# Principles/Social Media Mining

## CIS 600

## Weeks 5, 6: Social Network Analysis, Part 1: Graph Basics and Networkx

**Edmund Yu, PhD**
**Associate Teaching Professor**
**esyu@syr.edu**

**September 24, 29, 2020**

# Sorting in Python
# Needed for Assignment #2

# Sorting: Needed for Assignment #2

```
# the sort() function for lists
a = [5, 2, 3, 1, 4]
a.sort()
print(a)

# the sorted() function for lists
b = [5, 2, 3, 1, 4]
c = sorted(b) # or c = sorted(b, reverse=True)
print(b)
print(c)
```

**sorted**(*iterable*, *, *key=None*, *reverse=False*)

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

*key* specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

*reverse* is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use `functools.cmp_to_key()` to convert an old-style *cmp* function to a *key* function.

The built-in `sorted()` function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see Sorting HOW TO.

**@staticmethod**

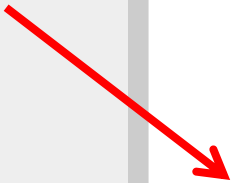
Transform a method into a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

# Sorting: Needed for Assignment #2

```
# sorted() for dictionaries
d = {"five": 5, "two": 2, "three": 3, "one": 1, "four": 4}
e = sorted(d)
f = sorted(d, key=d.get) # get() returns the value for a key
print(d)
print(e)
print(f)
```



**sorted**(*iterable*, *, *key=None*, *reverse=False*)

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

*key* specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

*reverse* is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use `functools.cmp_to_key()` to convert an old-style *cmp* function to a *key* function.

The built-in `sorted()` function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see Sorting HOW TO.

**@staticmethod**

Transform a method into a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

# Mapping Types — dict

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the *dictionary*. (For other containers see the built-in `list`, `set`, and `tuple` classes, and the `collections` module.)

A dictionary's keys are *almost* arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of `key: value` pairs within braces, for example: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`, or by the `dict` constructor.

*class* **dict**(***kwarg*)
*class* **dict**(*mapping*, ***kwarg*)
*class* **dict**(*iterable*, ***kwarg*)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created

**iter(d)**

Return an iterator over the keys of the dictionary. This is a shortcut for `iter(d.keys())`.

**clear**()

Remove all items from the dictionary.

**copy**()

Return a shallow copy of the dictionary.

*classmethod* **fromkeys**(*iterable*[, *value*]) ¶

Create a new dictionary with keys from *iterable* and values set to *value*.

`fromkeys()` is a class method that returns a new dictionary. *value* defaults to `None`.

**get**(*key*[, *default*])

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

**items**()

Return a new view of the dictionary's items (`(key, value)` pairs). See the documentation of view objects.

**keys**()

Return a new view of the dictionary's keys. See the documentation of view objects.

**pop**(*key*[, *default*])

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

**popitem**()

# Sorting: Needed for Assignment #2

```
# sorted() for tuples
student_tuples = [('John', 'C', 25), ('Jane', 'A', 22), ('Dave', 'B', 20)]


s1 = sorted(student_tuples)
print("Sort by name: ", s1)
s2 =sorted(student_tuples, key=lambda student: student[1])
print("Sort by grade: ", s2)
s3 = sorted(student_tuples, key=lambda student: student[2])
print("Sort by age: ", s3)
```

# Lambda Expressions (Review)

- Small anonymous functions can be created with the lambda keyword. The following example uses a lambda expression to return a function.

```
def make_incrementor(n):
    return lambda x: x + n


f = make_incrementor(42)
print(f(0))
42
print(f(1))
43
```

# Lambda Expressions (Review)

- Another use is to pass a small function as an argument.

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])

print(pairs)
```

*[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]*

# Sorting: Needed for Assignment #2

```
# sorted() for tuples using itemgetter
student_tuples = [('John', 'C', 25), ('Jane', 'A', 22), ('Dave', 'B', 20)]


# sorted() for tuples, using itemgetter
from operator import itemgetter
s4 = sorted(student_tuples, key=itemgetter(2))
print("Sort by age (using itemgetter): ", s4)
```

# operator — Standard operators as functions

**Source code:** Lib/operator.py

---

The operator module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, operator.add(x, y) is equivalent to the expression x+y. Many function names are those used for special methods, without the double underscores. For backward compatibility, many of these have a variant with the double underscores kept. The variants without the double underscores are preferred for clarity.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations and sequence operations.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

operator.**lt**(*a*, *b*)
operator.**le**(*a*, *b*)
operator.**eq**(*a*, *b*)
operator.**ne**(*a*, *b*)
operator.**ge**(*a*, *b*)
operator.**gt**(*a*, *b*)
operator.**__lt__**(*a*, *b*)
operator.**__le__**(*a*, *b*)
operator.**__eq__**(*a*, *b*)
operator.**__ne__**(*a*, *b*)
operator.**__ge__**(*a*, *b*)

operator.**itemgetter**(*item*)
operator.**itemgetter**(**items*)

Return a callable object that fetches *item* from its operand using the operand's `__getitem__()` method. If multiple items are specified, returns a tuple of lookup values. For example:

- After `f = itemgetter(2)`, the call `f(r)` returns `r[2]`.
- After `g = itemgetter(2, 5, 3)`, the call `g(r)` returns `(r[2], r[5], r[3])`.

Equivalent to:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

The items can be any type accepted by the operand's `__getitem__()` method. Dictionaries accept any hashable value. Lists, tuples, and strings accept an index or a slice:

```
>>> itemgetter('name')({'name': 'tu', 'age': 18})
'tu'
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1,3,5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFG')
'CDEFG'
```

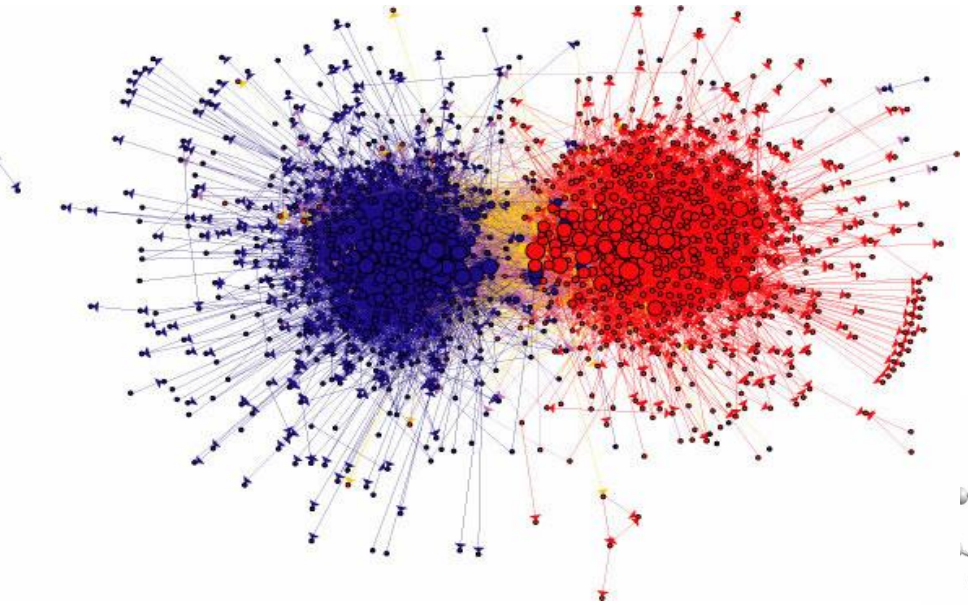
# Sorting By Multiple Keys

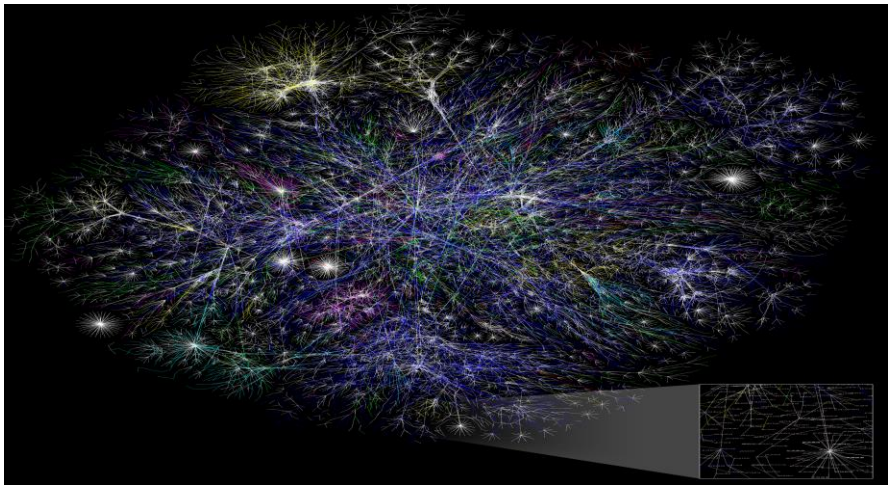
```
student_tuples = [('John', 'C', 25), ('Jane', 'A', 22), ('Dave', 'B', 20),
                  ('Ed', 'C', 30)]

s5 = sorted(student_tuples, key = lambda student: (student[1],
                    student[0]))   # sort by grade and then by name
print("Sort by grade, then by name: ", s5)

