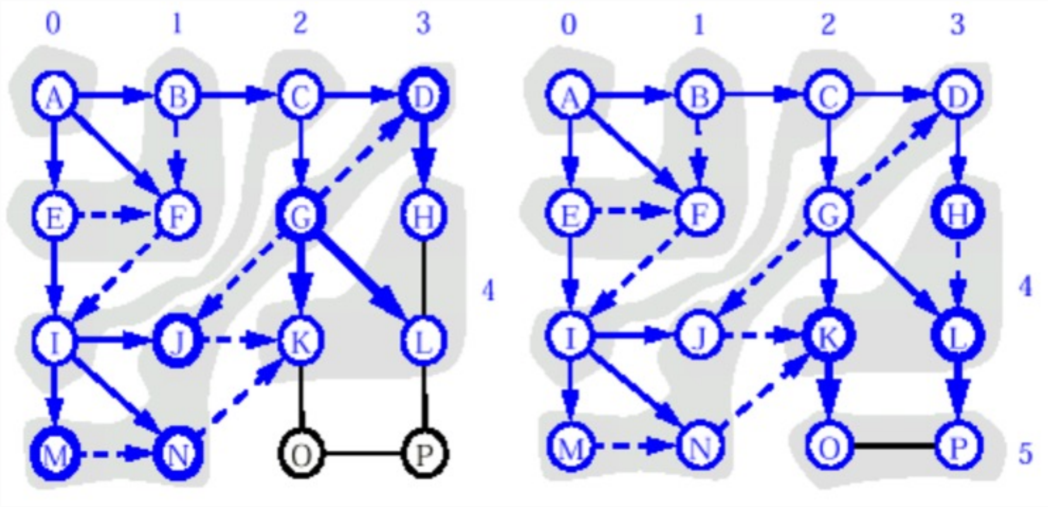


BFS - A Graphical Representation

queue: A *queue: B E F* *queue: C I* *queue: D G J M N*



queue: H K L *queue: O P*

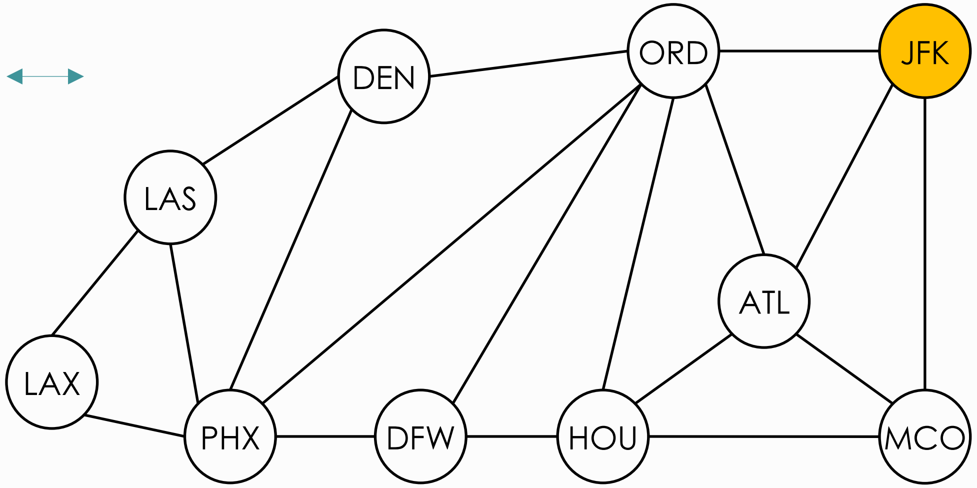


The figure adapted from Goodrich & Tamassia

Example Graph Representation:

routes.txt
edge list

JFK MCO
ORD DEN
ORD HOU
DFW PHX
JFK ATL
JFK ORD
ORD DFW
ORD PHX
ATL HOU
DEN PHX
ATL ORD
ATL MCO
DEN LAS
HOU MCO
PHX LAS
LAX PHX
LAX LAS
HOU DFW




```
def addEdge(self, v, w):
    if not self.hasVertex(v): self._adj[v] = list()
    if not self.hasVertex(w): self._adj[w] = list()
        if not self.hasEdge(v, w):
            self._e += 1
            self._adj[v].append(w)
            self._adj[w].append(v)
```

Getting each neighbor w of v

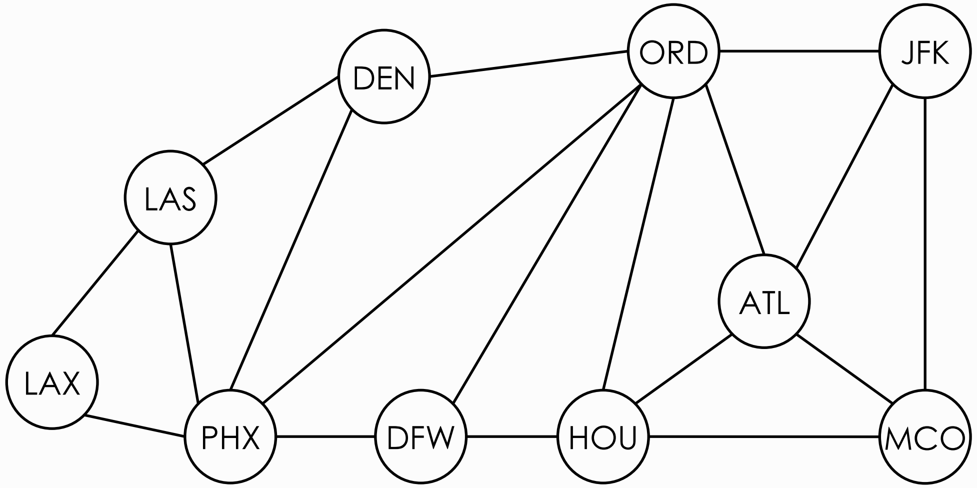
```
for w in g.neighbors(v):
```

Adjacency list

JFK: MCO ATL ORD
MCO: JFK ATL HOU
ORD: DEN HOU JFK DFW PHX ATL
DEN: ORD PHX LAS
HOU: ORD ATL MCO DFW
DFW: PHX ORD HOU
PHX: DFW ORD DEN LAS LAX
ATL: JFK HOU ORD MCO
LAS: DEN PHX LAX
LAX: PHX LAS

