파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

그래프의 기본적인 구조와

ADT 및 표현 방법들을 학습한다

Data Structures in Python
Chapter 9

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

Agenda

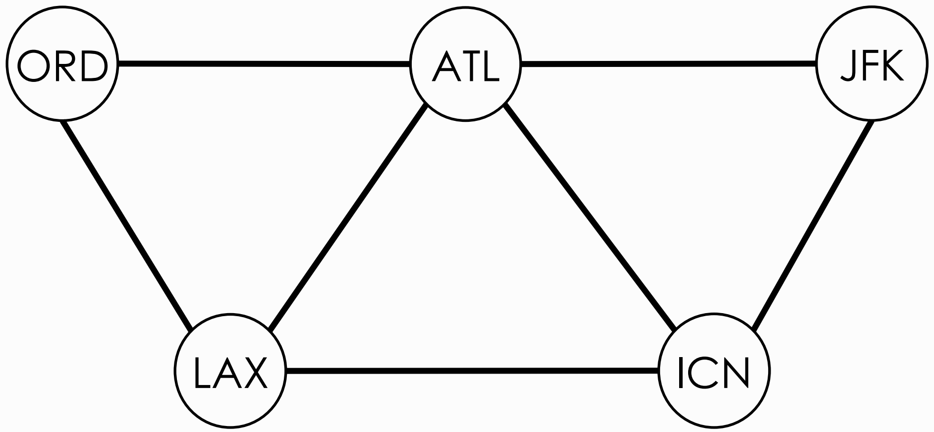
- Graph Introduction
  - Graph Definitions
  - Graph Representations
  - Graph ADT and Coding

- Reference:
  - Problem Solving with Algorithms and Data Structures
  - Wikipedia: Graph (Abstract Data Type)

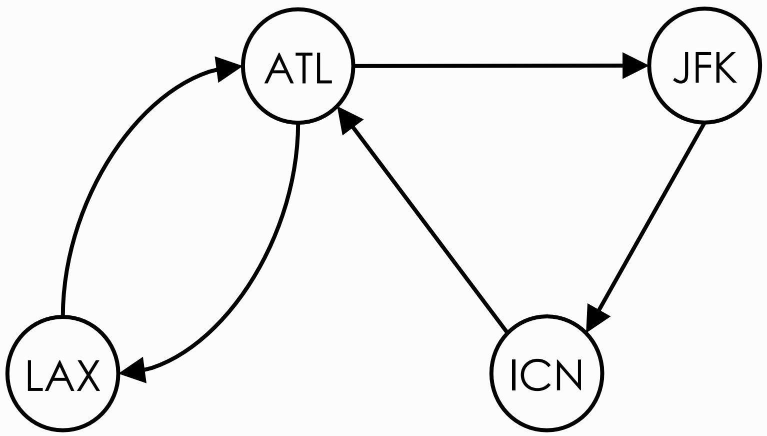
Graph Definitions

- A **graph** is composed of a set of **vertices** and a set of **edges.**
  - Graphs can be **undirected** or **directed**.
- Each **edge** represents a connection between two vertices.
- One vertex is **adjacent** to another vertex if there is an edge connection the two vertices. Then, two vertices are **neighbors.** The **degree** of a vertex is its number of neighbors.