from operator import itemgetter
s6 = sorted(student_tuples, key = itemgetter(1, 0))
print("Sort by grade, then by name (using itemgetter): ", s6)
```

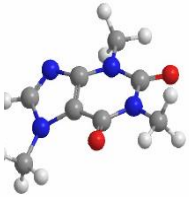
# Networks

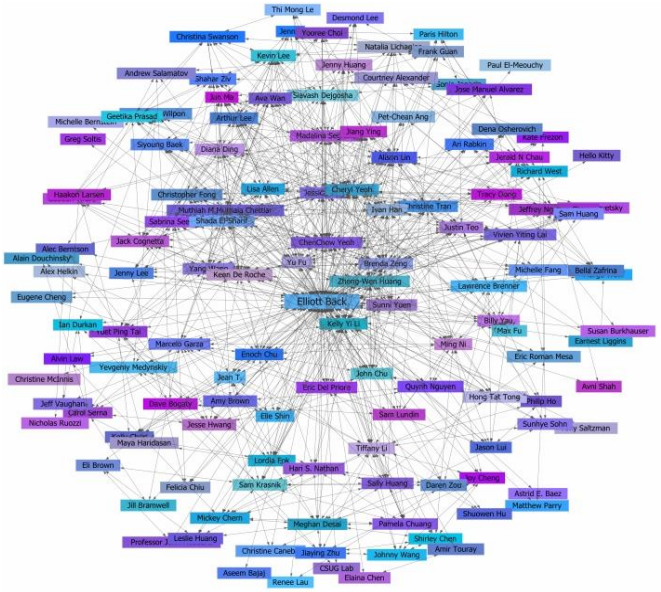
# Networks

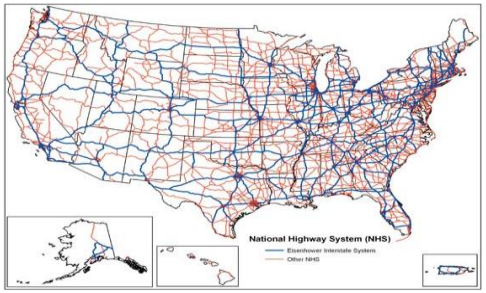
- Networks are everywhere:



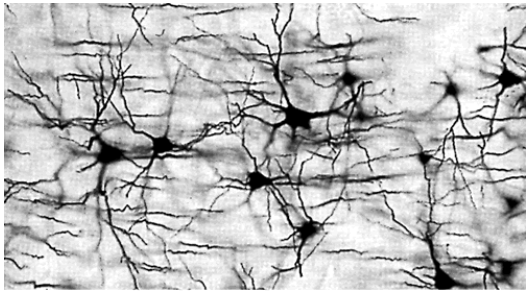






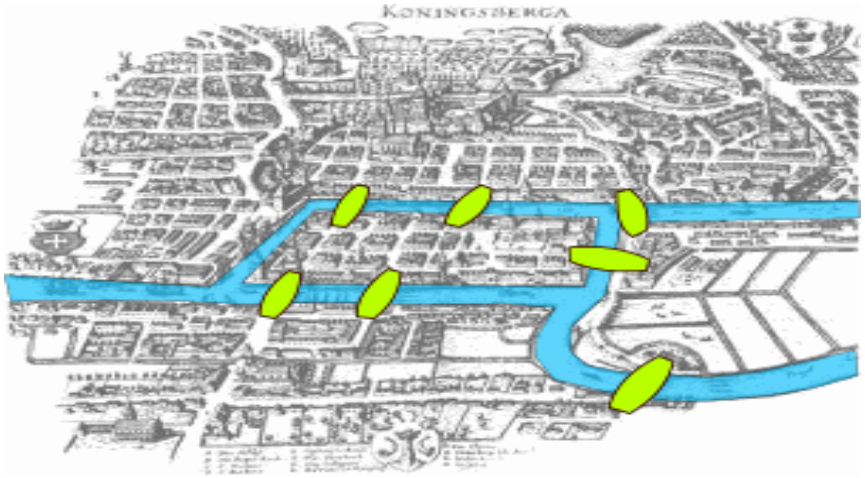



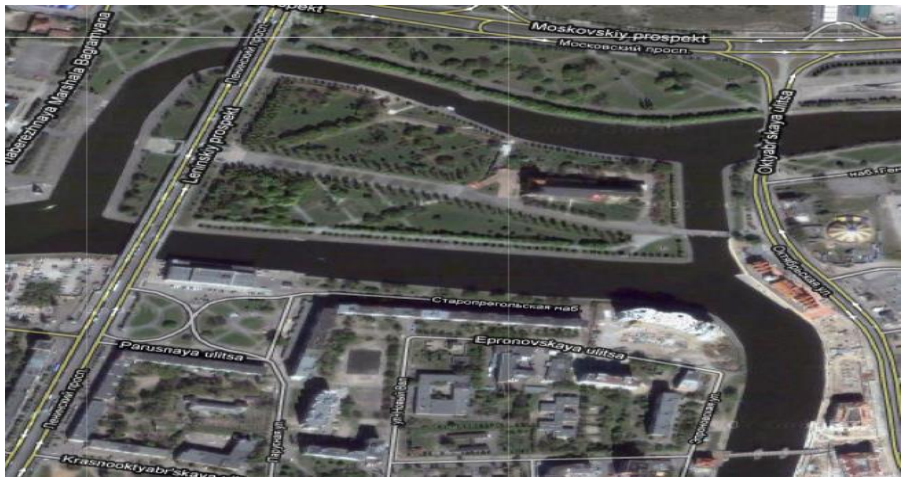

Chemical Compound



# Networks & Graph Theory

- Networks have been studied for a long time (Euler 1736):
  - The 7 Bridges of Konigsberg problem: devise a walk through the city that would cross each of those bridges once and only once.
    - The **story** goes like this: the philosopher Immanual Kant lived in Konigsberg at the time, and was well known for his regular long walks around the city.
    - He was frustrated by the fact that he had to pass over the same bridge on his way home as he took in the beginning of this walk. So he mentioned this to his friend Leonhard Euler.



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

# Leonhard Euler

From Wikipedia, the free encyclopedia

*"Euler" redirects here. For other uses, see Euler (disambiguation).*

**Leonhard Euler** (/ˈɔɪlər/ *OY-lər*;[2] German: [ˈɔʏlɐ] (listen); 15 April 1707 – 18 September 1783) was a Swiss mathematician, physicist, astronomer, geographer, logician and engineer who made important and influential discoveries in many branches of mathematics, such as infinitesimal calculus and graph theory, while also making pioneering contributions to several branches such as topology and analytic number theory. He also introduced much of the modern mathematical terminology and notation, particularly for mathematical analysis, such as the notion of a mathematical function.[3] He is also known for his work in mechanics, fluid dynamics, optics, astronomy and music theory.[4]

Euler was one of the most eminent mathematicians of the 18th century and is held to be one of the greatest in history. He is also widely considered to be the most prolific mathematician of all time. His collected works fill 92 volumes,[5] more than anyone else in the field. He spent most of his adult life in Saint Petersburg, Russia, and in Berlin, then the capital of Prussia.

A statement attributed to Pierre-Simon Laplace expresses Euler's influence on mathematics: "Read Euler, read Euler, he is the master of us all."[6][7]

Contents [hide]



**Leonhard Euler**

Portrait by Jakob Emanuel Handmann (1753)

| | |
|---|---|
| **Born** | 15 April 1707<br>Basel, Switzerland |
| **Died** | 18 September 1783<br>(aged 76)<br>[OS: 7 September 1783]<br>Saint Petersburg, Russian |


# Seven Bridges of Königsberg

From Wikipedia, the free encyclopedia

Coordinates: 54°42′12″N 20°30′56″E

*This article is about an abstract problem. For the historical group of bridges in the city once known as Königsberg, and those of them that still exist, see § Present state of the bridges.*

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.

*Find sources:* "Seven Bridges of Königsberg" – news · newspapers · books · scholar · JSTOR *(July 2015) (Learn how and when to remove this template message)*

The **Seven Bridges of Königsberg** is a historically notable problem in mathematics. Its negative resolution by Leonhard Euler in 1736[1] laid the foundations of graph theory and prefigured the idea of topology.[2]

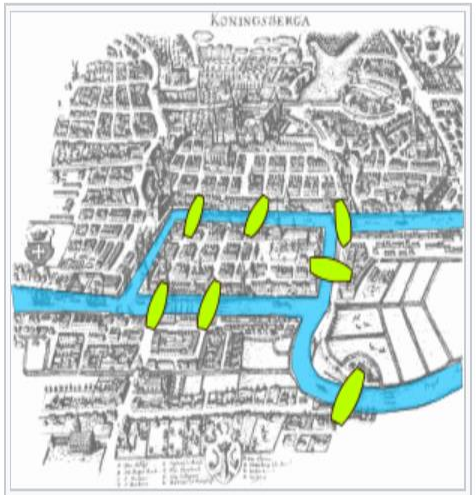
The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands—Kneiphof and Lomse—which were connected to each other, or to the two mainland portions of the city, by seven bridges. The problem was to devise a walk through the city that would cross each of those bridges once and only once.

By way of specifying the logical task unambiguously, solutions involving either

1. reaching an island or mainland bank other than via one of the bridges, or
2. accessing any bridge without crossing to its other end

are explicitly unacceptable.

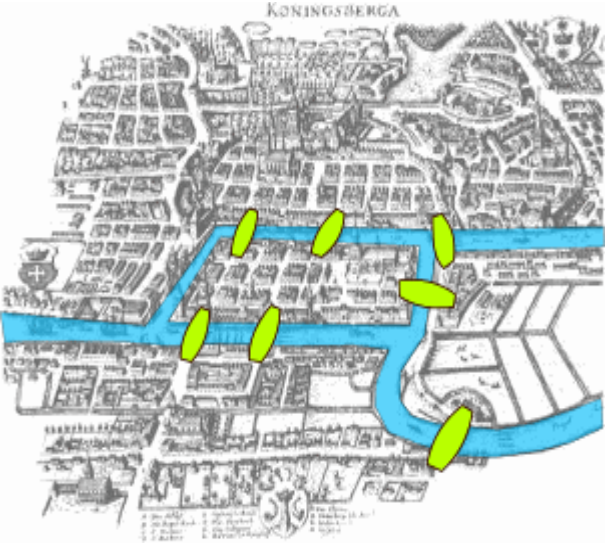
Euler proved that the problem has no solution. The difficulty he faced was the development of a suitable technique of analysis, and of subsequent tests that established this assertion with mathematical rigor.

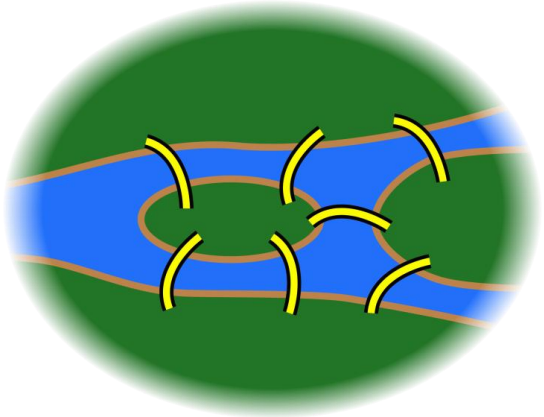


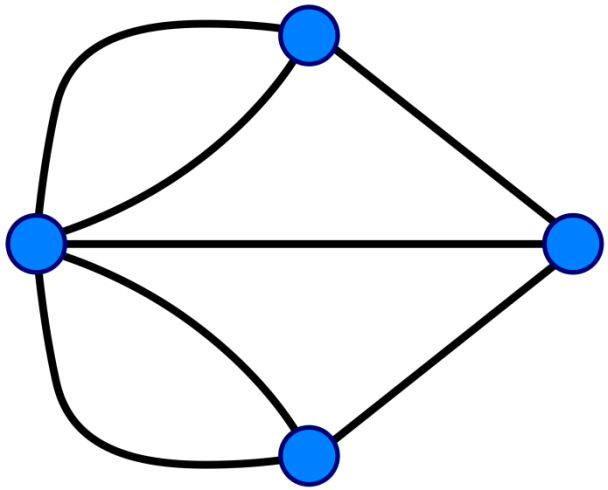

Map of Königsberg in Euler's time showing the actual layout of the seven bridges, highlighting the river Pregel and the bridges

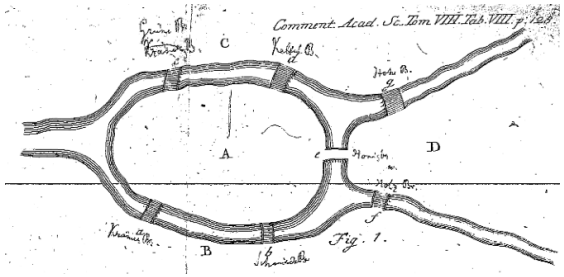
# Networks & Graph Theory

- Euler transformed this **7 Bridges of Konigsberg** problme into a graph, and accidentally invented a new branch of mathematics - **Graph Theory**.



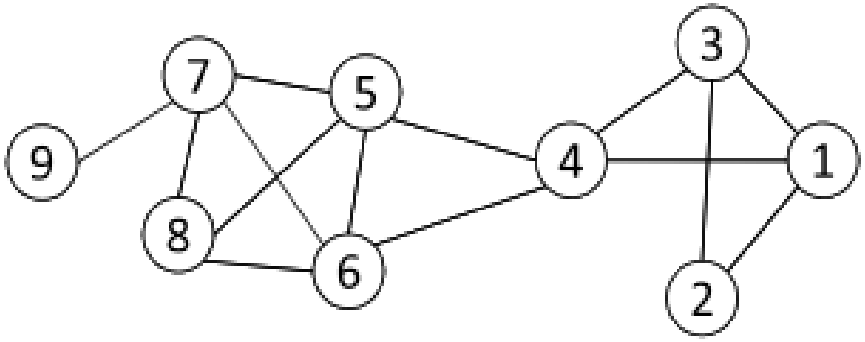




# Networks and Representation

A graph (**social network)** is made of **nodes** (individuals or organizations) and **edges** that connect nodes in various relationships like friendship, kinship etc.

❖ Graph Representation




❖ Adjacency Matrix Representation

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | - | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | - | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 1 | - | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 1 | - | 1 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | - | 1 | 1 | 1 | 0 |
| 6 | 0 | 0 | 0 | 1 | 1 | - | 1 | 1 | 0 |
| 7 | 0 | 0 | 0 | 0 | 1 | 1 | - | 1 | 1 |
| 8 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | - | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | - |

# Graphs: Basic Definitions

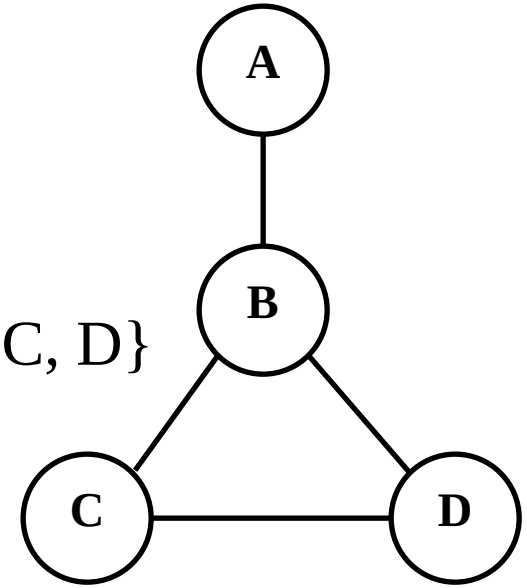
- Two nodes are **neighbors** if they are connected by an edge.
- The number of neighbors a node has is called its **degree**.

neighbors(A) = {B}

degree(A) = 1

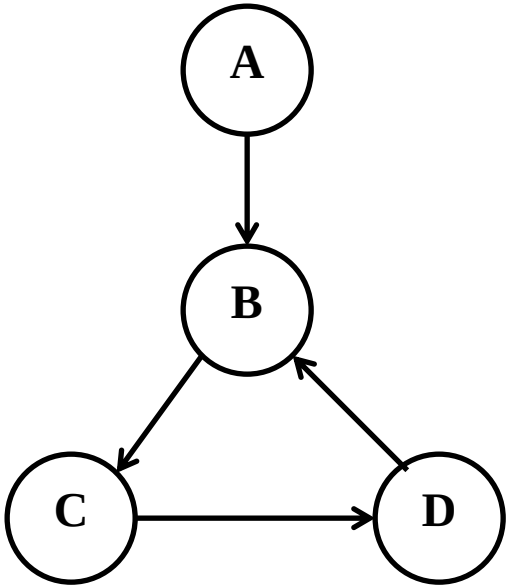
neighbors(B) = {A, C, D}

degree(B) = 3



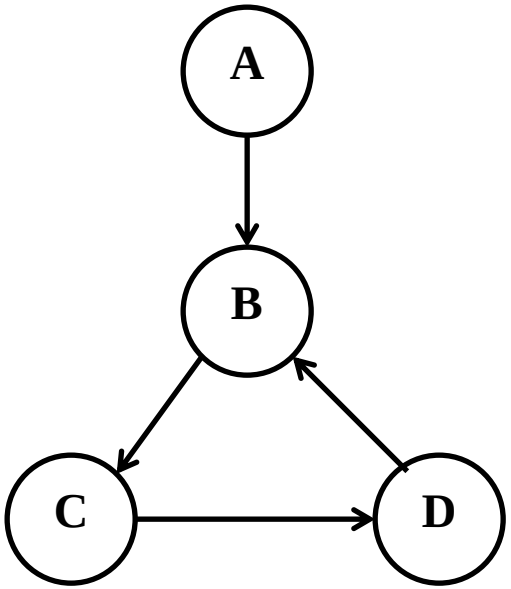
# Graphs: Basic Definitions

- A **directed** graph, as opposed to a un-directed graph in the previous example, consists of a set of nodes, as before, together with a set of **directed edges**
  - Each directed edge is a link from one node to another, with the direction being important.



# Graphs: Basic Definitions

- In Directed graphs:
  - **In-degrees** is the number of edges pointing towards a node
  - **Out-degree** is the number of edges pointing away from a node




In-degree(A) = 0

Out-degree(A) = 1

In-degree(B) = 2