V W

Example Graph ADT:



operation	description
`bfs = BFS(g, s)`	`find all shortest paths s in graph g`
`bfs.distanceTo(v)`	`distance between s and v`
`bfs.hasPathTo(v)`	`is there a path between s and v?`
`bfs.pathTo(v)`	`an iterable for the path from s to v`

Examples of shortest paths in a graph

source	target	distance	***a*** shortest path
JFK	LAX	3	JFK-ORD-PHX-LAX
LAS	MCO	4	LAS-DEN-ORD-HOU-MCO
HOU	JFK	2	HOU-ORD-JFK

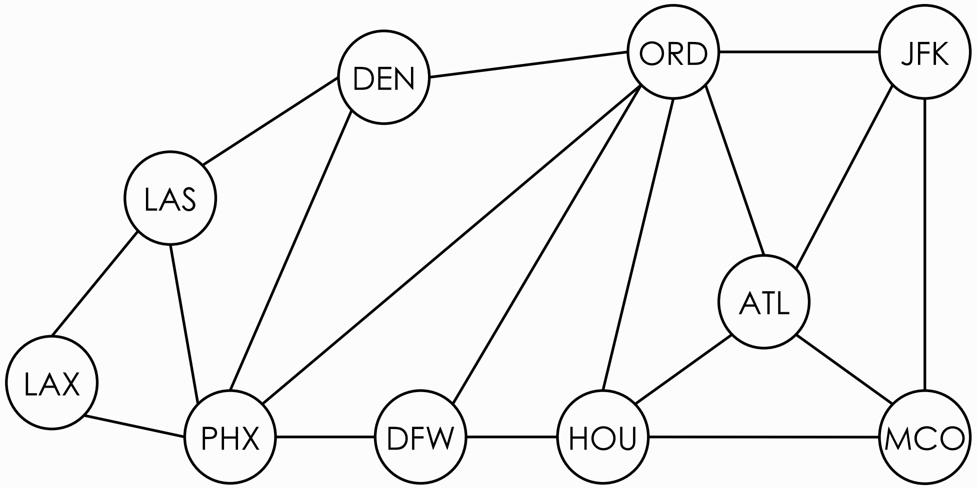
BFS Algorithm

1. Initialization
   1. Initialize the distance to the source vertex `s` as 0.
   2. Initialize `_distTo` dictionary that stores the distance to `s`.
      If `v` exists in `_distTo`, it indicates "visited", if `not in _distTo`, "unvisited".
      Initialize `_prevTo` dictionary that stores the vertex that one step nearer to the source `s`.
   3. Add the first vertex `s` to the queue.
2. While there are vertices in the queue:
   1. Take a vertex `v` out of the queue
   2. For all vertices `w` next to it `v` that we have not visited yet,
      add them `w` to the queue,
      set their distance `_distTo[w]` to the distance to the current vertex `distTo[v]` plus `1`
      set their `_prevTo[w]` to the current vertex `v` which one step nearer than them `w`

```
instance variables:
_distTo   dict type, distance to s
_prevTo   dict type, previous vertex on
                     shortest path from s
```

```
variables:
queue   queue of vertices to visit
g       graph
s       source
v       current vertex
w       neighbors of v
```

BFS Class



Adjacency list: ***g._adj***

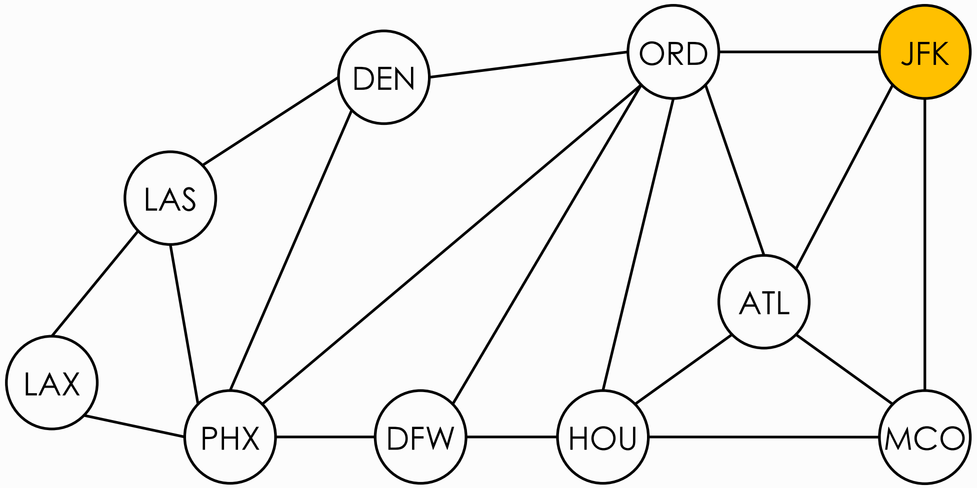
```
JFK: MCO ATL ORD
MCO: JFK ATL HOU
ORD: DEN HOU JFK DFW PHX ATL
DEN: ORD PHX LAS
HOU: ORD ATL MCO DFW
DFW: PHX ORD HOU
PHX: DFW ORD DEN LAS LAX
ATL: JFK HOU ORD MCO
LAS: DEN PHX LAX
LAX: PHX LAS
```

```
class BFS:
    def __init__(self, graph, s):
        self._distTo = dict()
        self._prevTo = dict()
        self._distTo[s] = 0
        self._prevTo[s] = None
        self._path = []
        queue = deque()
        queue.append(s)

        while queue:
            v = queue.popleft()
            for w in g.neighbors(v):
                if w not in self._distTo:
                    queue.append(w)
                    self._distTo[w] = 1 + self._distTo[v]
                    self._prevTo[w] = v
```

variables:

```
_distTo   distance to s
_prevTo   previous vertex on shortest path from s
queue     queue of vertices to visit
g         graph
s         source
v         current vertex
w         neighbors of v
```