Undirected Graph



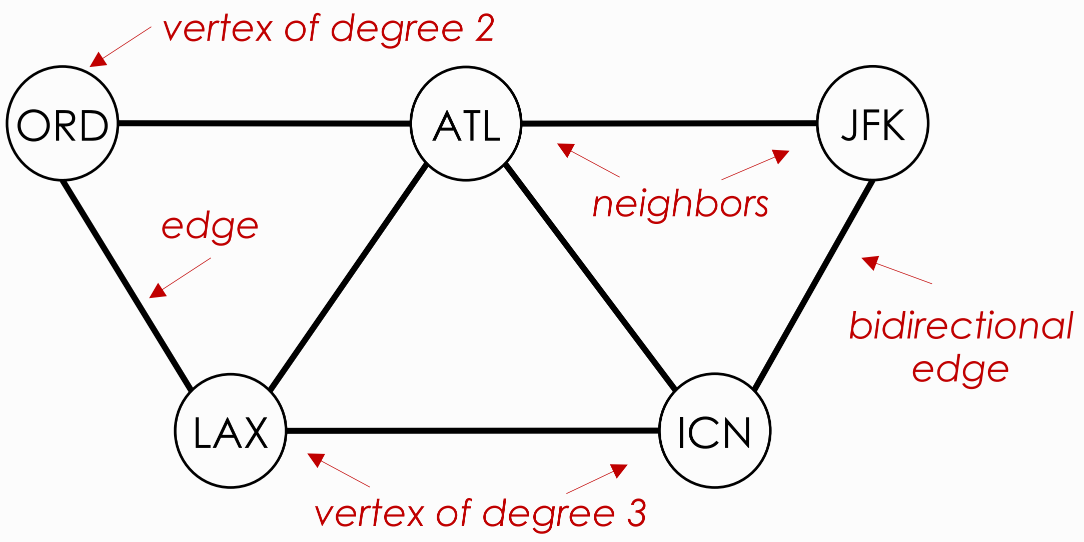
Digraph with a cycle



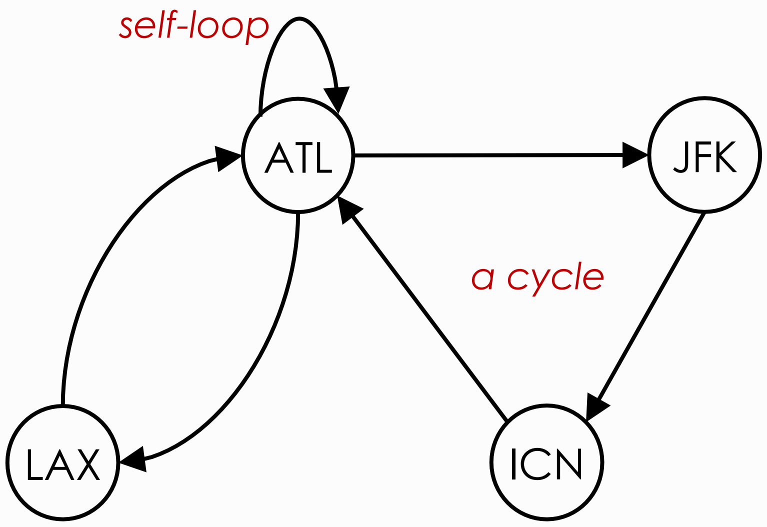
Graph Definitions

- A **graph** is composed of a set of **vertices** and a set of **edges**.
  - Graphs can be **undirected** or **directed**.
- Each **edge** represents a connection between two vertices.
- One vertex is **adjacent** to another vertex if there is an edge connection the two vertices. Then, two vertices are **neighbors.** The **degree** of a vertex is its number of neighbors.