Out-degree(B) = 1

# Graphs: Basic Definitions

**Theorem 1.** (Textbook #2) The summation of degrees in an undirected graph is twice the number of edges

$$\sum_i d_i = 2|E|$$

**Lemma 1.** The number of nodes with odd degree is even

**Lemma 2.** In any directed graph, the summation of in-degrees is equal to the summation of out-degrees,

$$\sum_i d_i^{out} = \sum_j d_j^{in}$$

# Degree Distribution

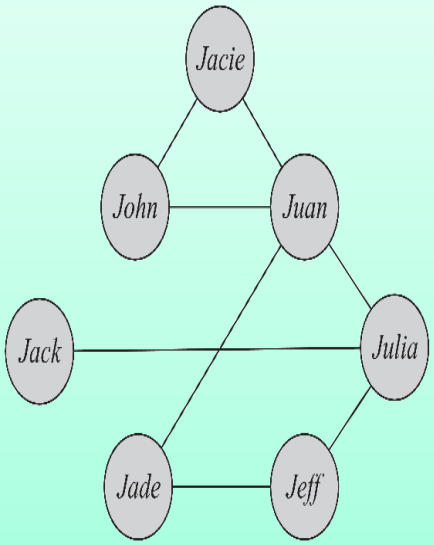
- When dealing with very large graphs, how nodes' degrees are distributed is an important concept to analyze and is called **Degree Distribution**

$$\pi(d) = \{d_1, d_2, \ldots, d_n\}$$

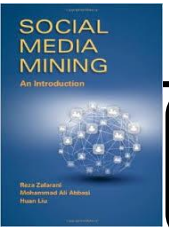
$n_d$ is the number of nodes with degree $d$

$$p_d = \frac{n_d}{n}$$

$$\sum_{d=0}^{\infty} p_d = 1$$




$$p_1 = \frac{1}{7}, p_2 = \frac{4}{7}, p_3 = \frac{1}{7}, p_4 = \frac{1}{7}$$

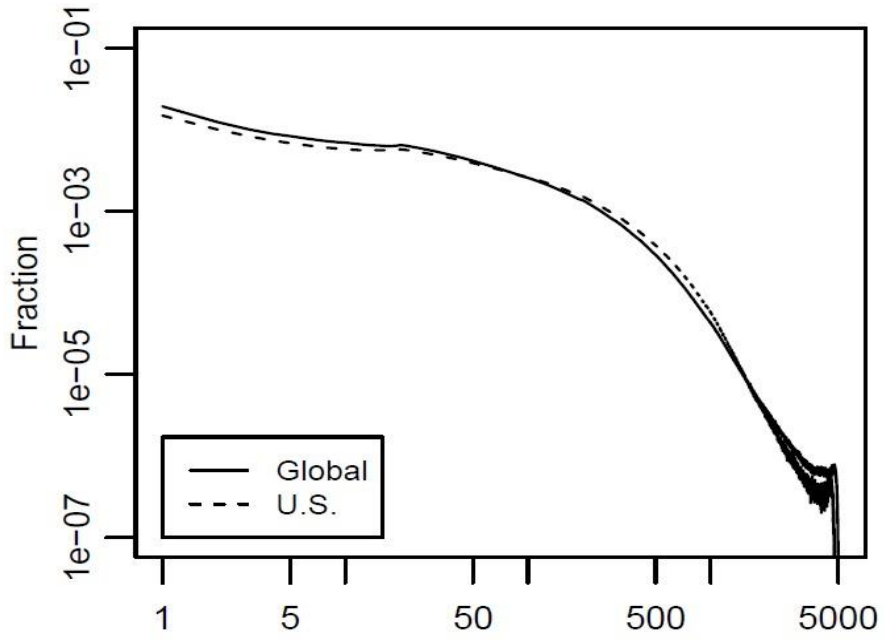


# Degree Distribution Plot

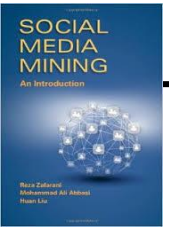
- On social networking sites, there exist many users with few connections and there exist a handful of users with very large numbers of friends.

  → **Power-law degree distribution**

- The $x$-axis represents the degree and the $y$-

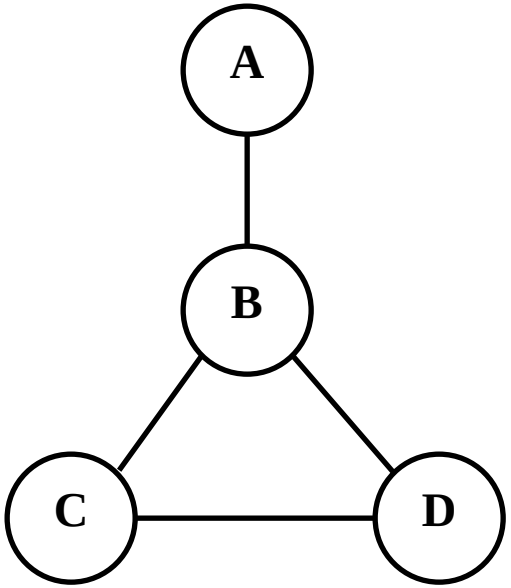



**Facebook Degree Distribution**



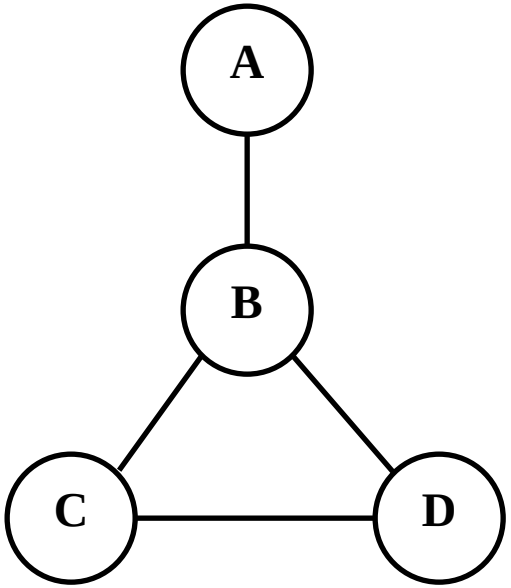
# Graphs: Basic Definitions

- A **path** is a sequence of nodes with the property that each consecutive pair in the sequence is connected by an edge.
  - A → B → C → D is a path; A → D is not



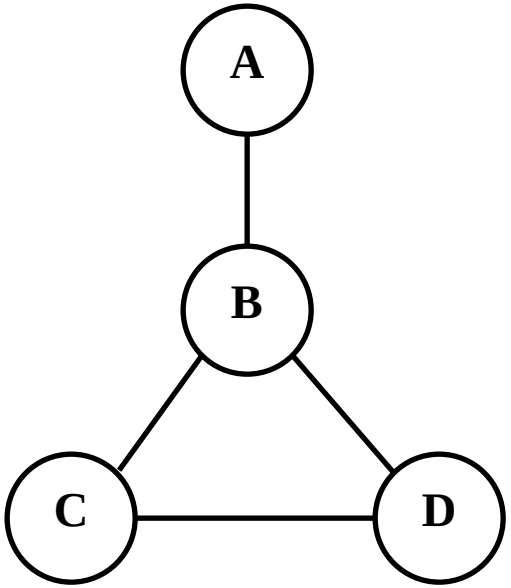
# Graphs: Basic Definitions

- A **simple path** is a path that does not repeat nodes.
  - A → B → C → D is a simple path
  - A → B → C → D → B is not



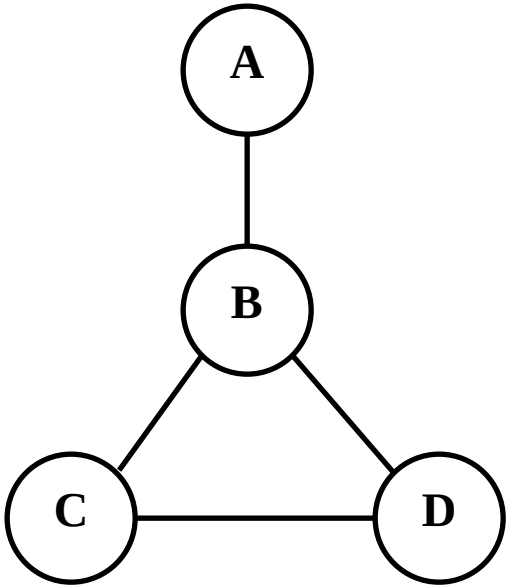
# Graphs: Basic Definitions

- A **cycle** is a path with at least **3** edges, in which the 1st and last nodes are the same, but otherwise all nodes are distinct.
  - B → C → D → B is a cycle; A → B → A is not



# Graphs: Basic Definitions

- A graph is **connected** if for every pair of nodes, there is a path between them.
  - Is the following graph connected? (How many pairs of nodes are there in a graph of ***n*** nodes?)

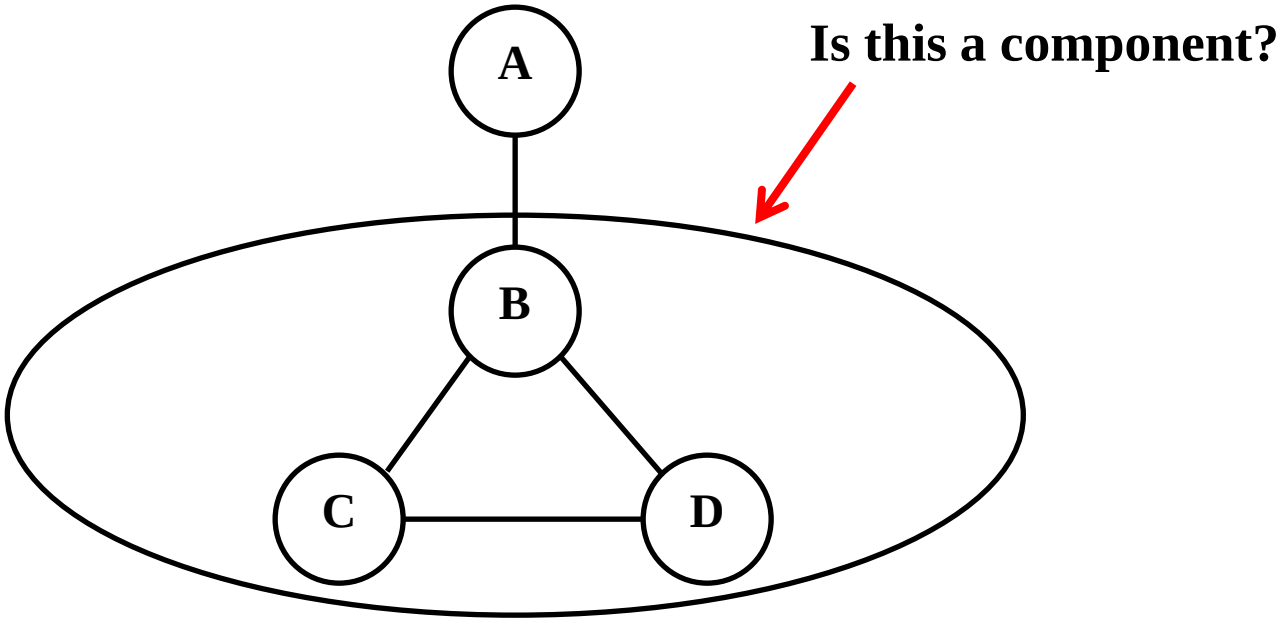


# Graphs: Basic Definitions

- A **component** of a graph is a subset of the nodes such that:

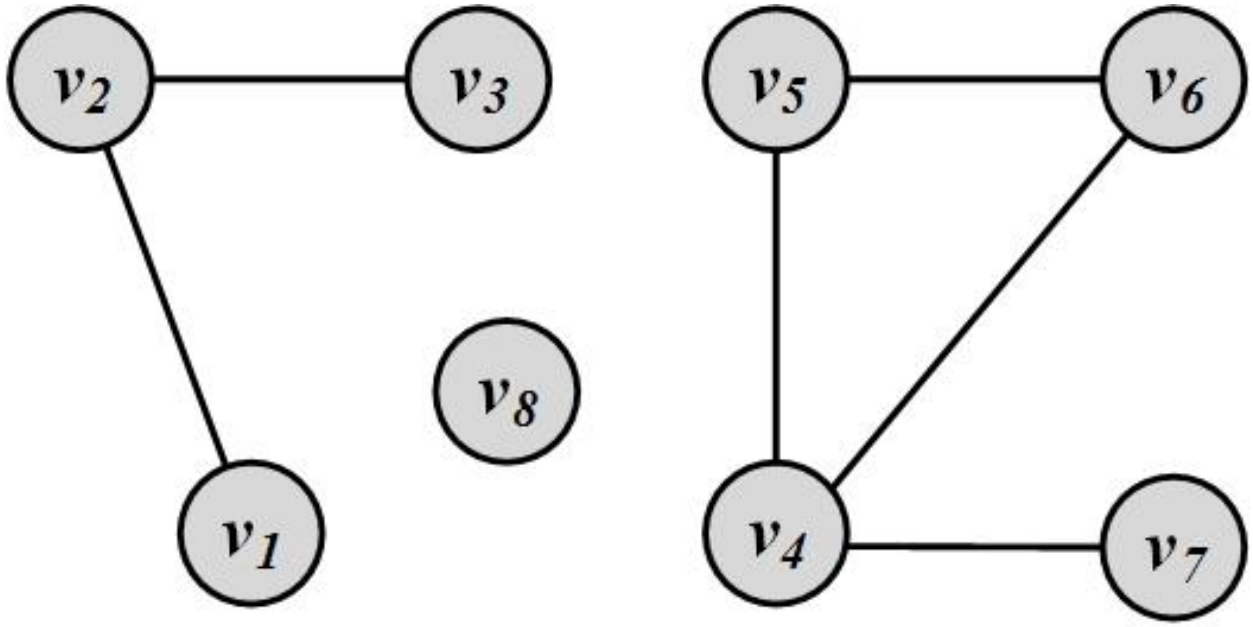
  (i) every node in the subset has a path to every other

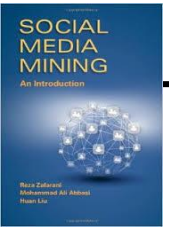
  (ii) the subset is not part of some larger set with the property that every node can reach every other.



# Graphs: Basic Definitions

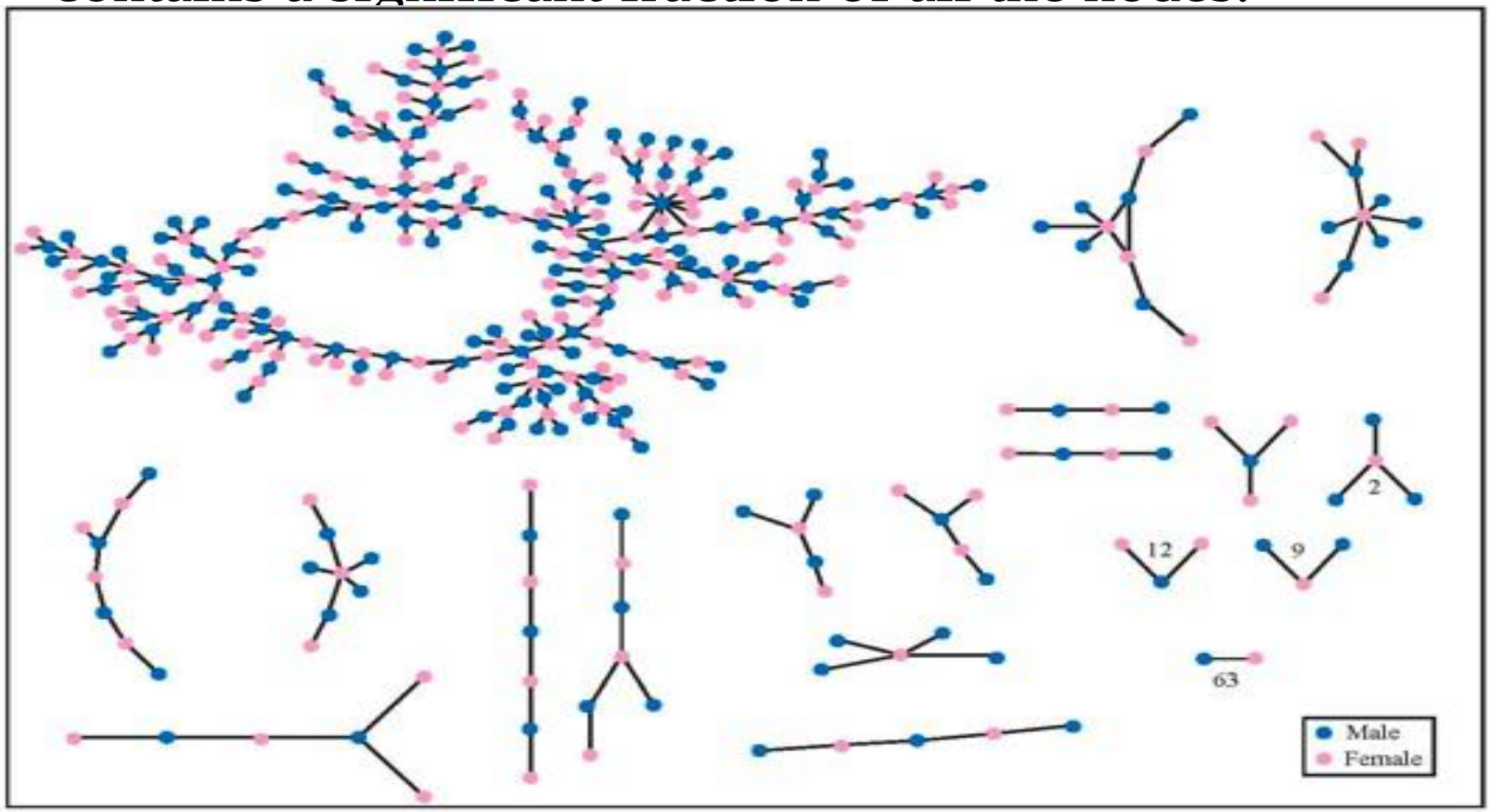
- 3 components:



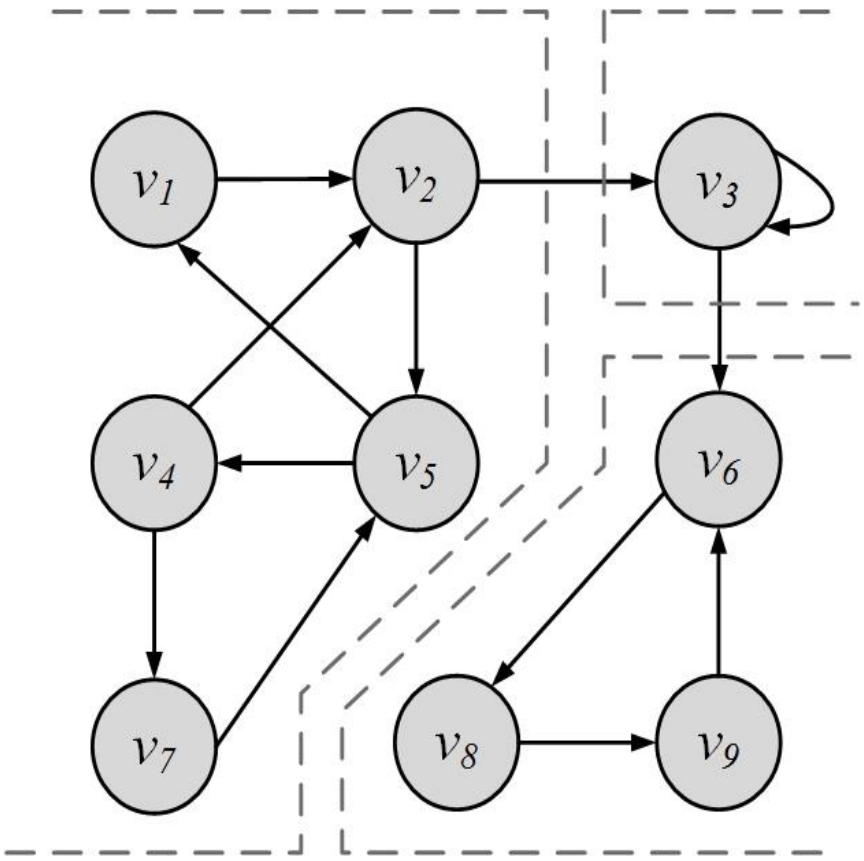
# Graphs: Basic Definitions

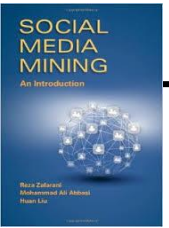
- A **giant** component is a (connected) component that contains a significant fraction of all the nodes.



# Graphs: Basic Definitions

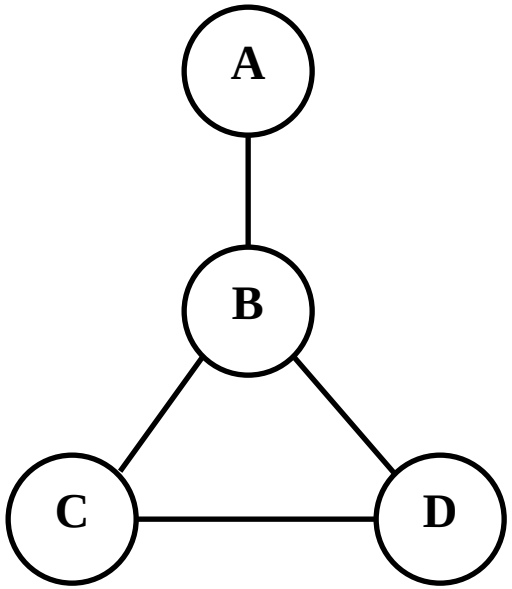
- In directed graphs, we have a **strongly connected** components when there is a path from $u$ to $v$ and one from $v$ to $u$ for every pair of nodes $u$ and $v$.
- The component is **weakly connected** if replacing directed edges with undirected edges results in a connected component



# Graphs: Basic Definitions

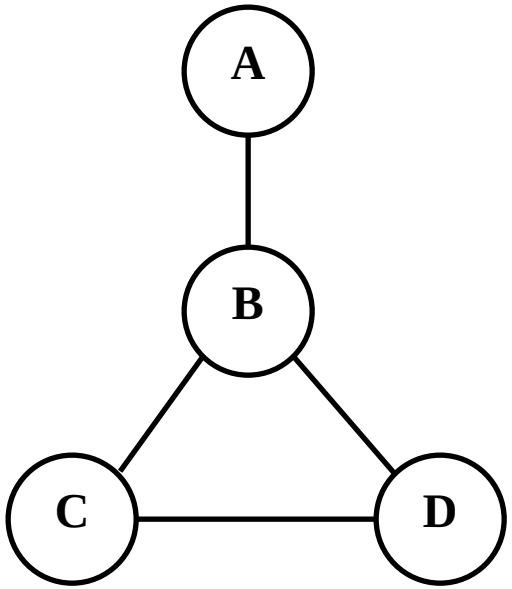
- The **length** of a path is the number of steps it contains from beginning to end, in other words, the number of edges in the sequence that comprises it.
  - The path of A → B → C → D has length of 3
  - The path of A → B → D has length of 2



# Graphs: Basic Definitions

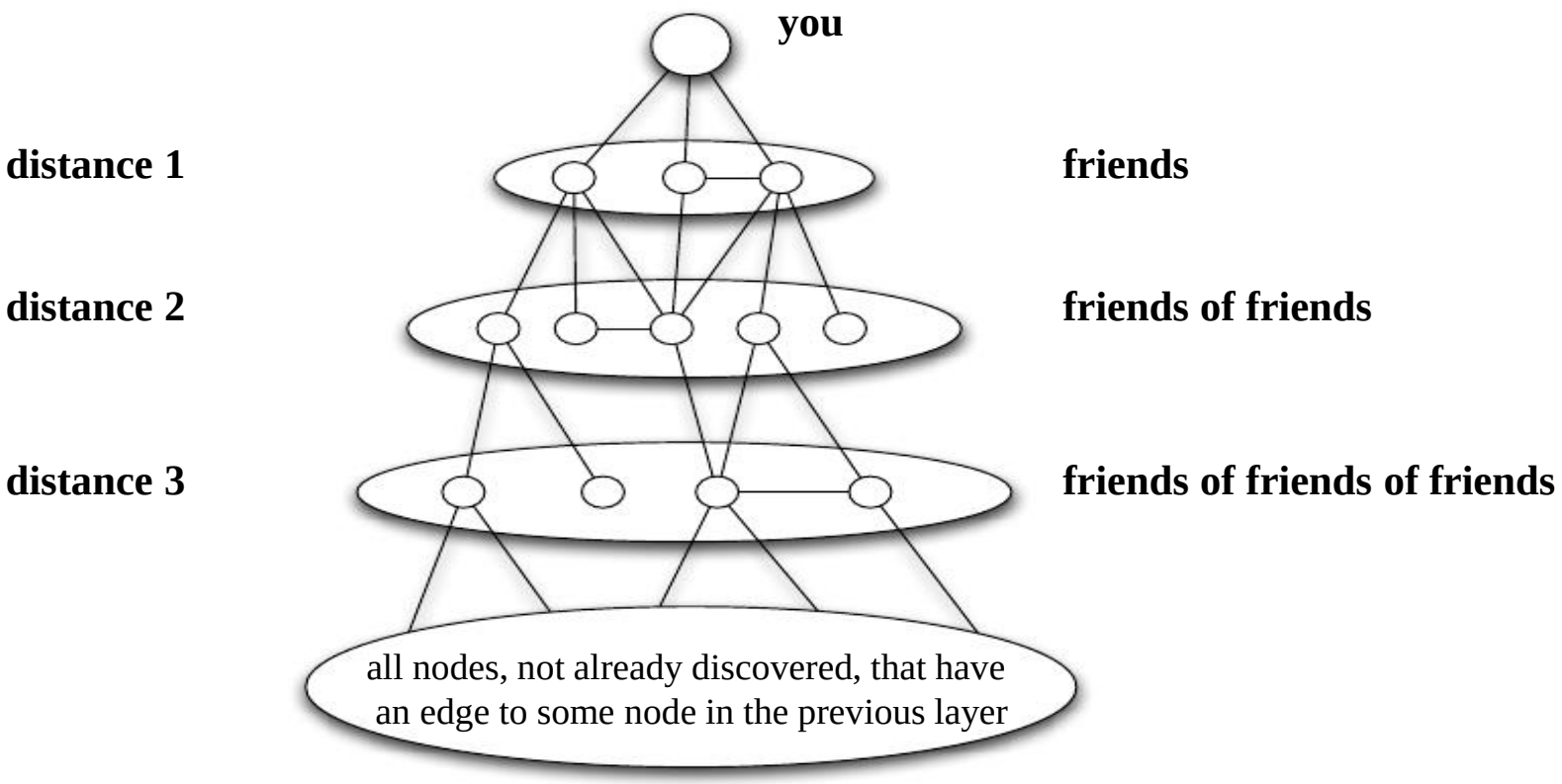
- The **distance** between two nodes in a graph to be the length of the **shortest** path between them.
  - There are two paths from A to D:
    - A → B → C → D, or A → B → D
    - A → B → D is the shortest, and hence the distance is 2

We denote the length of the shortest path between nodes $v_i$ and $v_j$ as $l_{i,j}$



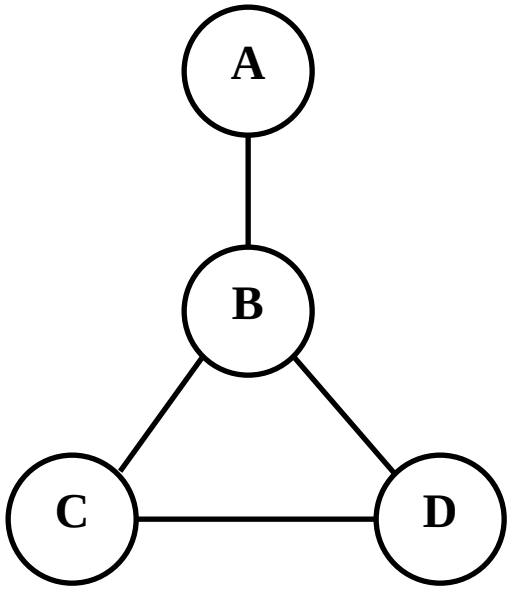
# Graphs: Basic Definitions

- For graphs that are a bit more complicated, we need a systematic method to determine distances: **breadth-first search**



# Graphs: Basic Definitions

- If you need a single aggregate measure to summarize the distances between the nodes in a given graph, there are two natural quantities: **diameter** and **average distance**

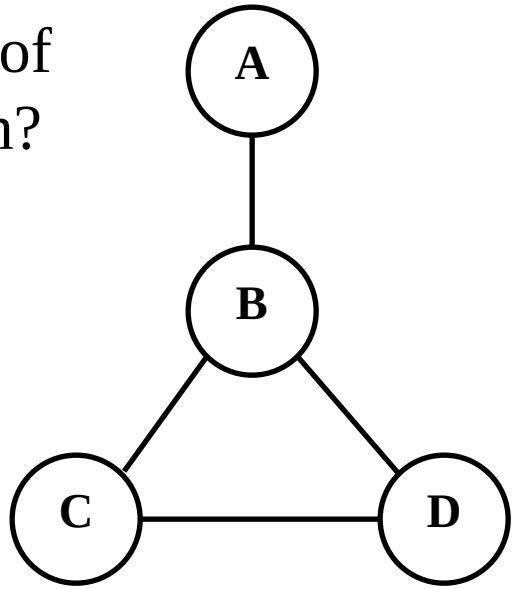


# Graphs: Basic Definitions

- **Diameter** is the maximum distance between any pair of nodes in the graph

$$diameter_G = \max_{(v_i, v_j) \in V \times V} l_{i,j}$$

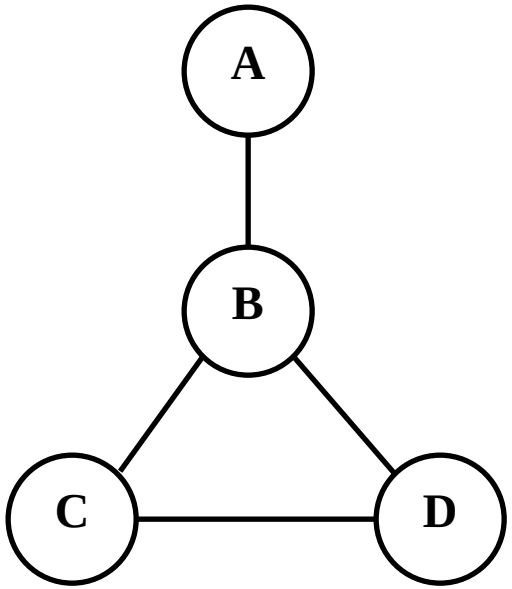
What is the diameter of this graph?




$l_{i,j}$ is the length of the shortest path between nodes $v_i$ and $v_j$

# Graphs: Basic Definitions

- **Average distance** is the average distance over all pairs of nodes in the graph
- What is the average distance of the following graph?



**pip install networkx**

## Contact

Mailing list
Issue tracker
Source

## Releases

### Stable (notes)

2.5 — August 2020
download | doc | pdf

### Latest (notes)

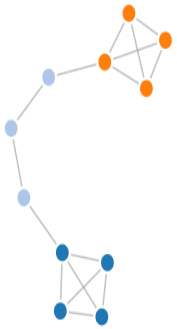
2.6 development
github | doc | pdf

### Archive

# NetworkX

Network Analysis in Python

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.



## Software for complex networks

- Data structures for graphs, digraphs, and multigraphs
- Many standard graph algorithms
- Network structure and analysis measures
- Generators for classic graphs, random graphs, and synthetic networks
- Nodes can be "anything" (e.g., text, images, XML records)
- Edges can hold arbitrary data (e.g., weights, time-series)
- Open source 3-clause BSD license
- Well tested with over 90% code coverage
- Additional benefits from Python include fast prototyping, easy to teach, and multi-platform

PyPI · The Python Package Index

https://pypi.org

Help Sponsor Log in Register

# Find, install and publish Python packages with the Python Package Index

Search projects

Or browse projects

263,411 projects 2,091,336 releases 3,295,739 files 454,043 users


**The Python Package Index (PyPI) is a repository of software for the Python programming language.**

PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages.

Package authors use PyPI to distribute their software. Learn how to package your Python code for PyPI.

networkx · PyPI

https://pypi.org/project/networkx/

Search projects

Help Sponsor Log in Register

# networkx 2.5