BFS Class Example: JFK



distance 0

```
s = 'JFK'
queue = ['JFK']
_distTo = {'JFK':0}
_prevTo = {None}
```

distance 1

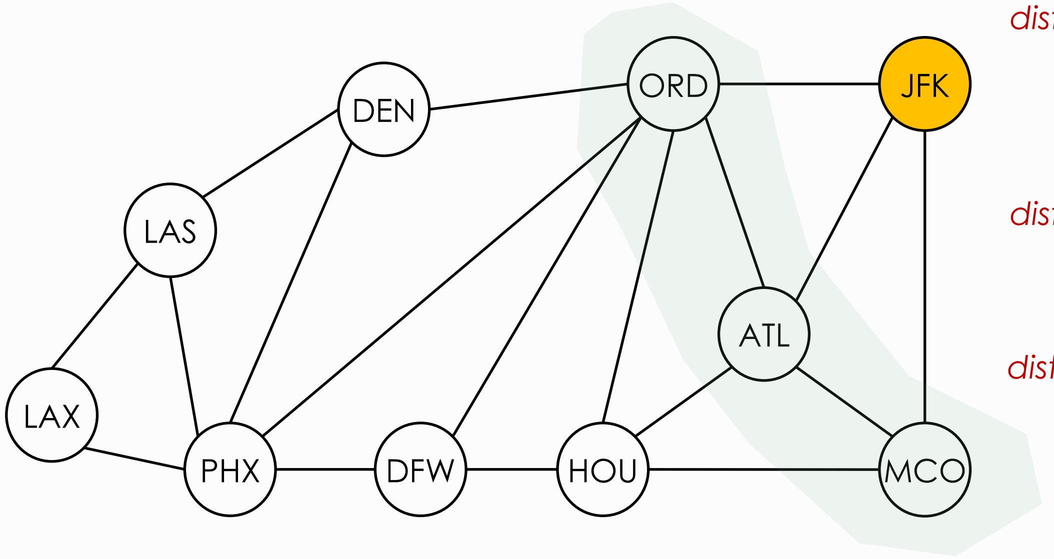
distance 2

distance 3

Adjacency list: ***g._adj***

```
JFK: MCO ATL ORD
MCO: JFK ATL HOU
ORD: DEN HOU JFK DFW PHX ATL
DEN: ORD PHX LAS
HOU: ORD ATL MCO DFW
DFW: PHX ORD HOU
PHX: DFW ORD DEN LAS LAX
ATL: JFK HOU ORD MCO
LAS: DEN PHX LAX
LAX: PHX LAS
```

BFS Class Example: JFK



distance 0

```
s = 'JFK'
queue = ['JFK']
_distTo = {'JFK':0}
_prevTo = {None}
```

Is the order of queue in random or fixed?

distance 1

```
queue = ['MCO', 'ATL', 'ORD']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK'}
```

Is this order of keys in random or fixed?

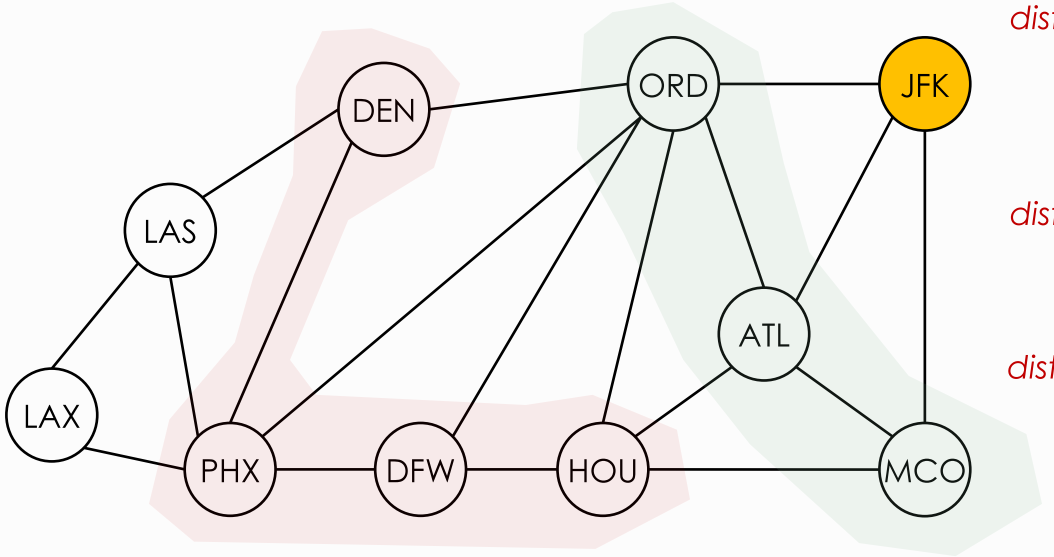
distance 2

distance 3

Adjacency list: ***g._adj***

```
JFK: MCO ATL ORD
MCO: JFK ATL HOU
ORD: DEN HOU JFK DFW PHX ATL
DEN: ORD PHX LAS
HOU: ORD ATL MCO DFW
DFW: PHX ORD HOU
PHX: DFW ORD DEN LAS LAX
ATL: JFK HOU ORD MCO
LAS: DEN PHX LAX
LAX: PHX LAS
```