Undirected Graph



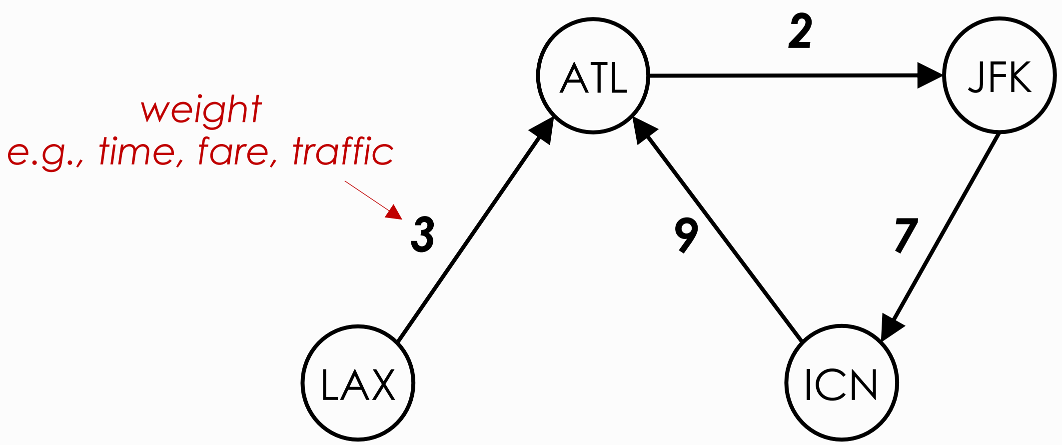
Digraph with a cycle



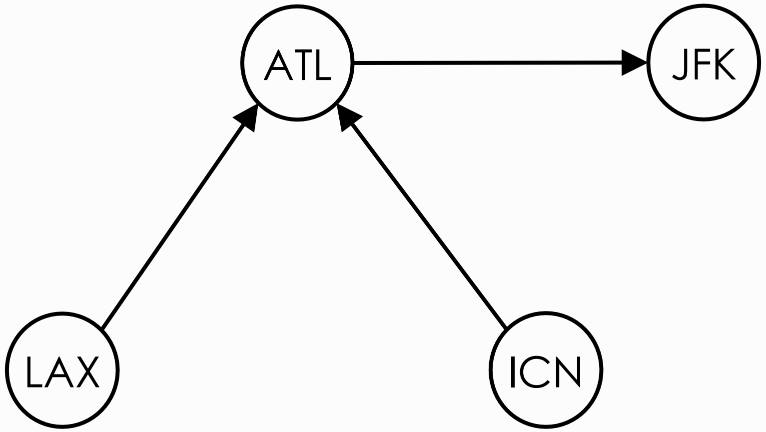
Graph Definitions

- Edges can be directed/undirected and/or have **weights**.
  - Lists and trees are special cases of directed graphs
- A **path** in a graph is a sequence of vertices connected by edges.
- A **cycle** is a path that begins and ends the same vertex
- A special case of **digraph** that contains no cycles is known as a **directed acyclic graph**, **DAG**.

Weighted graph

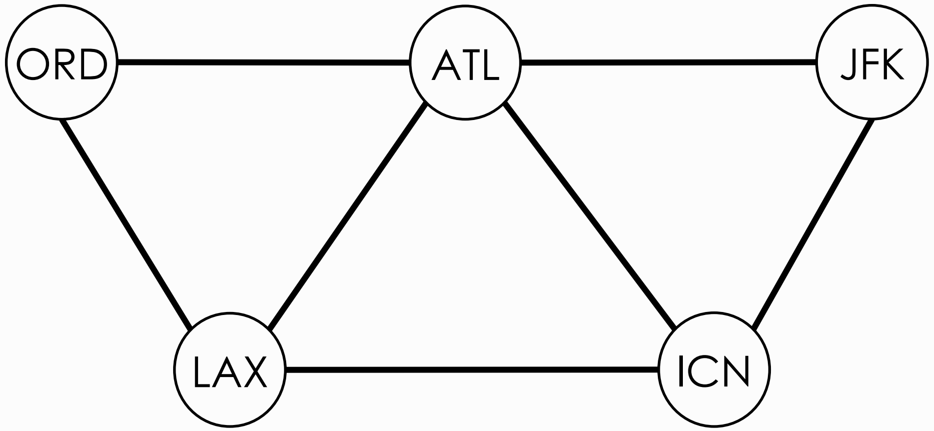


DAG: Directed acyclic graph



Graph Representations

- There are a few ways to represent graphs, each with its advantages and disadvantages. Here, we will see three ways; an **edge list**, **adjacency matrix** or **adjacency list** data structure.
- **Edge lists**
  - It is a list, or array, of edges $|E|$ in a graph. If edges have weights, add a third element to the list.
  - Edge lists are simple, but if we want to find whether the graph contains a particular edge, we must search through the edge list, $O(E)$, E is the number of edges in a graph.
  - It is commonly used to save the graph in a text format.



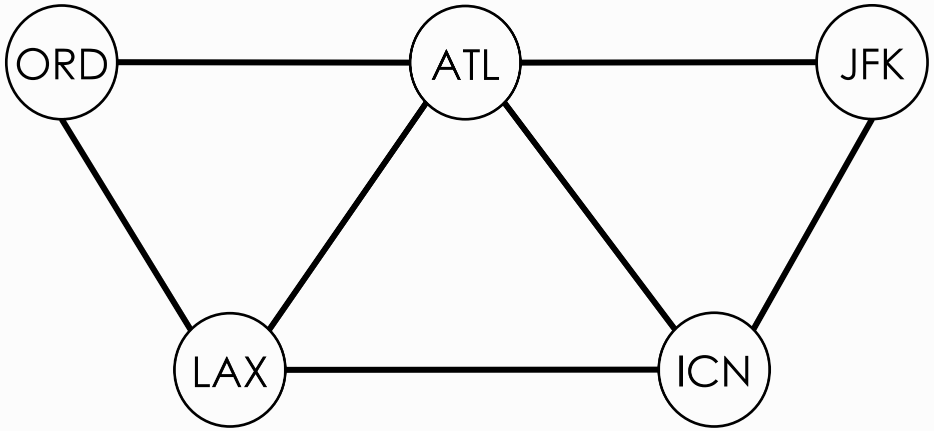
edge list

```
ATL ICN
ATL JFK
ICN JFK
LAX ICN
ATL LAX
ORD ATL
ORD LAX
```