```
pip install networkx
```

Latest version

Released: Aug 22, 2020

Python package for creating and manipulating graphs and networks

Navigation

- Project description
- Release history
- Download files

Project links

- Homepage
- Source Code
- Documentation
- Bug Tracker

## Project description

pypi v2.5 | python 3.6 | 3.7 | 3.8 | build passing | build passing | codecov 94%

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

- **Website (including documentation):** https://networkx.github.io
- **Mailing list:** https://groups.google.com/forum/#!forum/networkx-discuss
- **Source:** https://github.com/networkx/networkx
- **Bug reports:** https://github.com/networkx/networkx/issues

## Simple example

Find the shortest path between two nodes in an undirected graph:

# NetworkX: Graph Creation

1. Adding nodes and edges explicitly

2. Importing data from existing files

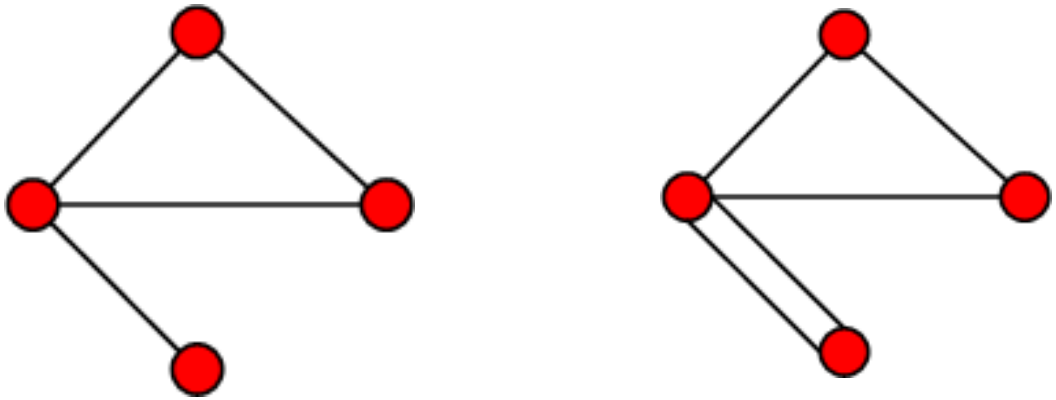
3. Graph generators – standard algorithms to create network topologies

# 1. Adding nodes and edges explicitly