BFS Class Example: JFK



distance 0

```
s = 'JFK'
queue = ['JFK']
_distTo = {'JFK':0}
_prevTo = {None}
```

Is the order of queue in random or fixed?

distance 1

```
queue = ['MCO', 'ATL', 'ORD']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK'}
```

Is this order of keys in random or fixed?

distance 2

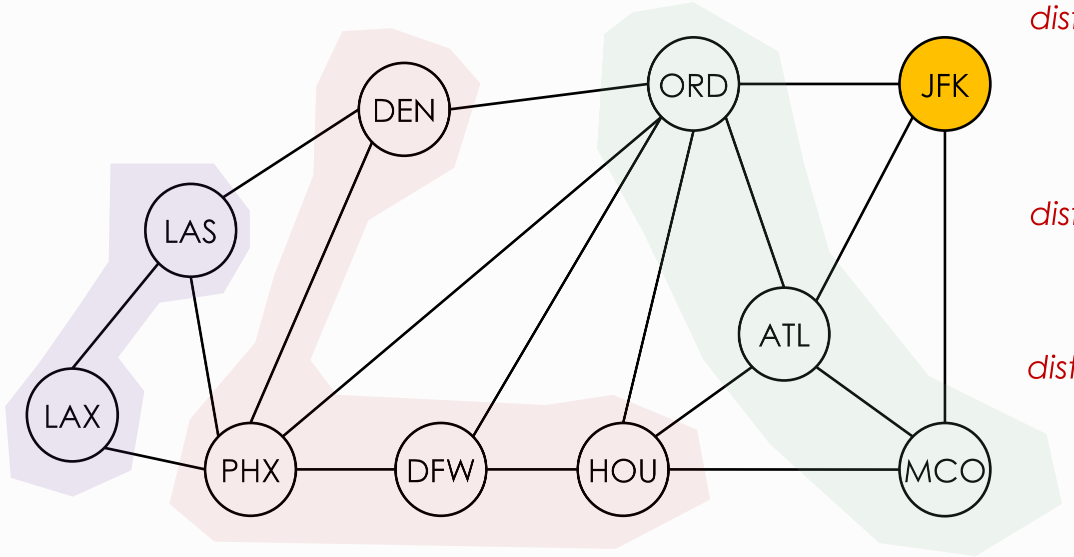
```
queue = ['HOU', 'DEN', 'DFW', 'PHX']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1,
           'HOU':2, 'DEN':2, 'DFW':2, 'PHX':2}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
           'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD', 'PHX':'ORD'}
```

distance 3

Adjacency list: ***g._adj***

```
JFK: MCO ATL ORD
MCO: JFK ATL HOU
ORD: DEN HOU JFK DFW PHX ATL
DEN: ORD PHX LAS
HOU: ORD ATL MCO DFW
DFW: PHX ORD HOU
PHX: DFW ORD DEN LAS LAX
ATL: JFK HOU ORD MCO
LAS: DEN PHX LAX
LAX: PHX LAS
```

BFS Class Example: JFK



Adjacency list: ***g._adj***

```
JFK: MCO ATL ORD
MCO: JFK ATL HOU
ORD: DEN HOU JFK DFW PHX ATL
DEN: ORD PHX LAS
HOU: ORD ATL MCO DFW
DFW: PHX ORD HOU
PHX: DFW ORD DEN LAS LAX
ATL: JFK HOU ORD MCO
LAS: DEN PHX LAX
LAX: PHX LAS
```

distance 0

```
s = 'JFK'
queue = ['JFK']
_distTo = {'JFK':0}
_prevTo = {None}
```

Is the order of queue in random or fixed?

distance 1

```
queue = ['MCO', 'ATL', 'ORD']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK'}
```

distance 2

```
queue = ['HOU', 'DEN', 'DFW', 'PHX']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1,
           'HOU':2, 'DEN':2, 'DFW':2, 'PHX':2}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
           'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD', 'PHX':'ORD'}
```

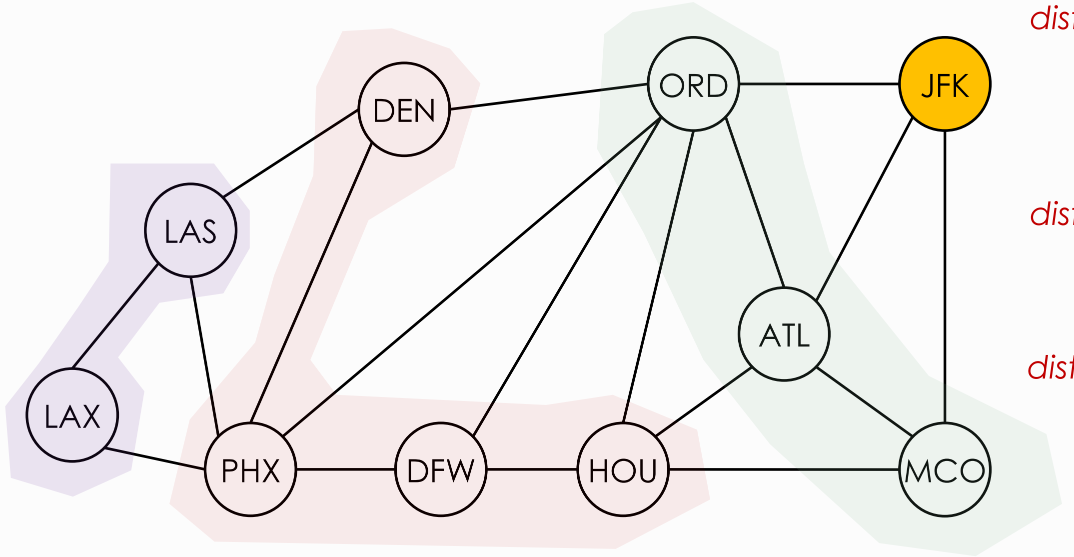
distance 3

```
queue = ['LAS', 'LAX']
_distTo = {'JFK':0, 'ORD':1, 'ATL':1, 'MCO':1,
           'HOU':2, 'DEN':2, 'DFW':2, 'PHX':2,
           'LAS':3, 'LAX':3}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
           'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD', 'PHX':'ORD',
           'LAS':'DEN', 'LAX':'PHX'}
```

can it be like the following?
'LAS':'PHX'