Use to save a graph in a file, route5.txt

Graph Representations

- There are a few ways to represent graphs, each with its advantages and disadvantages. Here, we will see three ways; an **edge list**, **adjacency matrix** or **adjacency list** data structure.
- **Adjacency Matrices**
  - For a graph with $|V|$, an adjacency matrix is a $|V| \times |V|$ matrix of 0s and 1s, where the entry in row i and column j is 1 if and only if the edge (i, j) is in the graph.
  - It takes a constant time $O(1)$ to find out whether an edge is present in a graph.
  - It takes a space complexity of $O(V^2)$, even if the graph is **sparse** (or relatively few edges).

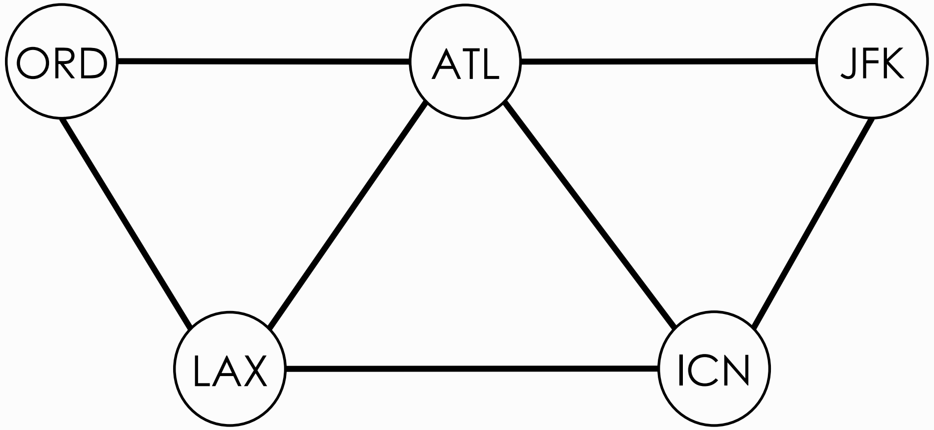


Adjacency matrix

	ATL	ICN	JFK	LAX	ORD
ATL	0	1	1	1	1
ICN	1	0	1	1	0
JFK	1	1	0	0	0
LAX	1	1	0	0	1
ORD	1	0	0	1	0

Graph Representations

- There are a few ways to represent graphs, each with its advantages and disadvantages. Here, we will see three ways; an **edge list**, **adjacency matrix** or **adjacency list** data structure.
- **Adjacency Lists**
  - It combines adjacency matrices with edge lists. For each vertex, store a list of the vertices adjacent to it. We typically have an array of $|V|$ adjacency lists, one adjacency list per vertex.
  - It takes a constant time to access a vertex's adjacency list, because we just index through a list.
  - For a directed graph, the adjacency lists contain a total of $|E|$ elements, one element per directed edge and $2 \times |E|$ for an undirected graph.



Adjacency list

vertex	*edges*
ATL:	[ICN, JFK, LAX, ORD]
ICN:	[ATL, JFK, LAX]
JFK:	[ATL, ICN]
LAX:	[ICN, ATL, ORD]
ORD:	[ATL, LAX]