```
>>> import networkx as nx
>>> G=nx.Graph()   # or DiGraph(), MultiGraph(), MultiDiGraph() // see next slide

>>> G.add_node(1)  # or G.remove_node(1)
>>> G.add_nodes_from([2, 3]) #  or G.remove_nodes_from(list)
>>> G.add_nodes_from([x for x in range(10)])

>>> G.add_edge(1,2)  # or G.remove_edge(1, 2)
>>> G.add_edges_from([(1,2),(1,3)])  # G.remove_edges_from(list_of_tuples)
>>> G.add_edges_from([(x,y) for x in range(1, 10) for y in range(10) if  x>y])

>>> G.clear()   # removes all nodes & edges
>>> print(G.number_of_nodes()  # or G.number_of_edges())
>>> print(G.nodes() # or G.edges())
```

# Simple Graphs vs Multigraphs

- **Simple graphs** are graphs where only a single edge can be between any pair of nodes
- **Multigraphs** are graphs where you can have multiple edges between two nodes and loops
- The adjacency matrix for multigraphs can include numbers larger than one, indicating multiple edges between nodes



# 2. Importing data from existing files

- **Read and write NetworkX graphs as adjacency lists**: adjacency list format is useful for graphs without data associated with nodes or edges and for nodes that can be meaningfully represented as strings.
  - The adjacency list format consists of lines with node labels. The first label in a line is the **source node**. Further labels in the line are considered **target nodes** and are added to the graph along with an edge between the source node and target node.
  - The graph with edges a-b, a-c, d-e can be represented as the following adjacency list (anything following the # in a line is a comment):

```
a b c # source target target
d e   # source target
```

# 9.1 Adjacency List

## 9.1.1 Adjacency List

Read and write NetworkX graphs as adjacency lists.

Adjacency list format is useful for graphs without data associated with nodes or edges and for nodes that can be meaningfully represented as strings.

### Format

The adjacency list format consists of lines with node labels. The first label in a line is the source node. Further labels in the line are considered target nodes and are added to the graph along with an edge between the source node and target node.

The graph with edges a-b, a-c, d-e can be represented as the following adjacency list (anything following the # in a line is a comment):

```
a b c # source target target
d e
```

read_adjlist(path[, comments, delimiter, ...]) Read graph in adjacency list format from path

# 2. Importing data from existing files

- **Read and write NetworkX graphs as multi-line adjacency lists**:
  - The first label in a line is the source node followed by its degree d. The next d lines are target nodes and optional edge data. That pattern repeats for all nodes in the graph.
  - The graph with edges a-b, a-c, d-e can be represented as the following adjacency list:

```
a 2
b
c
d 1
e
```

# 9.2 Multiline Adjacency List

## 9.2.1 Multi-line Adjacency List

Read and write NetworkX graphs as multi-line adjacency lists.

The multi-line adjacency list format is useful for graphs with nodes that can be meaningfully represented as strings. With this format simple edge data can be stored but node or graph data is not.

### Format

The first label in a line is the source node label followed by the node degree d. The next d lines are target node labels and optional edge data. That pattern repeats for all nodes in the graph.

The graph with edges a-b, a-c, d-e can be represented as the following adjacency list (anything following the # in a line is a comment):

```
# example.multiline-adjlist
a 2
b
c
d 1
e
```

# 2. Importing data from existing files

- **Read and write NetworkX graphs as edge lists**:
  - With the edge list format simple edge data can be stored but node or graph data is not. There is no way of representing isolated nodes unless the node has a self-loop edge.
  - You can read or write three formats of edge lists:
    - Node pairs with no data:

      1 2
    - Python dictionary as data:

      1 2 {‘weight’:7, ‘color’: ‘green’}
    - Arbitrary data:

      1 2 7 green

# 9.3 Edge List

## 9.3.1 Edge Lists

Read and write NetworkX graphs as edge lists.

The multi-line adjacency list format is useful for graphs with nodes that can be meaningfully represented as strings. With the edgelist format simple edge data can be stored but node or graph data is not. There is no way of representing isolated nodes unless the node has a self-loop edge.

# 2. Importing data from existing files

- **Read and write NetworkX graphs as Python pickles:**
  - "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy."
- For reading/writing files in json or other formats, please see the NetworkX Reference Manual.

# 9.6 Pickle

## 9.6.1 Pickled Graphs

Read and write NetworkX graphs as Python pickles.

"The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy."

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). For arbitrary data types it may be difficult to represent the data as text. In that case using Python pickles to store the graph data can be used.

# 9.8 JSON

## 9.8.1 JSON data

Generate and parse JSON serializable data for NetworkX graphs.

These formats are suitable for use with the d3.js examples https://d3js.org/

The three formats that you can generate with NetworkX are:

- node-link like in the d3.js example https://bl.ocks.org/mbostock/4062045
- tree like in the d3.js example https://bl.ocks.org/mbostock/4063550
- adjacency like in the d3.js example https://bost.ocks.org/mike/miserables/

| | |
|---|---|
| *node_link_data*(G[, attrs]) | Returns data in node-link format that is suitable for JSON serialization and use in Javascript documents. |
| *node_link_graph*(data[, directed, ...]) | Returns graph from node-link data format. |
| *adjacency_data*(G[, attrs]) | Returns data in adjacency format that is suitable for JSON serialization and use in Javascript documents. |
| *adjacency_graph*(data[, directed, ...]) | Returns graph from adjacency data format. |
| *cytoscape_data*(G[, attrs]) | Returns data in Cytoscape JSON format (cyjs). |

# 3. Graph Generators

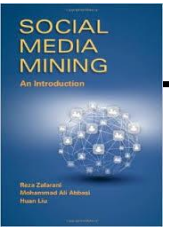
- Graph generators – standard algorithms to create network topologies (See Networkx Reference, 2.5, Chapter 5 **GRAPH GENERATORS**, for the complete list and descriptions)
- Common ones are:
  - **empty_graph**(n): Return the empty graph with n nodes and zero edges.
  - **complete_graph**(n): Return the complete graph K_n with n nodes.
  - **complete_bipartite_graph**(n1, n2): Return the complete bipartite graph K_{n1_n2}.
  - **barbell_graph**(m1, m2): Return the Barbell Graph: two complete graphs connected by a path.
  - **star_graph**(n): Return the Star graph with n+1 nodes: one center node, connected to n outer nodes → **Egocentric Network**
  - grid_graph(dim): Return the n-dimensional grid graph.
  - hypercube_graph(n): Return the n-dimensional hypercube.

# Null Graph vs Empty Graph

- A **null graph** is one where the node set is empty (there are no nodes)
  - Since there are no nodes, there are also no edges (V is the set of vertices in the graph (G); E is the set of edges in G)

$$G(V, E), V = E = \emptyset$$

- An **empty graph** or **edge-less graph** is one where the edge set is empty,



# 3. Graph Generators

- Graph generators – standard algorithms to create network topologies (See Networkx Reference, 2.4, Chapter 5 **GRAPH GENERATORS**, for the complete list and descriptions)
- Common ones are:
  - **empty_graph**(n): Return the empty graph with n nodes and zero edges.
  - **complete_graph**(n): Return the complete graph K_n with n nodes.
  - **complete_bipartite_graph**(n1, n2): Return the complete bipartite graph K_{n1_n2}.
  - **barbell_graph**(m1, m2): Return the Barbell Graph: two complete graphs connected by a path.
  - **star_graph**(n): Return the Star graph with n+1 nodes: one center node, connected to n outer nodes → **Egocentric Network**
  - grid_graph(dim): Return the n-dimensional grid graph.
  - hypercube_graph(n): Return the n-dimensional hypercube.

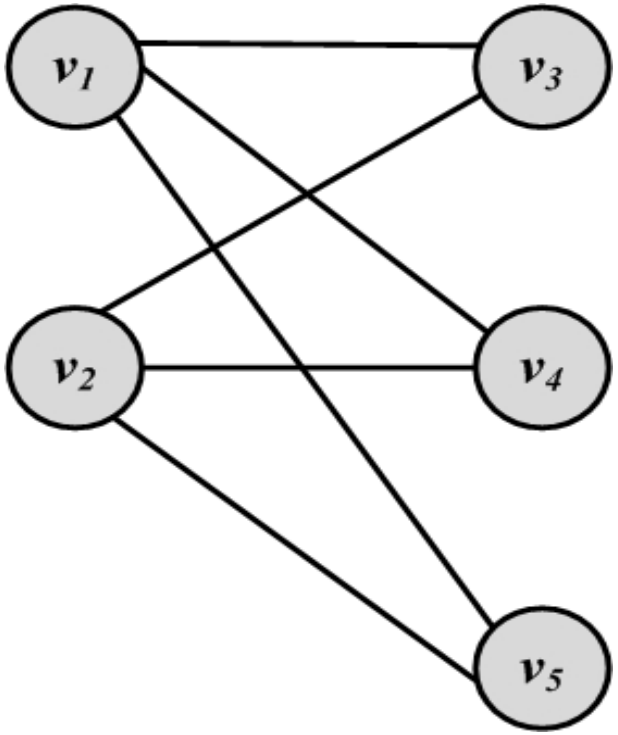
# Bipartite Graph

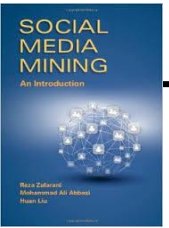
- A **bipartite graph** $G(V, E)$ is a graph where the node set can be partitioned into two sets such that, for all edges, one end-point is in one set and the other end-point is in the other set.

$$V = V_L \cup V_R,$$

$$V_L \cap V_R = \emptyset,$$

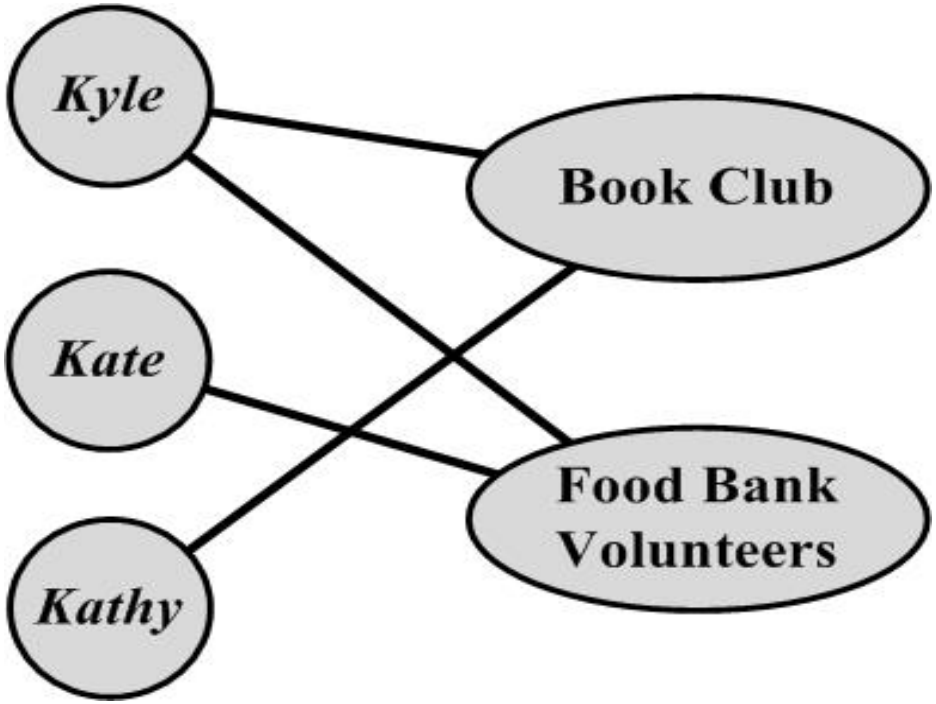
$$E \subset V_L \times V_R.$$

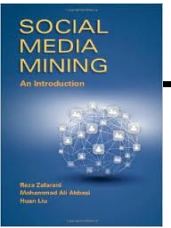


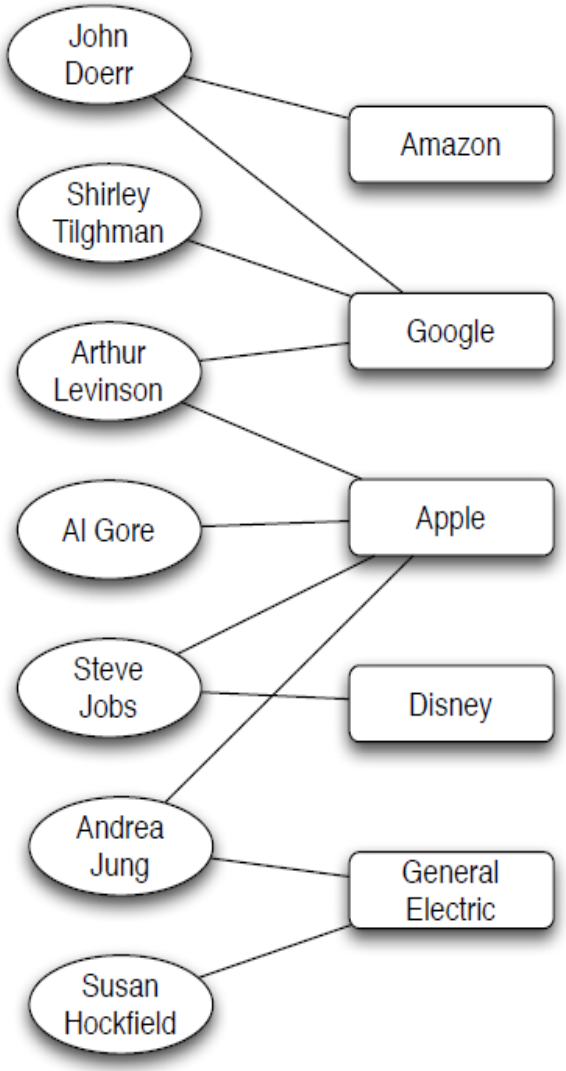



# Affiliation Network

- An **affiliation network** is a bipartite graph. If an individual is associated with an affiliation, an edge connects the corresponding nodes.



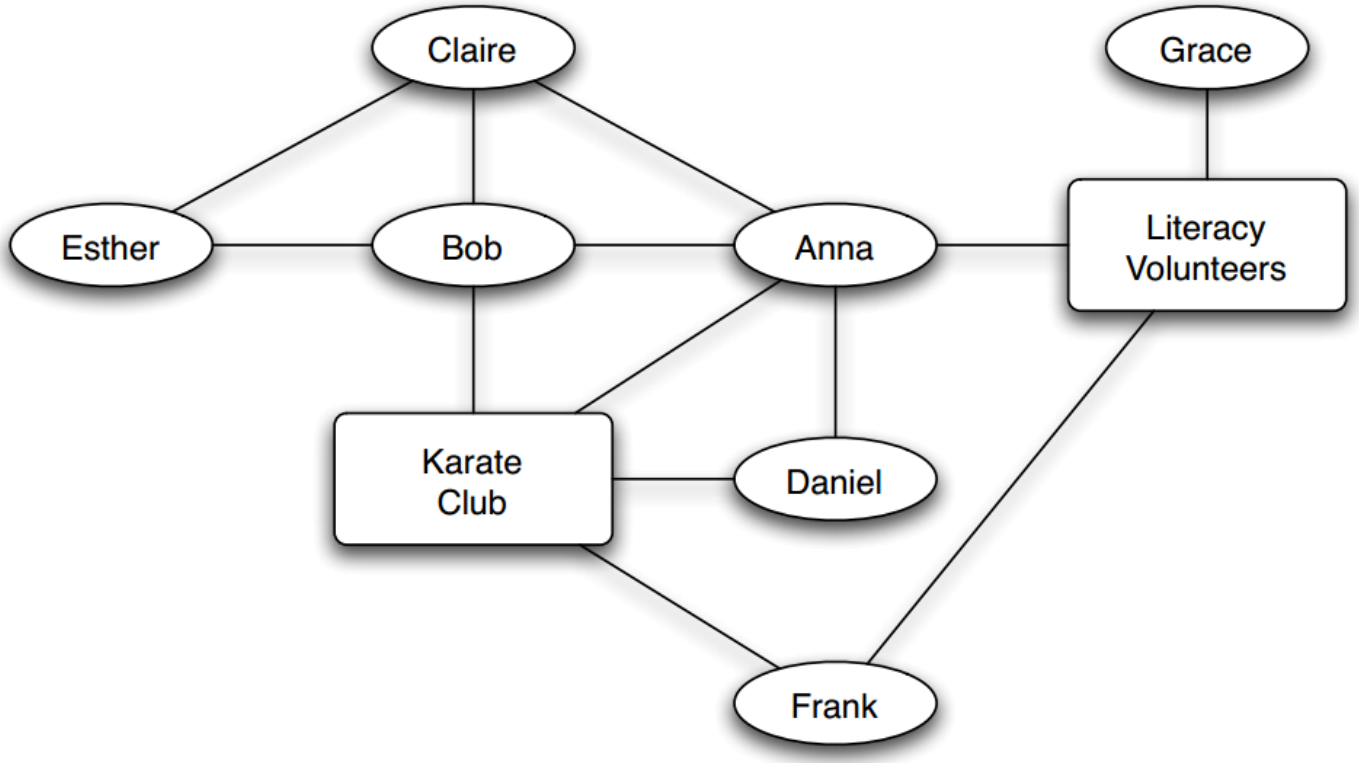


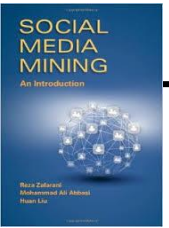

**Networks, Crowds, and Markets: Reasoning About a Highly Connected World**, David Easley & Jon Kleinberg, 2010, Cambridge University Press.

Figure 4.4: One type of affiliation network that has been widely studied is the memberships of people on corporate boards of directors [301]. A very small portion of this network (as of mid-2009) is shown here. The structural pattern of memberships can reveal subtleties in the interactions among both the board members and the companies.

# Social-Affiliation Network

- **Social-Affiliation network** is a combination of a social network and an affiliation network.



# NetworkX: Algorithms

- NetworkX provides many useful algorithms for analyzing graphs (See the Networkx Reference, 2.5, Chapter 3 **ALGORITHMS**, for the complete list and descriptions)
- We will introduce some of them when we discuss related topics (Centrality Analysis next week, Clustering & Community Detection after that, etc.)
- For Assignment #2, you should take a look at the built-in functions listed in **3.23 Distance Measures** & **3.51 Shortest Paths.**
  - There are a couple of functions there that might be useful for your Assignment #2. (**Note:** Locating and learning to use them is part of the exercise.)

# Drawing Graphs (Optional)