Question: *Explain how it gets the order of airports in the queue [] at distance 2.*

BFS Class Example: JFK



Adjacency list: ***g._adj***

```
JFK: MCO ATL ORD
MCO: JFK ATL HOU
ORD: DEN HOU JFK DFW PHX ATL
DEN: ORD PHX LAS
HOU: ORD ATL MCO DFW
DFW: PHX ORD HOU
PHX: DFW ORD DEN LAS LAX
ATL: JFK HOU ORD MCO
LAS: DEN PHX LAX
LAX: PHX LAS
```

distance 0

```
s = 'JFK'
queue = ['JFK']
_distTo = {'JFK':0}
_prevTo = {None}
```

distance 1

```
queue = ['MCO', 'ATL', 'ORD']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK'}
```

distance 2

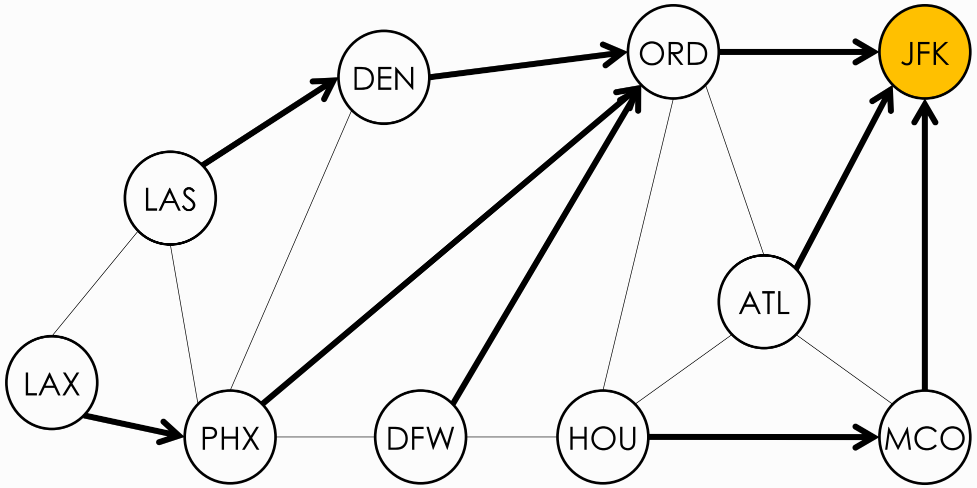
```
queue = ['HOU', 'DEN', 'DFW', 'PHX']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1,
           'HOU':2, 'DEN':2, 'DFW':2, 'PHX':2}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
           'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD', 'PHX':'ORD'}
```

distance 3

```
queue = ['LAS', 'LAX']
_distTo = {'JFK':0, 'ORD':1, 'ATL':1, 'MCO':1,
           'HOU':2, 'DEN':2, 'DFW':2, 'PHX':2}
           'LAS':3, 'LAX':3}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
           'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD', 'PHX':'ORD',
           'LAS':'DEN', 'LAX':'PHX'}
```

- Find the distance to LAX: `g._distTo['LAX']` → `3`
- Find the shortest path to LAX: `g._prevTo['LAX']` → `'PHX'`
  `g._prevTo['PHX']` → `'ORD'`
  `g._prevTo['ORD']` → `'JFK'`
  `g._prevTo['JFK']` → `None`

BFS Class Example: JFK shortest paths tree



shortest paths tree

distance 0

```
s = 'JFK'
queue = ['JFK']
_distTo = {'JFK':0}
_prevTo = {None}
```

distance 1

```
queue = ['MCO', 'ATL', 'ORD']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK'}
```

distance 2

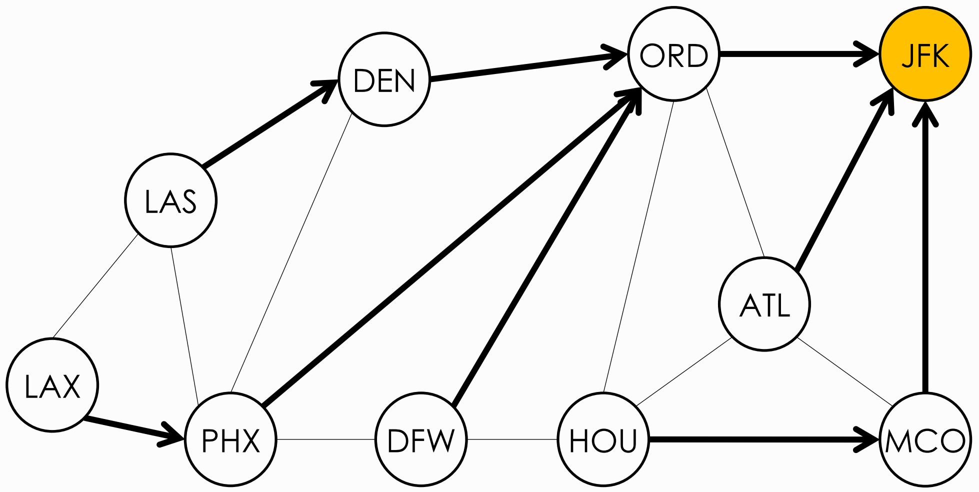
```
queue = ['HOU', 'DEN', 'DFW', 'PHX']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1,
           'HOU':2, 'DEN':2, 'DFW':2, 'PHX':2}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
           'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD', 'PHX':'ORD'}
```

distance 3

```
queue = ['LAS', 'LAX']
_distTo = {'JFK':0, 'ORD':1, 'ATL':1, 'MCO':1,
           'HOU':2, 'DEN':2, 'DFW':2, 'PHX':2}
           'LAS':3, 'LAX':3}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
           'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD', 'PHX':'ORD',
           'LAS':'DEN', 'LAX':'PHX'}
```