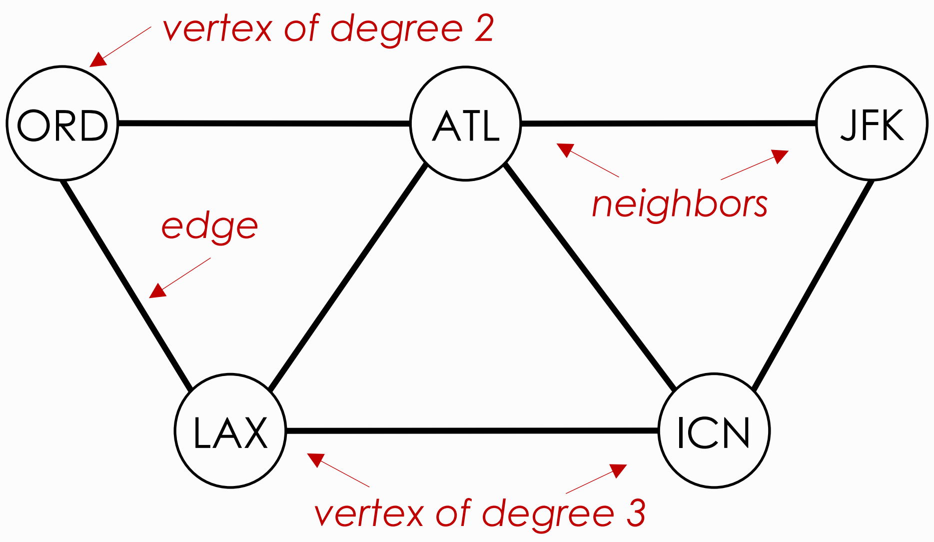
Use an array, dict, or hash table

Graph Applications

- Graphs serve as models of a wide range of objects:
- **Transportation systems:** Subway tracks connect stations, roads connect intersections, and airline routes connect airports, so all these systems naturally admit a simple graph model. What is the best way to get from here to there?
- **Social networks:** People have relationships with other people. How does information propagate in online networks?
- **Communication systems:** From electric circuits, to the telephone system, to the Internet, to wireless services, communications systems are all based on the idea of connecting devices. What is the best way to connect the devices?
- **Financial systems:** Transactions connect accounts, and accounts connect customers to financial institutions. Which transactions are routine or not?

Graph ADT

- Processing graphs typically involves building a graph from information in a database and then answering question about the graph. For example,
  - **How many** vertices and edges does the graph have?
  - Which are **neighbors** of a given vertex?
  - Is there **a path** connecting two given vertices?
  - What is the **shortest path** of two given vertices?



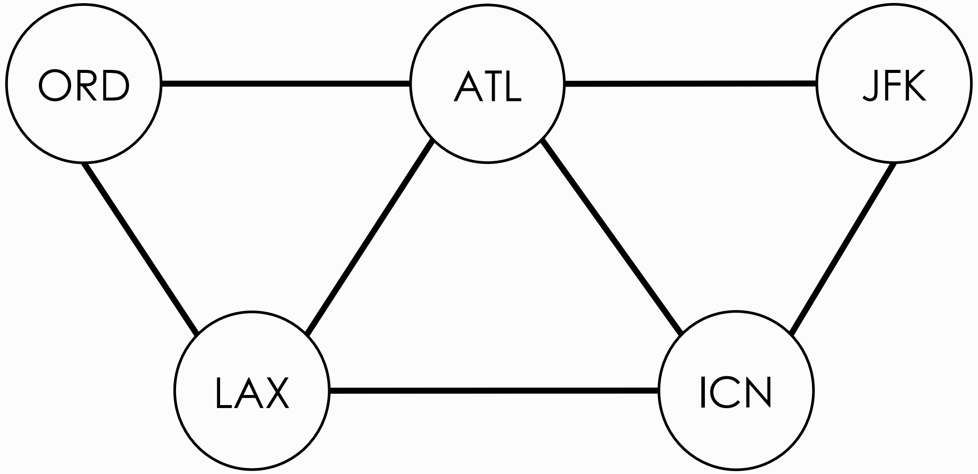
Graph ADT

- Graph processing algorithms first build an internal memory representation of a graph by adding edges, then process it by iterating through the vertices and through edges that are adjacent to a vertex.

Operations	Description
`g = Graph()`	construct a new Graph object g
`g.addEdge(v, w)`	add two edges v-w and w-v to g for undirected
`g.countV()`	the number of vertices in g
`g.countE()`	the number of edges in g
`g.degree(v)`	the number of neighbors of v in g
`g.hasVertex(v)`	is v a vertex in g?
`g.hasEdge(v, w)`	is v-w an edge in g?
`g.vertices()`	an iterable for the vertices of g
`g.neighbors(v)`	an iterable for the neighbors of vertex v in g
`str(g)`	string representation of g

Graph ADT Example

- The internal representation of a graph is a symbol table of sets: the **keys** are vertices, and the values are the sets of **neighbors** - the vertices adjacent to the key.