```
# Plotting using matplotlib
import matplotlib.pyplot as plt


nx.draw(G)
plt.savefig("mygraph.png")
plt.show()
```

# matplotlib

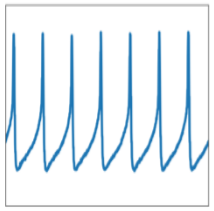
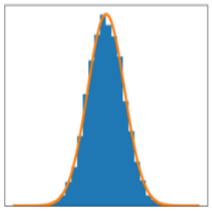
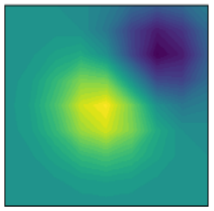
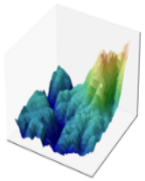
Version 3.3.2

Fork me on GitHub



## Matplotlib: Visualization with Python

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Matplotlib makes easy things easy and hard things possible.

### Create

- Develop publication quality plots with just a few lines of code

### Customize

- Take full control of line styles, font properties, axes properties...

### Extend

- Explore tailored functionality provided by third party packages

**Latest release**
3.3.1: docs | changelog

**Last release for Python 2**
2.2.5: docs | changelog

**Development version**
docs

Support Matplotlib

# Reading Assignments

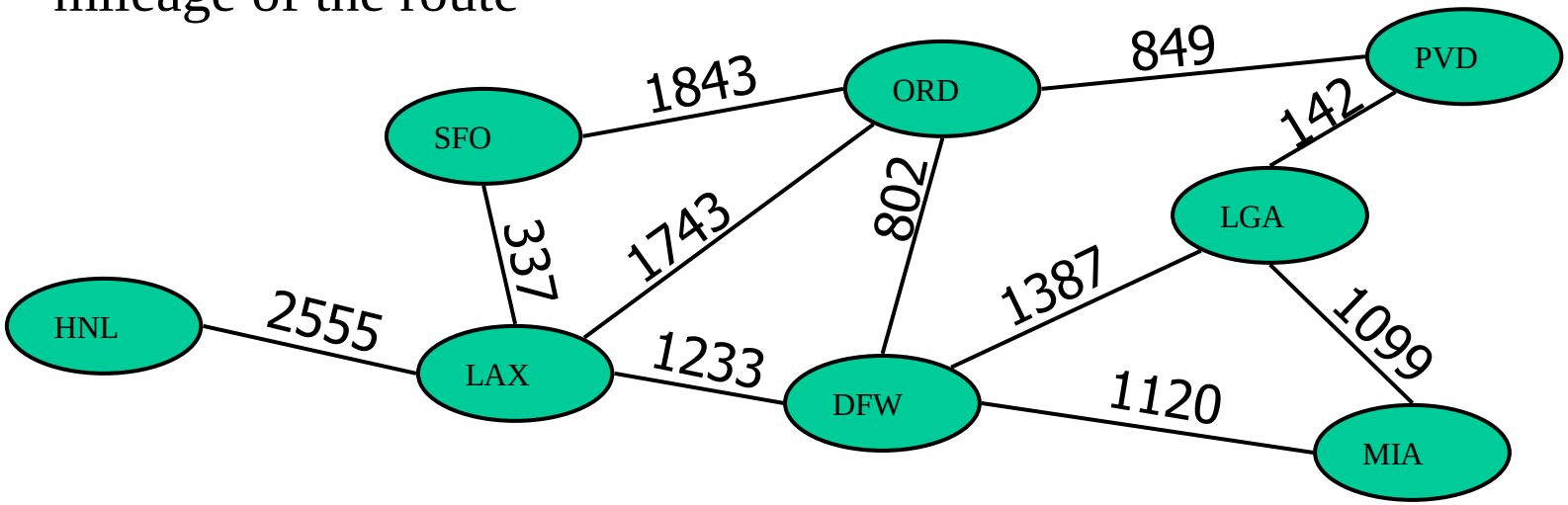
- Textbook #2 : Social Media Mining: An Introduction. By Reza Zafarani, Mohammad Ali Abbasi, and Huan Liu, 2015, Cambridge. (Click its link in Blackboard.) **Chapters 1 & 2**.

- Textbook #4: Community Detection and Mining in Social Media, Tang & Liu, 2010, Morgan and Claypool Publishers. (Click its link in Blackboard.) Chapter 1.

# Supplementary Slides for Graphs From My Data Structure Class (Just FYI)

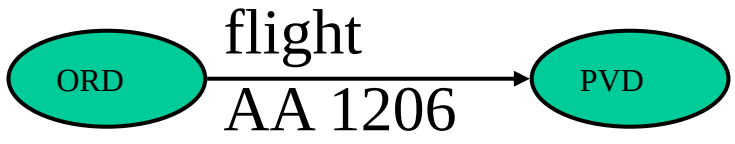
# Graphs

- A graph is a pair (***V, E***), where
  - ***V*** is a set of nodes, called **vertices**
  - ***E*** is a collection of pairs of vertices, called **edges**
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



# Edge Types

- **Directed edge**
  - ordered pair of vertices (***u***,***v***)
  - first vertex ***u*** is the **origin**
  - second vertex ***v*** is the **destination**
  - e.g., a flight

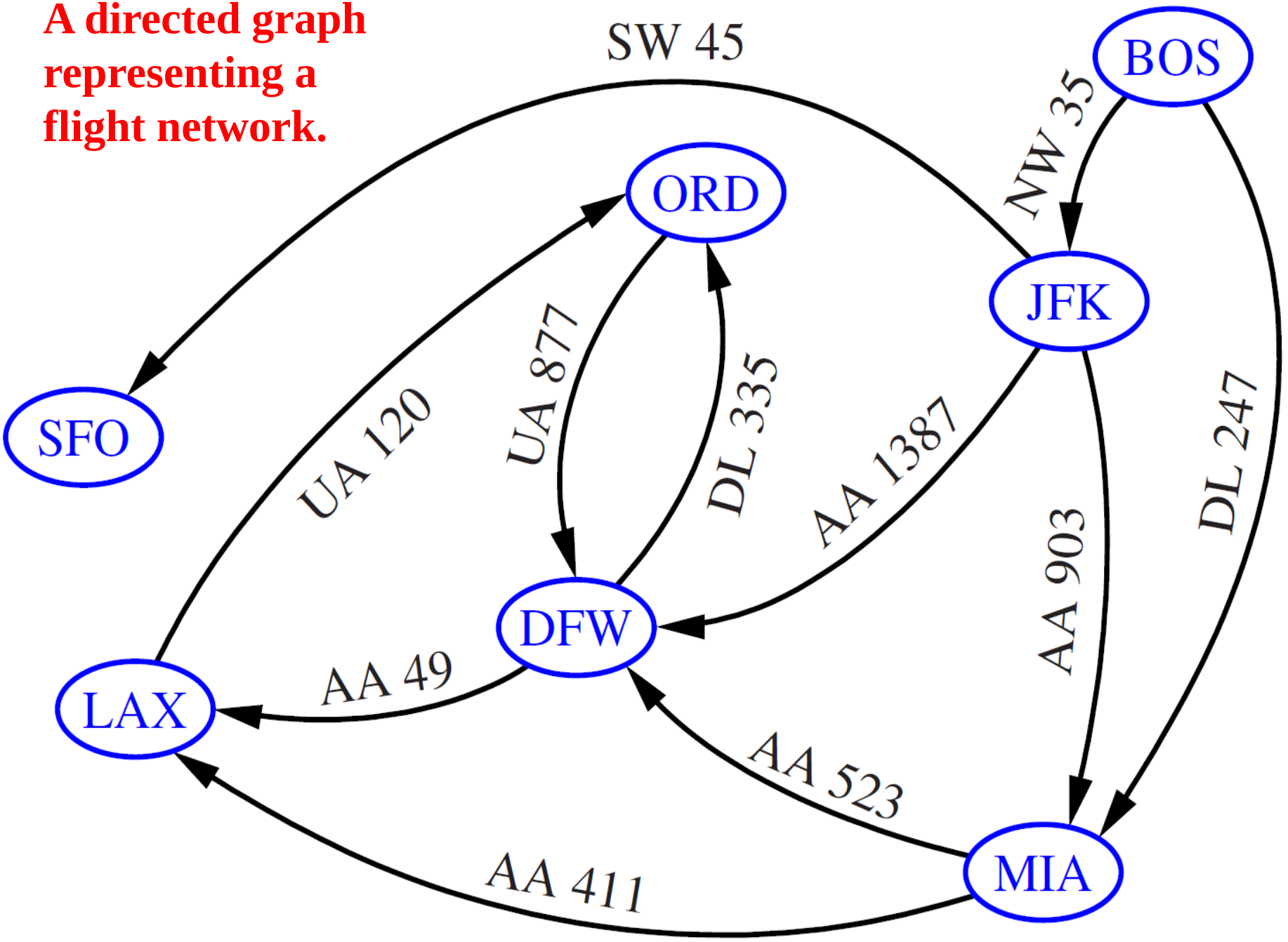


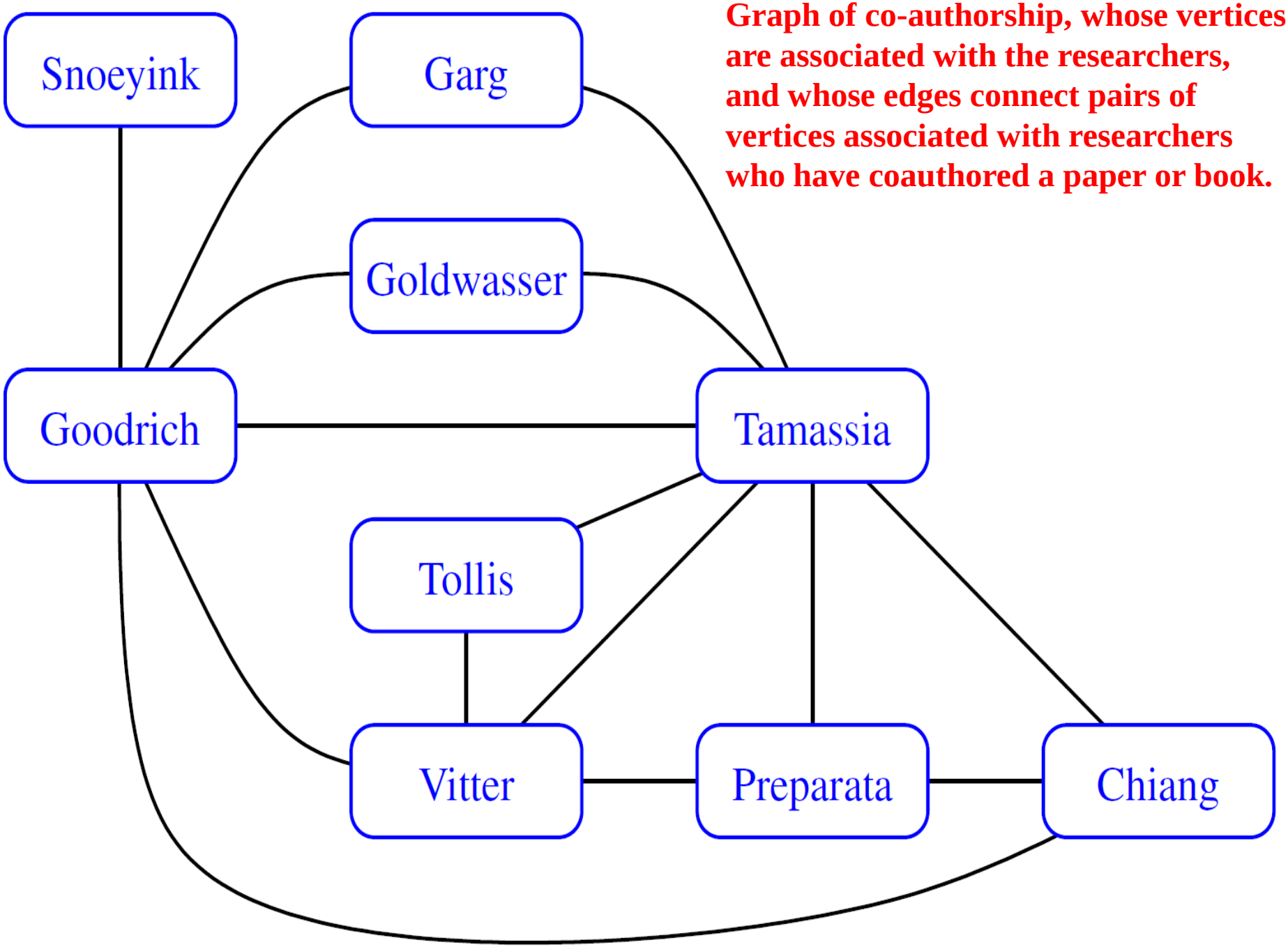

- **Undirected edge**
  - unordered pair of vertices (***u***,***v***)
  - e.g., a flight route


- **Directed graph** (aka **Digraph**)
  - all the edges are directed
  - e.g., flight network
- **Undirected graph**
  - all the edges are undirected
  - e.g., a coauthorship network

A graph that has both directed and undirected edges is often called a **mixed graph**.

**A directed graph representing a flight network.**

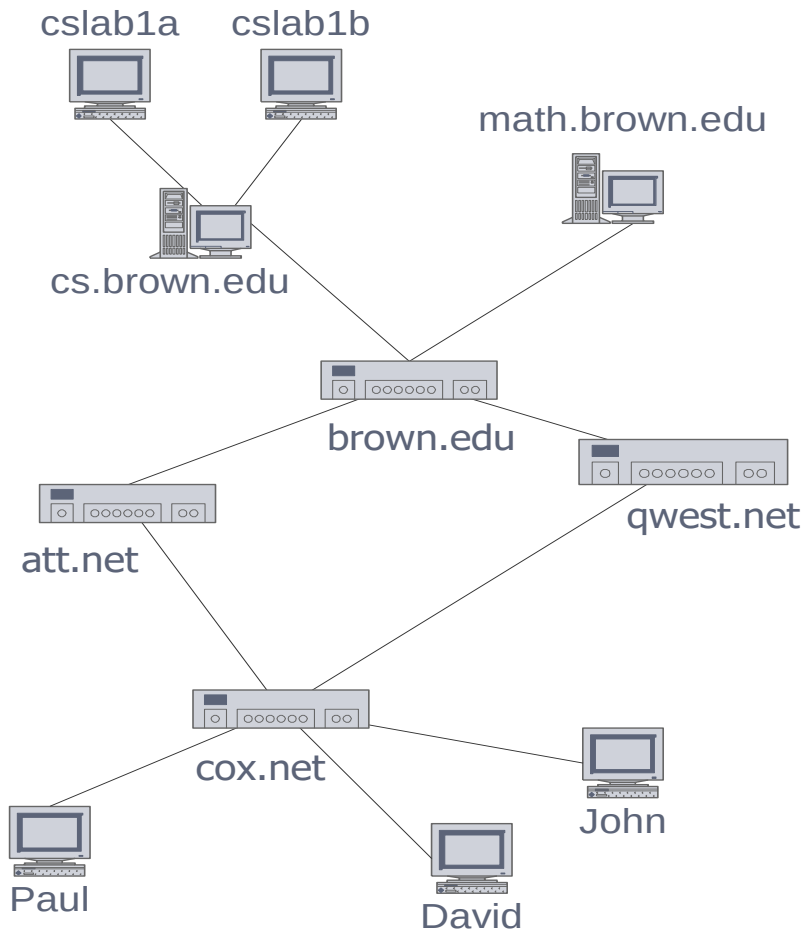





**Graph of co-authorship, whose vertices are associated with the researchers, and whose edges connect pairs of vertices associated with researchers who have coauthored a paper or book.**

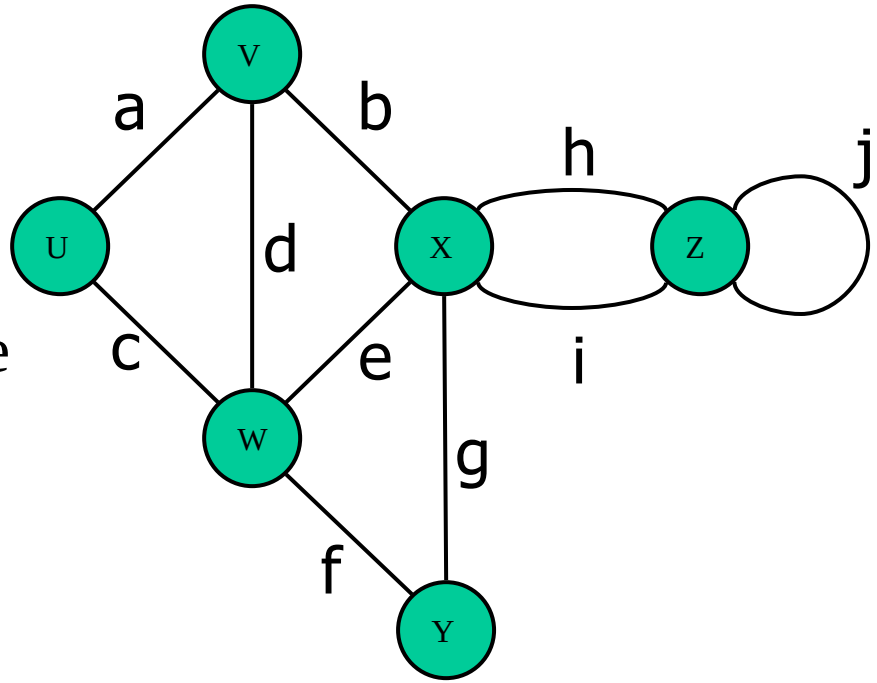
# Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - The Web
- Databases
  - Entity-relationship diagram
- Social Networks



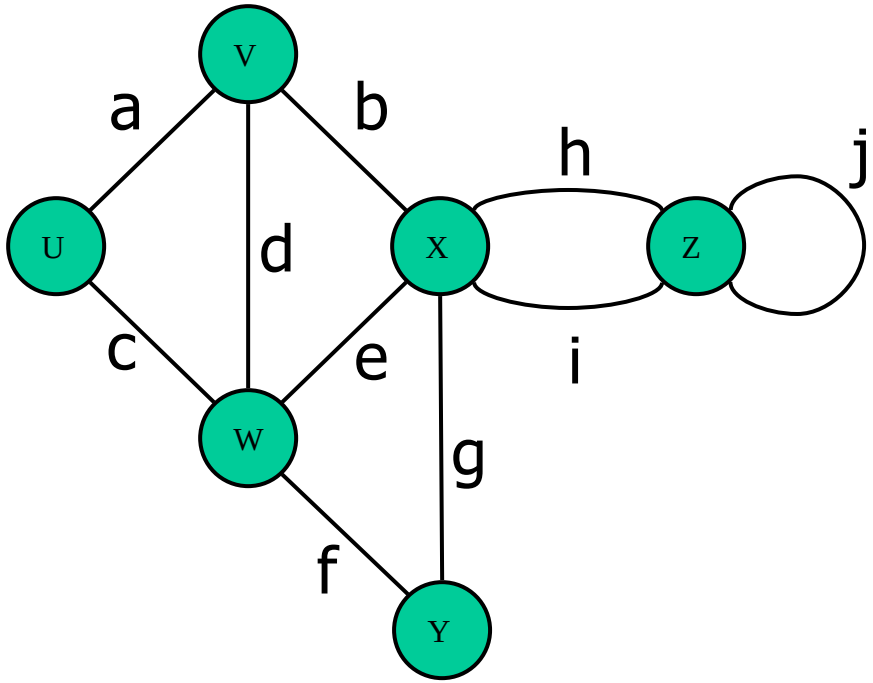
# Terminology

- End vertices (or **endpoints**) of an edge
  - U and V are the endpoints of a
- If an edge is directed, its first endpoint is its **origin** and the other is the **destination** of the edge.
- Edges **incident** on a vertex
  - a, d, and b are incident on V
- The **outgoing** edges of a vertex are the directed edges whose origin is that vertex.
- The **incoming** edges of a vertex are the directed edges whose destination is that vertex.



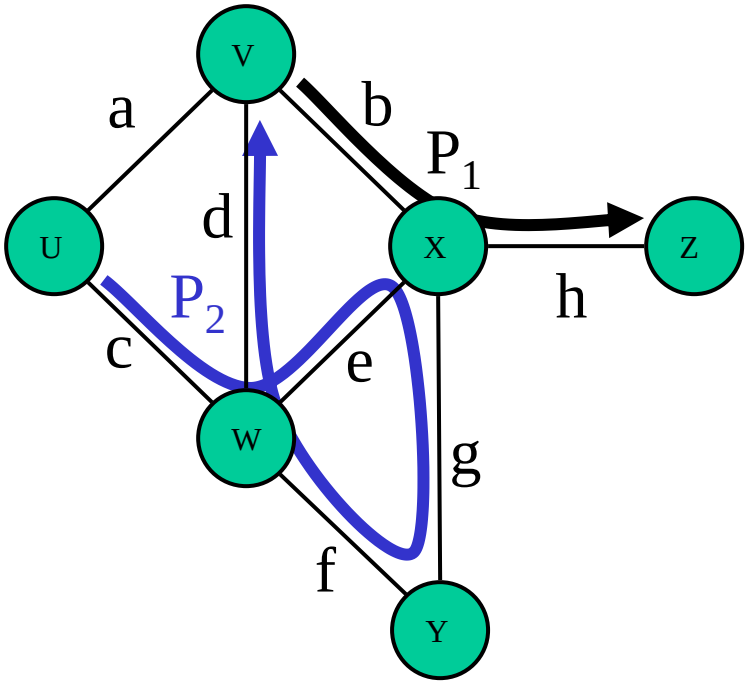
# Terminology

- **Adjacent vertices**
  - U and V are adjacent
- **Degree** of a vertex
  - X has degree 5
- The **in-degree** and **out-degree** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted **indeg(v)** and **outdeg(v)**, respectively.
- **Parallel edges**
  - h and i are parallel edges
- **Self-loop**
  - j is a self-loop



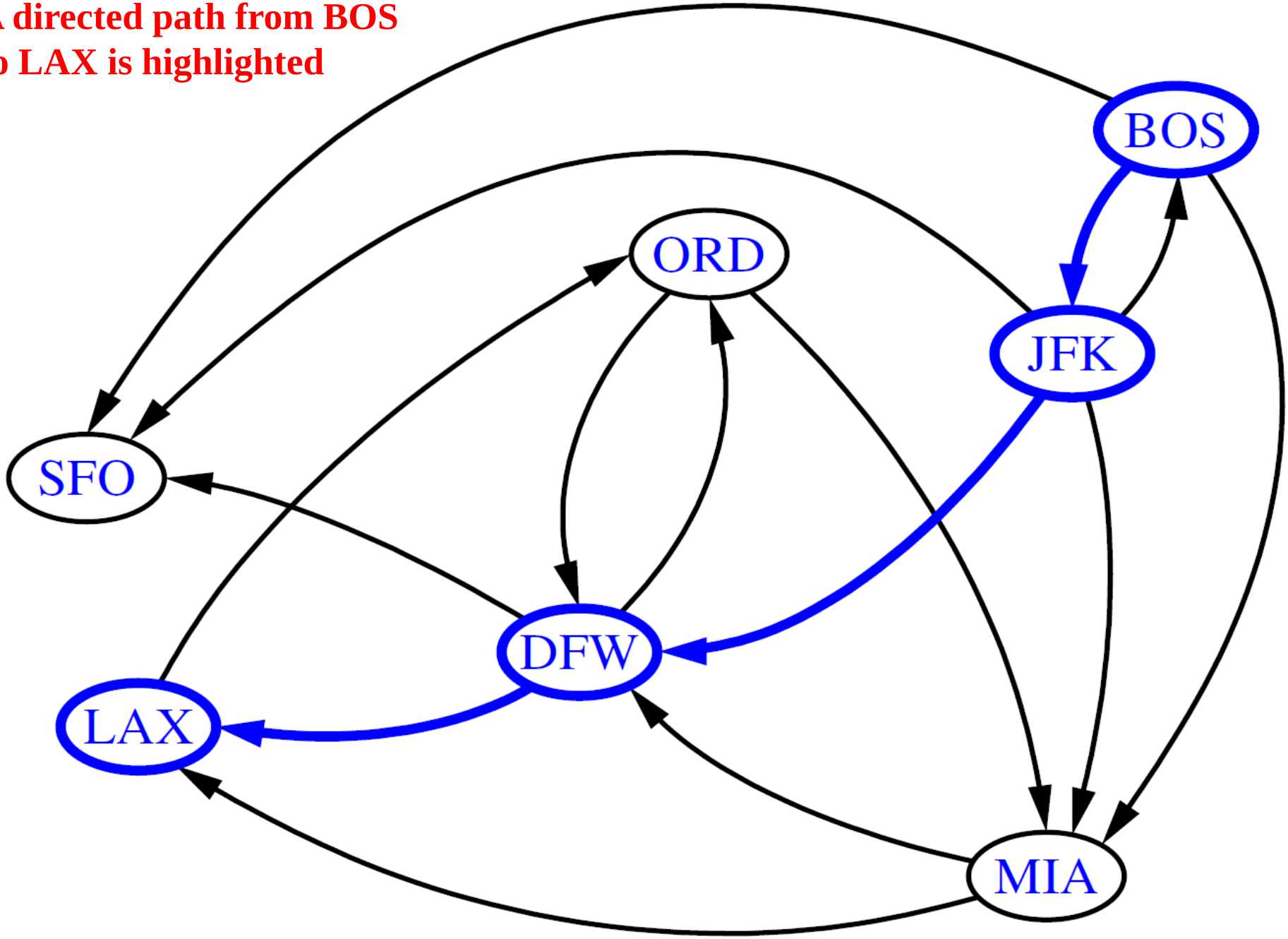
# Terminology

- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- **Simple path**
  - path such that all its vertices and edges are distinct



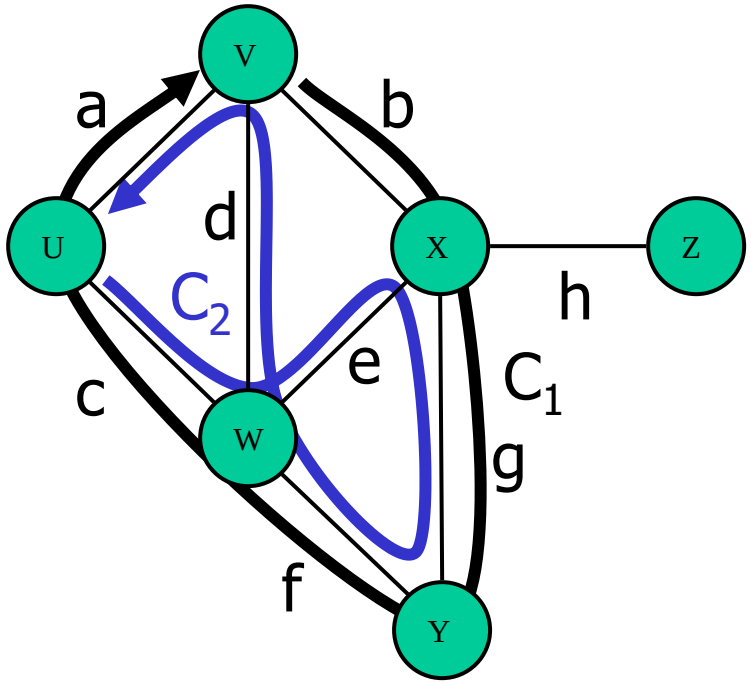

- Examples
  - $P_1$=(V, b, X, h, Z) is a simple path
  - $P_2$=(U,c,**W**,e,X,g,Y,f,**W**,d,V) is a path that is not simple

**A directed path from BOS to LAX is highlighted**



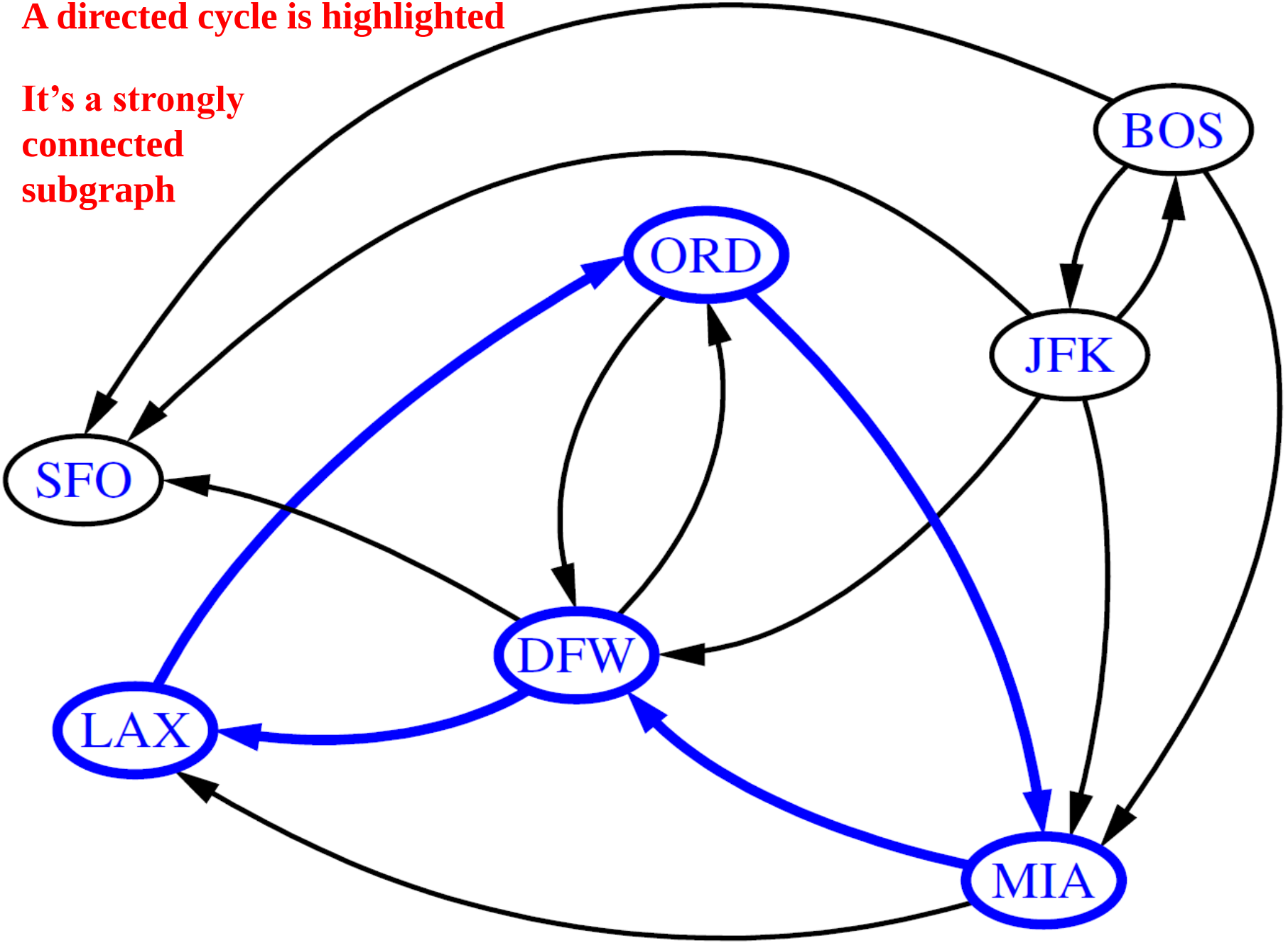
# Terminology

- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- **Simple cycle**
  - cycle such that all its vertices and edges are distinct
- Examples
  - $C_1$=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - $C_2$=(U,c,**W**,e,X,g,Y,f,**W**,d,V,a,↵) is a cycle that is not simple



**A directed cycle is highlighted**

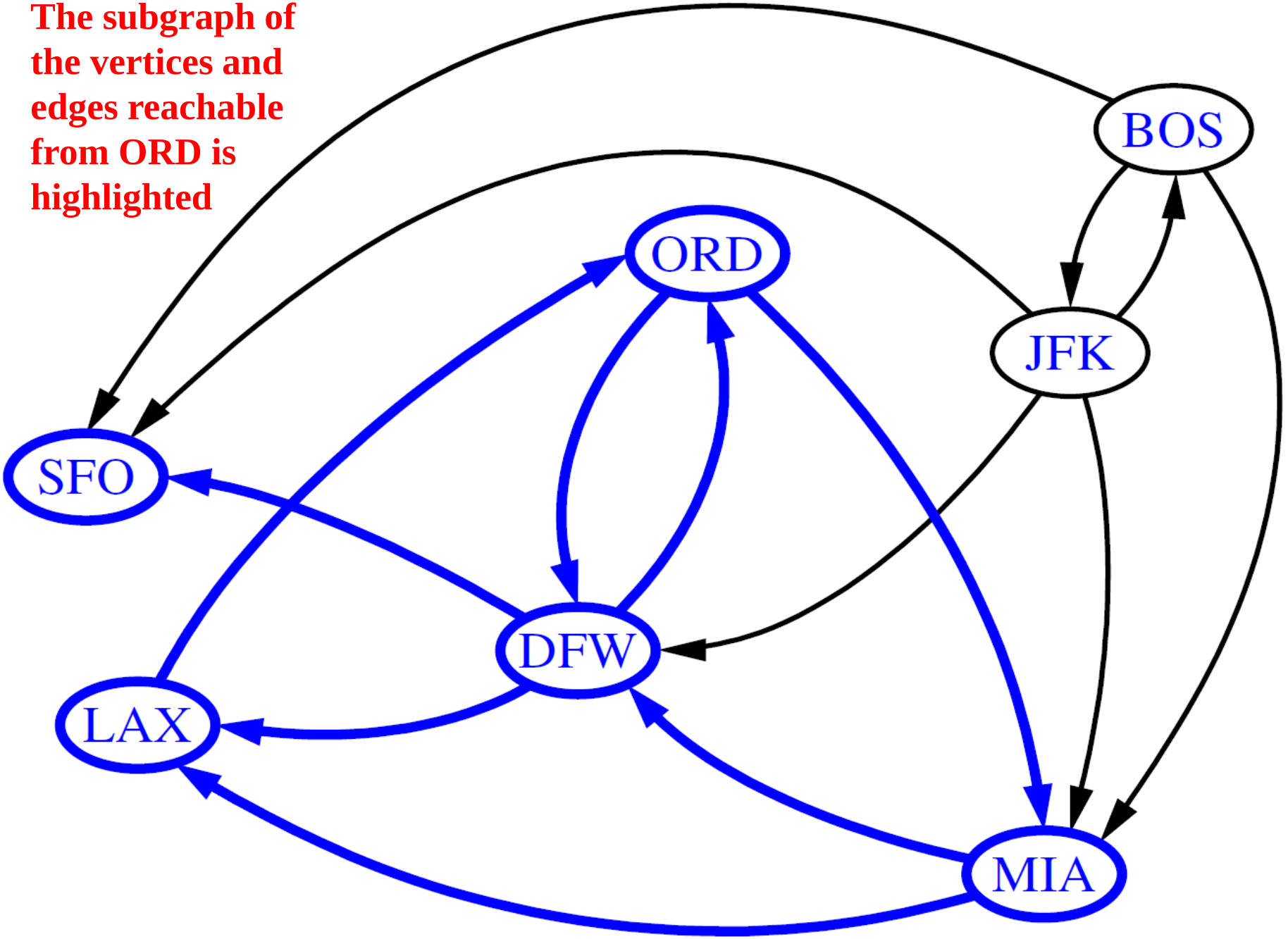
**It's a strongly connected subgraph**

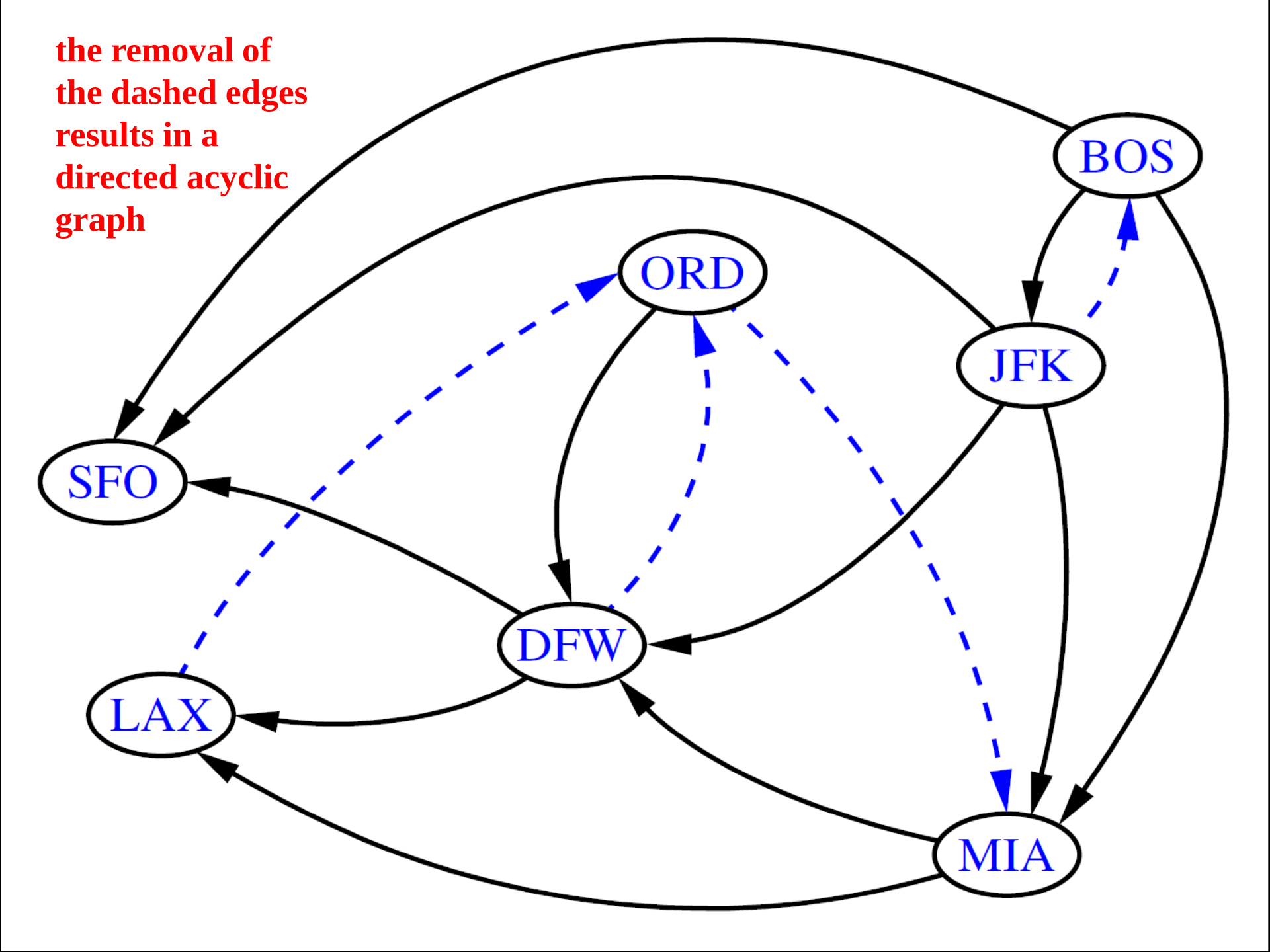


# Terminology

- Given vertices u and v of a (directed) graph G, we say that u **reaches** v, and that v is **reachable** from u, if G has a (directed) path from u to v.
  - In an undirected graph, the notion of reachability is symmetric, that is to say, u reaches v if an only if v reaches u.
  - In a directed graph, it is possible that u reaches v but v does not reach u, because a directed path must be traversed according to the respective directions of the edges. (see slide 5)
- A graph is **connected** if, for any two vertices, there is a path between them.
  - A directed graph G is **strongly connected** if for any two vertices u and v of G, u reaches v and v reaches u.

**The subgraph of the vertices and edges reachable from ORD is highlighted**




the removal of
the dashed edges
results in a
directed acyclic
graph
BOS
ORD
JFK
SFO
DFW
LAX
MIA

# Properties

Property 1

$\sum_v \deg(v) = 2m$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

$m \leq n\,(n - 1)/2$

Proof: each vertex has degree at most $(n - 1)$

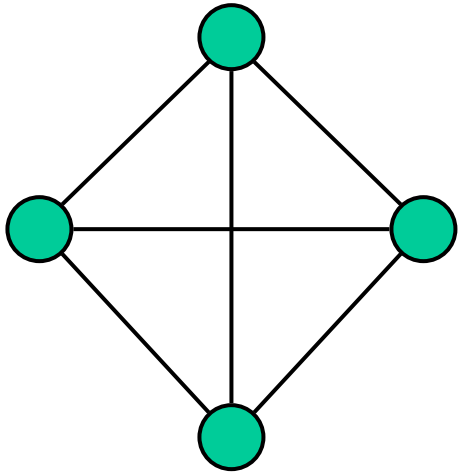
What is the bound for a directed graph?

$m \leq n\,(n - 1)$

Notation

| | |
|---|---|
| $n$ | number of vertices |
| $m$ | number of edges |
| $\deg(v)$ | degree of vertex $v$ |

Example



- $n = 4$
- $m = 6$
- $\deg(v) = 3$

# From My SMM Class

**Theorem 1.** The summation of degrees in an undirected graph is twice the number of edges

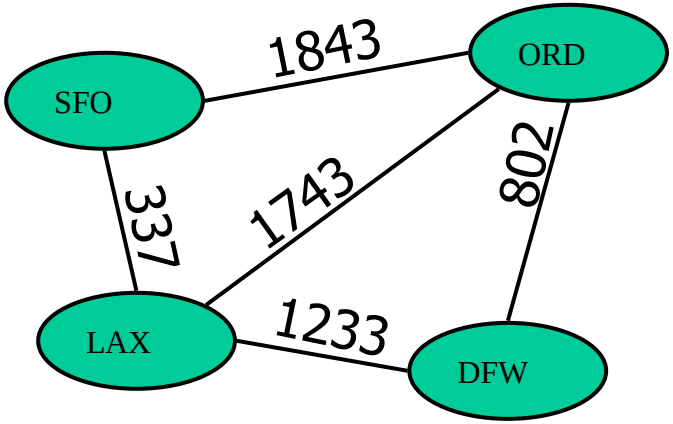