- `Find the distance to LAX:      g._distTo['LAX'] → 3`
- `Find the shortest path to LAX: g._prevTo['LAX'] → 'PHX'`
  `g._prevTo['PHX'] → 'ORD'`
  `g._prevTo['ORD'] → 'JFK'`
  `g._prevTo['JFK'] → None`

BFS Class Example: JFK shortest paths tree



distance 0

```
s = 'JFK'
queue = ['JFK']
_distTo = {'JFK':0}
_prevTo = {None}
```

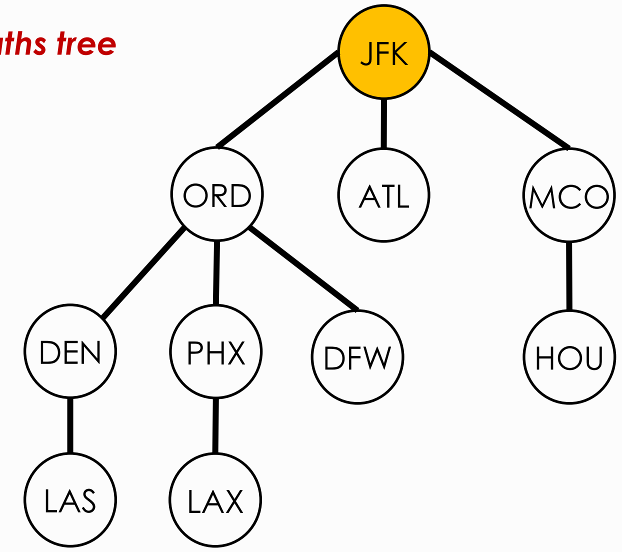
distance 1

```
queue = ['MCO', 'ATL', 'ORD']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK'}
```

distance 2

```
queue = ['HOU', 'DEN', 'DFW', 'PHX']
_distTo = {'JFK':0, 'MCO':1, 'ATL':1, 'ORD':1,
           'HOU':2, 'DEN':2, 'DFW':2, 'PHX':2}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
           'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD', 'PHX':'ORD'}
```

shortest paths tree

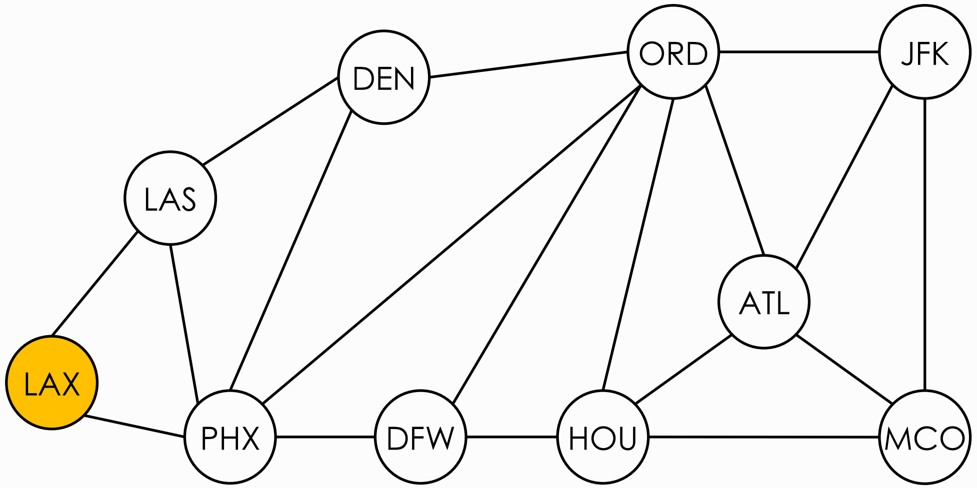


distance 3

```
queue = ['LAS', 'LAX']
_distTo = {'JFK':0, 'ORD':1, 'ATL':1, 'MCO':1,
           'HOU':2, 'DEN':2, 'DFW':2, 'PHX':2}
           'LAS':3, 'LAX':3}
_prevTo = {'JFK':None, 'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
           'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD', 'PHX':'ORD',
           'LAS':'DEN', 'LAX':'PHX'}
```

- Find the distance to LAX: g._distTo['LAX'] → 3
- Find the shortest path to LAX: g._prevTo['LAX'] → 'PHX'
  g._prevTo['PHX'] → 'ORD'
  g._prevTo['ORD'] → 'JFK'
  g._prevTo['JFK'] → None