Use **dict**

vertex	*set of neighbors*
ATL	ICN JFK LAX ORD
ICN	ATL JFK LAX
JFK	ATL ICN
LAX	ATL ICN ORD
ORD	ATL LAX

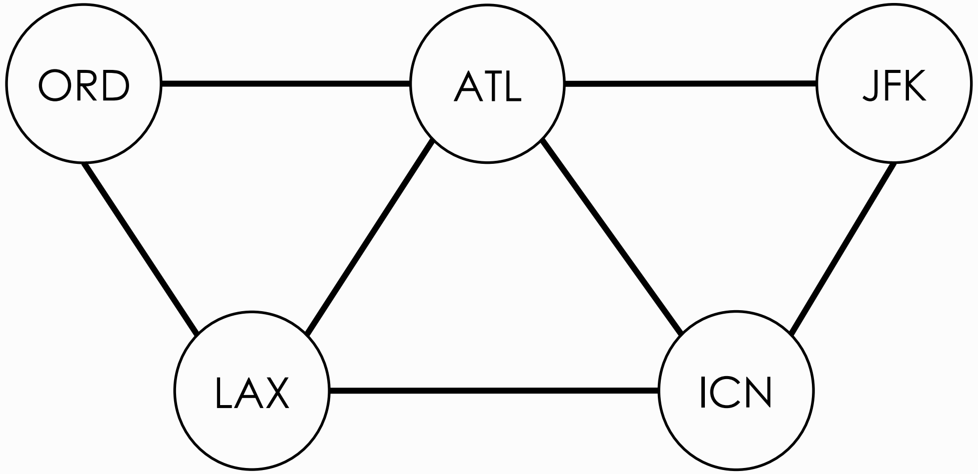
*Use set or **list** type*

key *value*

- *Use **set** for random order access, O(1), but **list** for orderly access, O(n)*

Graph ADT Example

- The internal representation of a graph is a symbol table of sets: the **keys** are vertices, and the values are the sets of **neighbors** - the vertices adjacent to the key.



vertex	*set of neighbors*
ATL	ICN JFK LAX ORD
ICN	ATL JFK LAX
JFK	ATL ICN
LAX	ATL ICN ORD
ORD	ATL LAX

*Use set or **list** type*

*Use **dict***

key

value

Internal representation of the graph

```
graph = { 'ATL': ['ICN', 'JFK', 'LAX', 'ORD'],
          'JFK': ['ICN', 'ATL'],
          'ICN': ['JFK', 'ATL', 'LAX'],
          'LAX': ['ICN', 'ATL', 'ORD'],
          'ORD': ['LAX', 'ATL'] }
```

Graph ADT

```
#%%writefile graph.py
class Graph:
    def __init__(self, filename=None, delimiter=None):
        self._e = 0
        self._adj = dict()                              # adjacency list
        if filename is not None:
            with open(filename, 'r') as f:
                for line in f.read().splitlines():  # read all lines
                    names = line.split(delimiter)   # get names for two vertices
                    for i in range(1, len(names)):  # add edges - undirected
                        self.addEdge(names[0], names[1])

    def __str__(self):                              # string representation of graph
        ...
```

- This code uses the built-in types **dict** and **list** to implement the graph data type.
- Clients can build graphs by adding edges one at a time or by reading from a file.
  They can process graphs by iterating over the set of all vertices or over the set of vertices adjacent to a given vertex.