$$\sum_i d_i = 2|E|$$

**Lemma 1.** The number of nodes with odd degree is even

**Lemma 2.** In any directed graph, the summation of in-degrees is equal to the summation of out-degrees,

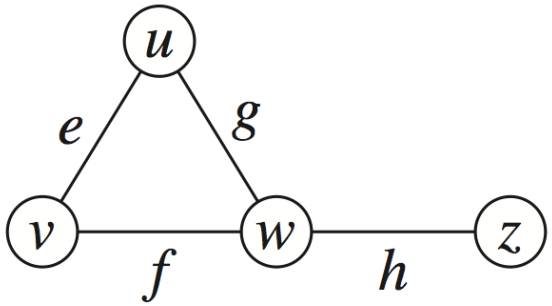
$$\sum_i d_i^{out} = \sum_j d_j^{in} = |E|$$

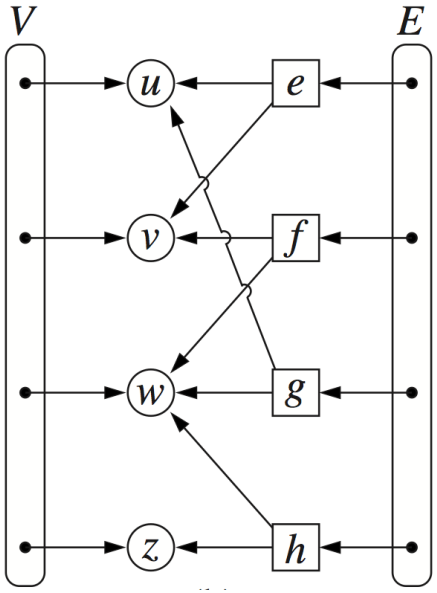
# Data Structures for Graphs



# Edge List Structure

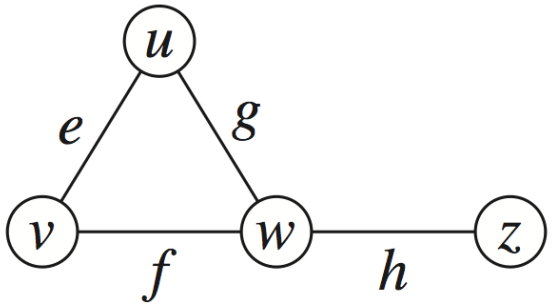
- The edge list structure is possibly the simplest, though not the most efficient, representation of a graph G.
- All vertex objects are stored in an unordered list V, and all edge objects are stored in an unordered list E.
- See the example on the right.

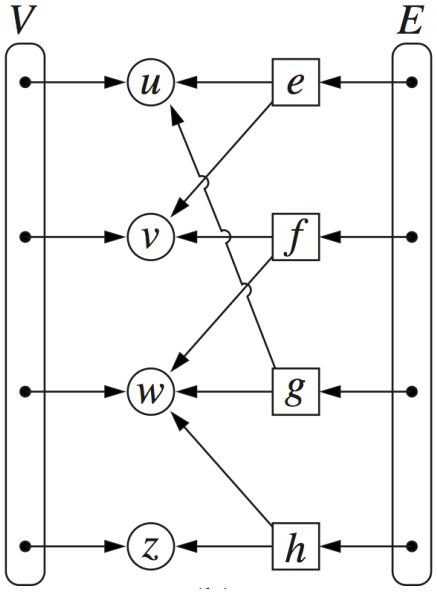


# Edge List Structure

- **Vertex object**
  - element
  - reference to position in vertex sequence
- **Edge object**
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- **Vertex sequence**
  - sequence of vertex objects
- **Edge sequence**
  - sequence of edge objects



# Edge Lists: Performance

- **Space usage**:

  **O(n + m)** for representing a graph with n vertices and m edges. Each individual vertex or edge instance uses O(1) space, and the additional lists V and E use space proportional to their number of entries.

# Edge Lists: Performance

- **Running time**:
  - By querying the respective list V or E, the **numVertices** and **numEdges** methods run in O(1) time.
  - By iterating through the appropriate list, the methods **vertices** and **edges** run respectively in **O(n)** and **O(m)** time.
  - The most significant limitations of an edge list structure, especially when compared to the other graph representations, are the **O(m)** running times of methods **getEdge**(u, v), **outDegree**(v), and **outgoingEdges**(v) (and corresponding methods **inDegree** and **incomingEdges**).
    - The problem is that with all edges of the graph in an unordered list E, the only way to answer those queries is through an exhaustive inspection of all edges.

# Edge Lists: Performance

- **Running time**:
  - It is easy to add a new vertex or a new edge to the graph in **O(1) time**.
    - For example, **insertEdge**(u, v, x)a new edge can be added to the graph by creating an Edge instance storing the given element as data, adding that instance to the positional list E, and recording its resulting Position within E as an attribute of the edge.
    - That stored position can later be used to locate and remove this edge from E in O(1) time, and thus implement the method **removeEdge**(e).

# Edge Lists: Performance

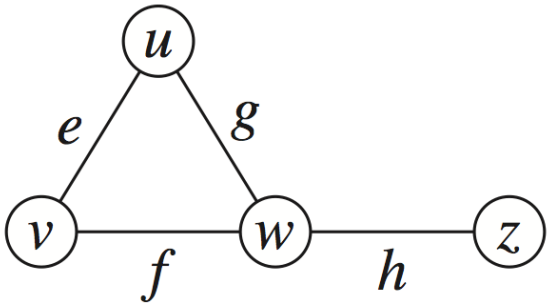
- It is worth discussing why the **removeVertex**(v) method has a running time of **O(m)**.
  - As stated in the graph ADT, when a vertex v is removed from the graph, all edges incident to v must also be removed (otherwise, we would have a contradiction of edges that refer to vertices that are not part of the graph).
  - To locate the incident edges to the vertex, we must examine all edges of E.

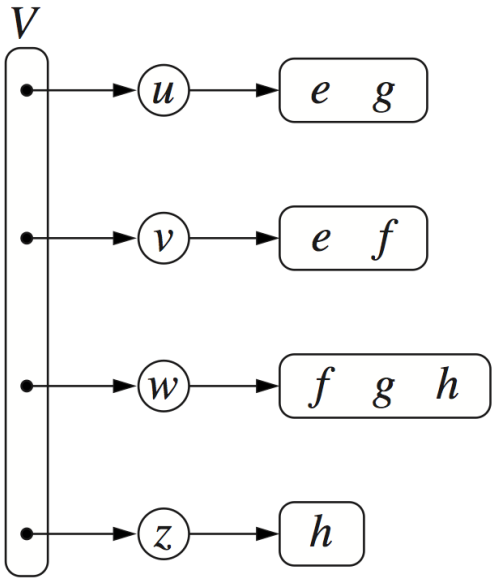
| Method | Running Time |
|---|---|
| numVertices( ), numEdges( ) | $O(1)$ |
| vertices( ) | $O(n)$ |
| edges( ) | $O(m)$ |
| getEdge($u$, $v$), outDegree($v$), outgoingEdges($v$) | $O(m)$ |
| insertVertex($x$), insertEdge($u$, $v$, $x$), removeEdge($e$) | $O(1)$ |
| removeVertex($v$) | $O(m)$ |

Table 14.2: Running times of the methods of a graph implemented with the edge list structure. The space used is $O(n+m)$, where $n$ is the number of vertices and $m$ is the number of edges.

# Adjacency List Structure

- The adjacency list structure for a graph adds extra information to the edge list structure that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex.
- Specifically, for each vertex v, we maintain a collection **I(v)**, called the **incidence collection** of v, whose entries are edges incident to v.
- Although not shown, we presume that each edge of the graph is represented with a unique Edge instance that maintains references to its endpoint vertices.



# Adjacency List Structure

- In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, **Iout(v)** and **Iin(v)**.

- Traditionally, the incidence collection **I(v)** for a vertex v is a **list**, which is why we call this way of representing a graph the **adjacency list** structure.

# Adjacency List Structure

- We require that the primary structure for an adjacency list maintain the collection V of vertices in a way so that we can locate the secondary structure I(v) for a given vertex v in **O(1)** time.
  - This could be done by using a positional list to represent V, with each Vertex instance maintaining a direct reference to its I(v) incidence collection;
    - The adjacency list structure on slide 28 is such an example.
  - If vertices can be uniquely numbered from 0 to n − 1, we could instead use a primary array-based structure to access the appropriate secondary lists.

# Adjacency List Structure

- The primary benefit of an adjacency list is that the collection I(v) (or more specifically, **Iout(v)**) contains exactly those edges that should be reported by the method **outgoingEdges**(v) (in the Graph ADT, slide 19)
  - Therefore, we can implement this method by iterating the edges of I(v) in ***O(deg(v))*** time, where deg(v) is the degree of vertex v.
  - This is the best possible outcome for any graph representation, because there are deg(v) edges to be reported.

# Adjacency Lists: Performance

- Asymptotically, the space requirements for an adjacency list are the same as an edge list structure, using ***O(n + m)*** space for a graph with n vertices and m edges.
- It is clear that the primary lists of vertices and edges use O(n + m) space.
- In addition, the sum of the lengths of all secondary lists is ***O(m)***. (See slide 17 again)