BFS Class Exercise: LAX



distance 0

```
s = 'LAX'
queue = ['LAX']
_distTo = {'LAX':0}
_prevTo = {None}
```

distance 1

```
queue = ['LAS', 'PHX']
_distTo =
_prevTo =
```

distance 2

distance 3

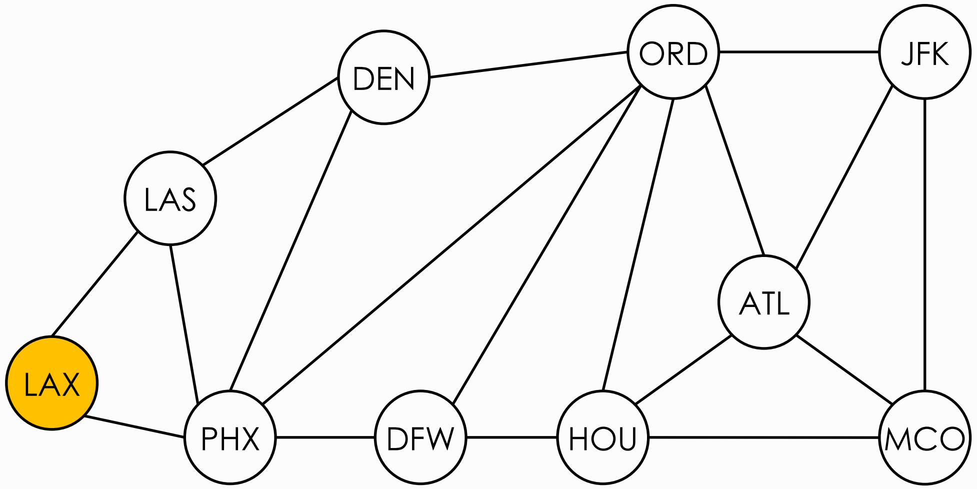
distance 4

Adjacency list: ***g._adj***

```
JFK: MCO ATL ORD
MCO: JFK ATL HOU
ORD: DEN HOU JFK DFW PHX ATL
DEN: ORD PHX LAS
HOU: ORD ATL MCO DFW
DFW: PHX ORD HOU
PHX: DFW ORD DEN LAS LAX
ATL: JFK HOU ORD MCO
LAS: DEN PHX LAX
LAX: PHX LAS
```

- Find the distance to MCO: g._distTo['MCO'] →
- Find the shortest path to MCO: g._prevTo['MCO'] →

BFS Class Exercise: LAX shortest paths tree



shortest paths tree

distance 0

```
s = 'LAX'
queue = ['LAX']
_distTo = {'LAX':0}
_prevTo = {None}
```

distance 1

```
queue = ['LAS', 'PHX']
_distTo =
_prevTo =
```

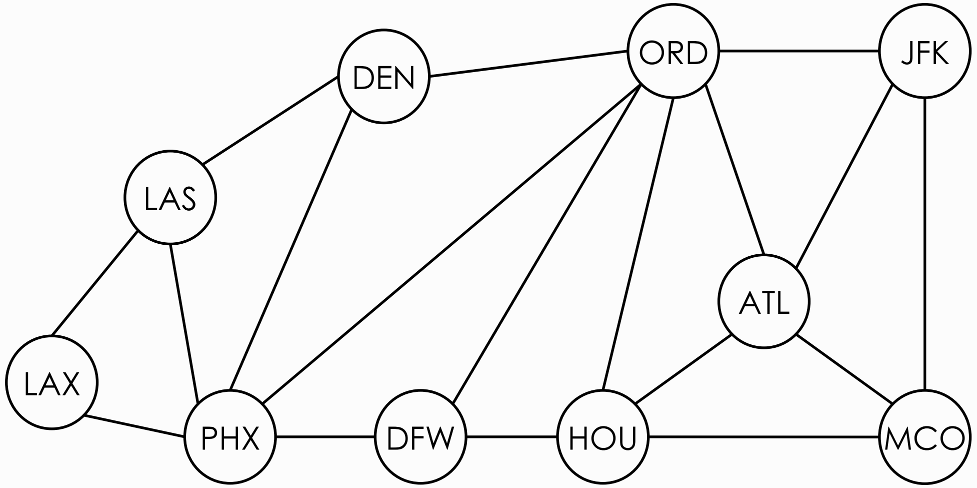
distance 2

distance 3

distance 4

- Find the distance to MCO: `g._distTo['MCO']` →
- Find the shortest path to MCO: `g._prevTo['MCO']` →

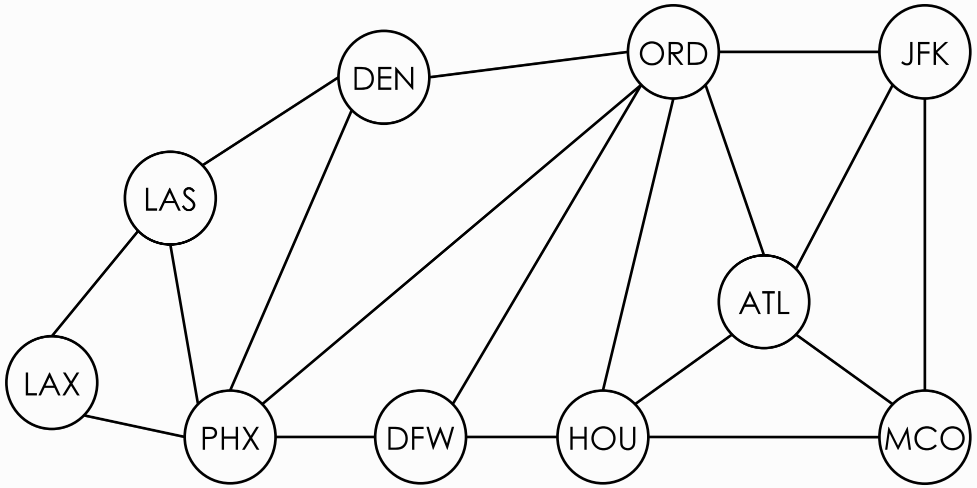
BFS Class



operation	*description*
`bfs = BFS(g, s)`	`find all shortest paths s in graph g`
`bfs.distanceTo(v)`	`distance between s and v`
`bfs.hasPathTo(v)`	`is there a path between s and v?`
`bfs.pathTo(v)`	`an iterable for the path from s to v`

```
from collections import deque
from graph import Graph

class BFS:
    def __init__(self, graph, s):
        ...

    def distanceTo(self, v):
        return self._distTo[v]

    def hasPathTo(self, v):
        return v in self._distTo

    def pathTo(self, v):
        path = []
        while v is not None:
            path += [v]
            v = self._prevTo[v]
        return reversed(path)
```