Graph ADT

```
#%%writefile graph.py
class Graph:
    def __init__(self, filename=None, delimiter=None):
        ...

    def __str__(self):                              # string representation of graph
        s = ''
        for v in self.vertices():
            s += v + ': ' + ' '.join([w for w in self.neighbors(v)]) + '\n'
        return s
    ...
```

- A natural way to write a Graph is to put the vertices one per line, each followed by a list of its immediate neighbors. Accordingly, we support the built-in function **`str()`** by implementing **`__str__()`** as shown above:

a vertex and its neighbors

__str__()

Sample Output:

```
ATL: ICN JFK LAX ORD
JFK: ICN ATL
ICN: JFK ATL LAX
LAX: ICN ATL ORD
ORD: LAX ATL
```

Building a graph Example

```
#%%writefile graph.py
class Graph:
    def __init__(self, filename=None, delimiter=None):
        ...

    def __str__(self):                                  # string representation of graph
        ...

    def addEdge(self, v, w):                            # add edges v-w and w-v to graph
        if not self.hasVertex(v): self._adj[v] = list()  # set() for random access O(1)
        if not self.hasVertex(w): self._adj[w] = list()  # list() for orderly access, O(n)
        if not self.hasEdge(v, w):
            self._e += 1
            self._adj[v].append(w)                      # for list type, use add() for set
            self._adj[w].append(v)

    def neighbors(self, v):   return iter(self._adj[v])  # iterable for neighbors of v
    def vertices(self):       return iter(self._adj)     # iterable for the vertices of graph
    def hasVertex(self, v):   return v in self._adj      # is v a vertex in graph
    def hasEdge(self, v, w):  return w in self._adj[v]   # is v-w and edge in graph
    def countV(self):         return len(self._adj)      # the number of vertices in graph
    def countE(self):         return self._e             # the number of edges in graph
    def degree(self, v):      return len(self._adj[v])   # the number of neighbors of v
```

Exercise: Build a graph from a file

```
if __name__ == "__main__":
    g = Graph("route5.txt")
    s = 'ATL'
    print('no. of vertices:', g.countV())
    print
    print
    print
    print
    print
    print
```

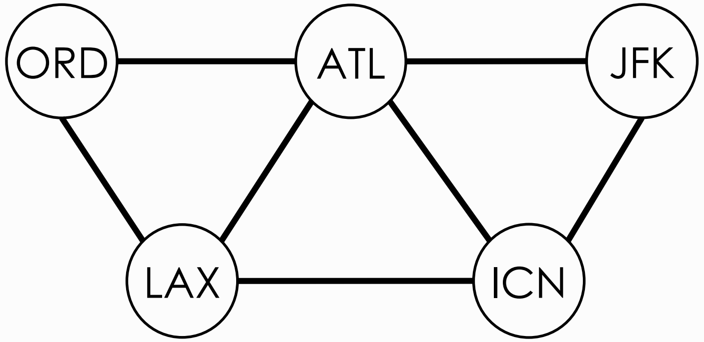
```
%%writefile route5.txt
ATL ICN
ATL JFK
ICN JFK
LAX ICN
ATL LAX
ORD ATL
ORD LAX
```

Complete the test code to produce the sample output as shown below:

```
no. of vertices: 5
no. of edges: 7
vertices: ['ATL', 'ICN', 'JFK', 'LAX', 'ORD']
degree of ATL: 4
neighbors of ATL: ['ICN', 'JFK', 'LAX', 'ORD']
graph:
ATL: ICN JFK LAX ORD
ICN: ATL JFK LAX
JFK: ATL ICN
LAX: ICN ATL ORD
ORD: ATL LAX

adjacency list:
 {'ATL': ['ICN', 'JFK', 'LAX', 'ORD'], 'ICN': ['ATL', 'JFK', 'LAX'],
'JFK': ['ATL', 'ICN'], 'LAX': ['ICN', 'ATL', 'ORD'],
'ORD': ['ATL', 'LAX']}
```

Using an edge list in a file



Summary

- A graph is a **non-linear data structure** consisting of vertices and edges between vertices.
  - In undirected graph, an edge that connects v to w is the same as one that connects w to v.
  - Directed graph is called as a **digraph**, directed graph without a cycle is called as a **DAG**(or directed acyclic graph).
- Three most used graph representations:
  - **Edge lists** are commonly used to save graphs in a file.
  - **Adjacency matrices** are more efficient when finding the relationships in a graph but takes too much space.
    Adjacency lists are more efficient for the storage of the graph, especially **sparse** graphs, when there is a lot less edges than vertices.

학습 정리

1) 그래프는 비선형 자료 구조이다

2) 그래프를 표현하는 방법들은 간선리스트(edge list), 인접 행렬 (adjacency matrix), 인접 리스트(adjacency list)이다


파이썬으로 배우는
데이터 구조
수고했습니다
곧 다음 시간에
다시 뵙겠습니
다