BFS Class Exercise



operation	*description*
`bfs = BFS(g, s)`	find all shortest paths s in graph g
`bfs.distanceTo(v)`	distance between s and v
`bfs.hasPathTo(v)`	is there a path between s and v?
`bfs.pathTo(v)`	an iterable for the path from s to v
`bfs.source()`	return the source vertex
`bfs.shortestPaths()`	print all shortest paths from source

```
from collections import deque
from graph import Graph

class BFS:
    def __init__(self, graph, s):
        ...
```

```
Shortest Paths from: JFK
['JFK']
['JFK', 'MCO']
['JFK', 'ATL']
['JFK', 'ORD']
['JFK', 'MCO', 'HOU']
['JFK', 'ORD', 'DEN']
['JFK', 'ORD', 'DFW']
['JFK', 'ORD', 'PHX']
['JFK', 'ORD', 'DEN', 'LAS']
['JFK', 'ORD', 'PHX', 'LAX']
```

Do it for LAX

```
    def source(self):
        pass

    def shortestPaths(self):
        pass
```

Time Complexity

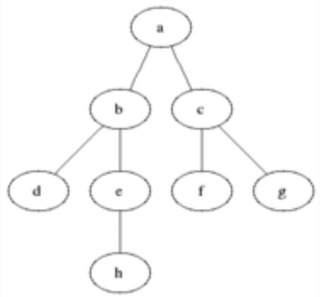
- We can obtain the time complexity by counting the number to visit the vertices.
  - The repeats in the while-loop become V in the worst case. The repeats will be the sum of the vertices at most because we only add the vertices not visited.
  - The repeats in the for-loop become $2E$ in the worst case. The repeats in the only one for-loop will be the degree. So, the repeats in all for-loop will be the sum of the degrees or $2E$.
  - Therefore, we get **the time complexity of the breadth-first search as $\boldsymbol{O(V + E)}$**.

  - **Handshaking lemma**:
    The sum of all the degrees always doubles as the sum of all edges for **undirected graph**.

$$\sum_{v \in V} deg(v) = 2|E|$$

Summary

- BFS(Breadth-First Search) traverses by adding any each one of the graph's vertices at the back of a queue, starting from the source vertex.
- BFS always **finds the shortest path** if there is more than one path between two vertices.
- The time complexity of BFS is linear, $O(V + E)$.



학습 정리

1) 그래프의 너비우선탐색(Breadth-First Search)은

Step 1: 탐색 시작 노드 v를 큐에 삽입하고 방문 처리를 한다
Step 2: 큐에서 노드 v를 꺼내 그 노드의 인접 노드 중에서
방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리한다
Step 3: Step 2의 과정을 더 이상 수행할 수 없을 때까지 반복한다

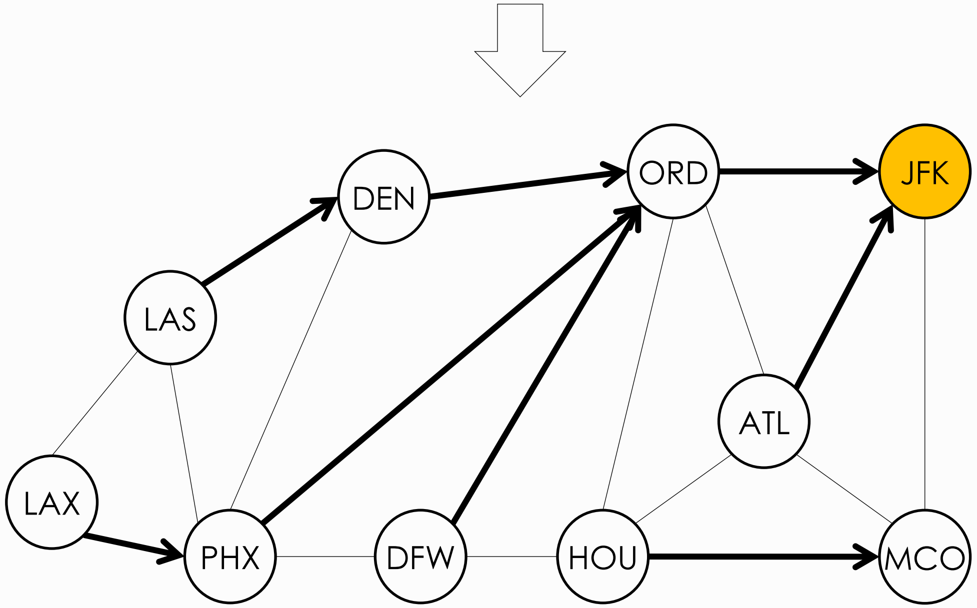
2) BFS로 최단 거리 경로를 찾을 수 있다

3) BFS의 시간 복잡도는 O(V + E)이다


파이썬으로 배우는
데이터 구조
수고했습니다
곧 다음 시간에
다시 뵙겠습니
다

BFS Class Example: JFK shortest paths tree

```
_prevTo = {
'JFK':None,
'MCO':'JFK', 'ATL':'JFK', 'ORD':'JFK',
'HOU':'MCO', 'DEN':'ORD', 'DFW':'ORD',
'PHX':'ORD', 'LAS':'DEN', 'LAX':'PHX'}
```



shortest paths tree