# Adjacency Lists: Performance

- To locate a specific edge for implementing **getEdge(u, v)**, we can search through either I(u) and I(v) (or for a directed graph, either Iout(u) or Iin(v)).
  - By choosing the smaller of the two, we get **O(min(deg(u),deg(v)))** running time.

# Adjacency Lists: Performance

- To efficiently support deletions of edges, an edge (u,v) would need to maintain a reference to its positions within both I(u) and I(v), so that it could be deleted from those collections in **O(1)** time.
- To remove a vertex v, we must also remove any incident edges, but at least we can locate those edges in **O(deg(v))** time

| Method | Running Time |
|---|---|
| numVertices( ), numEdges( ) | $O(1)$ |
| vertices( ) | $O(n)$ |
| edges( ) | $O(m)$ |
| getEdge($u$, $v$) | $O(\min(\deg(u), \deg(v)))$ |
| outDegree($v$), inDegree($v$) | $O(1)$ |
| outgoingEdges($v$), incomingEdges($v$) | $O(\deg(v))$ |
| insertVertex($x$), insertEdge($u$, $v$, $x$) | $O(1)$ |
| removeEdge($e$) | $O(1)$ |
| removeVertex($v$) | $O(\deg(v))$ |

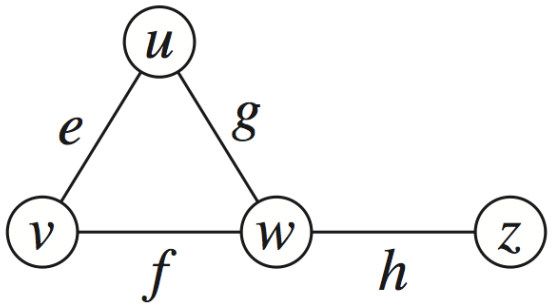
**Table 14.3:** Running times of the methods of a graph implemented with the adjacency list structure. The space used is $O(n+m)$, where $n$ is the number of vertices and $m$ is the number of edges.

# Adjacency Maps

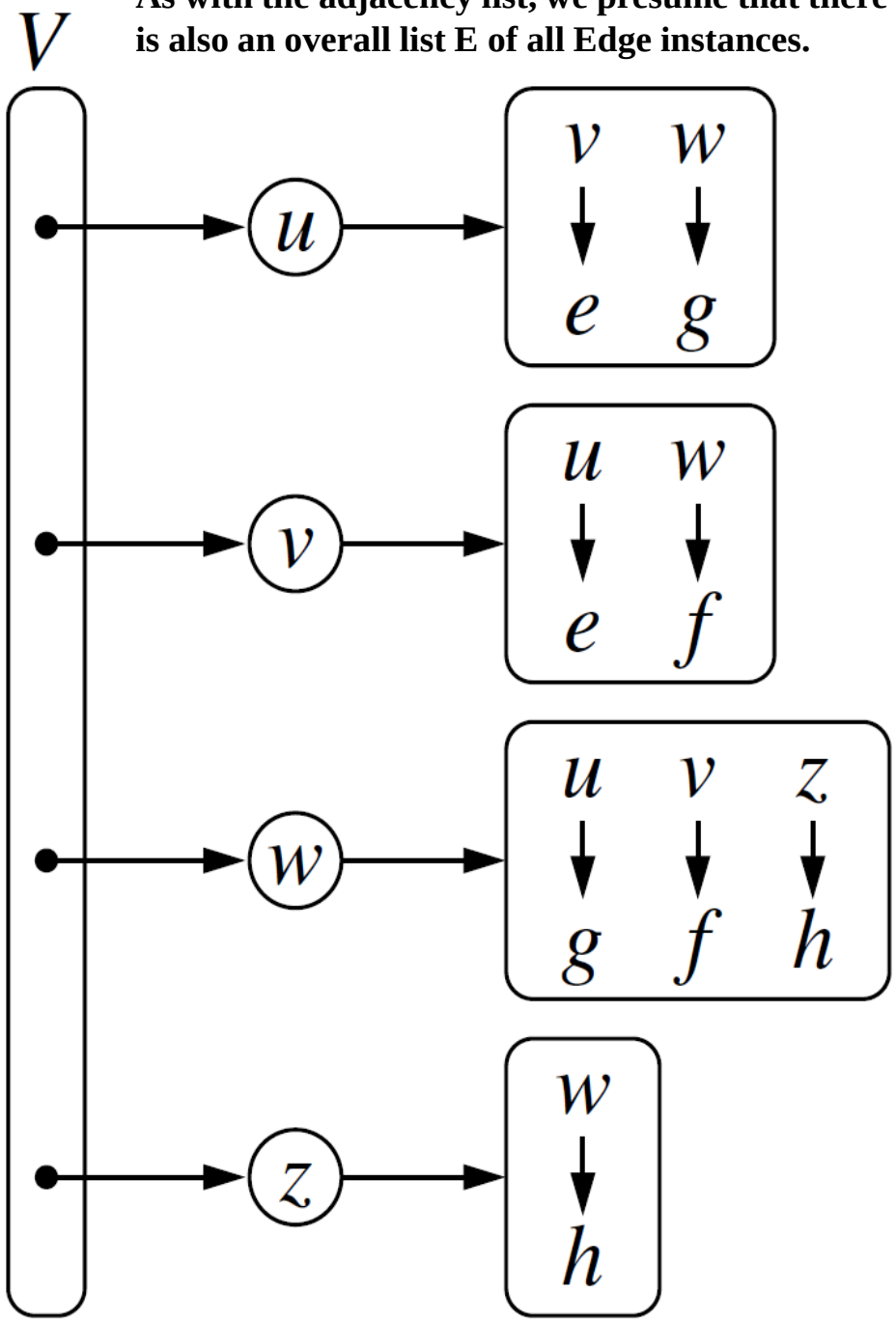
- In the adjacency list structure, we assume that the secondary incidence collections are implemented as unordered linked lists.
  - Therefore, **getEdge(u, v)** requires **O(min(deg(u),deg(v)))** time, because we must search through either I(u) or I(v). (See slide 33 again)

# Adjacency Maps

- We can improve the performance by using a **hash table** to implement I(v) for each vertex v.
- Specifically, we let the opposite endpoint of each incident edge serve as a key in the map, with the edge structure serving as the value. (See next slide)
- We call such a graph representation an **adjacency map**.
- The space usage for an adjacency map remains O(n + m), because I(v) uses **O(deg(v))** space for each vertex v, as with the adjacency list.




**As with the adjacency list, we presume that there is also an overall list E of all Edge instances.**



# Adjacency Maps

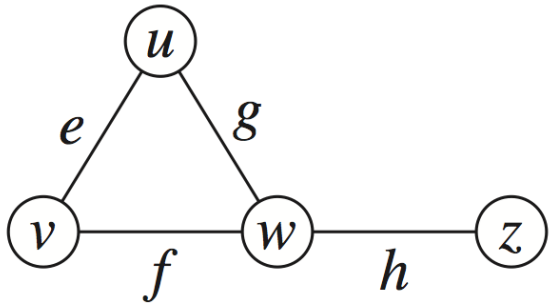
- The advantage of the adjacency map, relative to an adjacency list, is that the **getEdge(u, v)** method can be implemented in expected O(1) time by searching for vertex u as a key in I(v), or vice versa.
  - This provides a likely improvement over the adjacency list, while retaining the worst-case bound of O(min(deg(u),deg(v))).
- In comparing the performance of adjacency map to other representations, we find that it essentially achieves optimal running times for all methods, making it an excellent all-purpose choice as a graph representation. (See next slide)

| Method | Edge List | Adj. List | Adj. Map | Adj. Matrix |
|---|---|---|---|---|
| numVertices( ) | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| numEdges( ) | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| vertices( ) | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| edges( ) | $O(m)$ | $O(m)$ | $O(m)$ | $O(m)$ |
| getEdge($u$, $v$) | $O(m)$ | $O(\min(d_u, d_v))$ | $O(1)$ exp. | $O(1)$ |
| outDegree($v$)<br>inDegree($v$) | $O(m)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| outgoingEdges($v$)<br>incomingEdges($v$) | $O(m)$ | $O(d_v)$ | $O(d_v)$ | $O(n)$ |
| insertVertex($x$) | $O(1)$ | $O(1)$ | $O(1)$ | $O(n^2)$ |
| removeVertex($v$) | $O(m)$ | $O(d_v)$ | $O(d_v)$ | $O(n^2)$ |
| insertEdge($u$, $v$, $x$) | $O(1)$ | $O(1)$ | $O(1)$ exp. | $O(1)$ |
| removeEdge($e$) | $O(1)$ | $O(1)$ | $O(1)$ exp. | $O(1)$ |

**Table 14.1:** A summary of the running times for the methods of the graph ADT, using the graph representations discussed in this section. We let $n$ denote the number of vertices, $m$ the number of edges, and $d_v$ the degree of vertex $v$. Note that the adjacency matrix uses $O(n^2)$ space, while all other structures use $O(n+m)$ space.

# Adjacency Matrix Structure

- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge




| | | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| u | 0 | | e | g | |
| v | 1 | e | | f | |
| w | 2 | g | f | | h |
| z | 3 | | | h | |

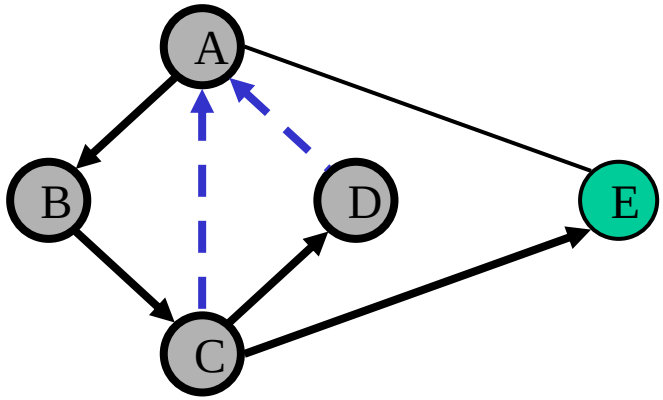
# Adjacency Matrix Structure

- The most significant advantage of an adjacency matrix is that any **edge(u,v)** can be accessed in worst-case **O(1)** time
  - Recall that the adjacency map supports that operation in O(1) expected time.
- However, several operation are less efficient with an adjacency matrix.
  - For example, to find the edges incident to vertex v, we must presumably examine all n entries in the row associated with v (i.e. **O(n)**)
  - Recall that an adjacency list or map can locate those edges in optimal **O(deg(v))** time.
- Adding or removing vertices from a graph is problematic, as the matrix must be resized. (See slide 40 again)

# Adjacency Matrix Structure

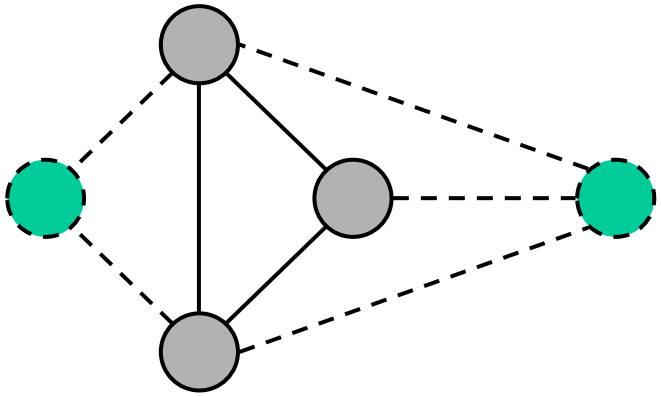
- **Space Usage**
  - The **$O(n^2)$** space usage of an adjacency matrix is typically far worse than the **$O(n + m)$** space required of the other representations.
  - Although, in the worst case, the number of edges in a dense graph will be proportional to $n^2$, most real-world graphs are sparse. In such cases, use of an adjacency matrix is inefficient.
  - However, if a graph is dense, the constants of proportionality of an adjacency matrix can be smaller than that of an adjacency list or map.
    - In fact, if edges do not have auxiliary data, a Boolean adjacency matrix can use one bit per edge slot.

# Depth First Search

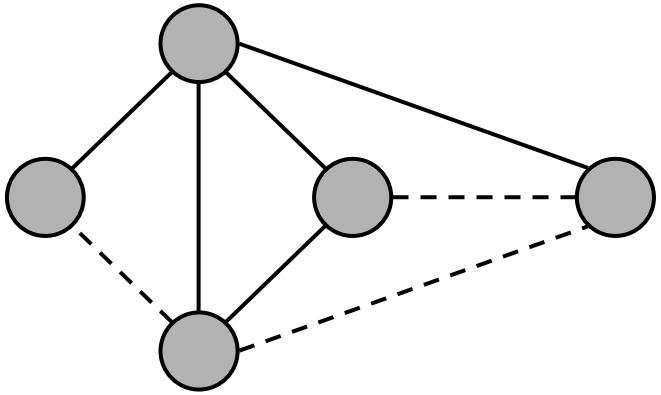


# Subgraphs

- A **subgraph** S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A **spanning subgraph** of G is a subgraph that contains all the vertices of G



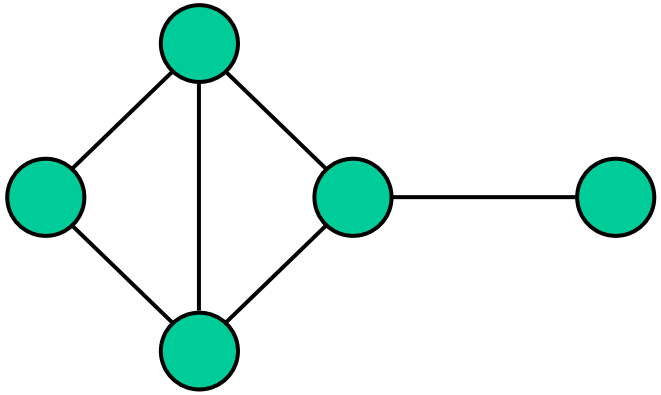
Subgraph



Spanning subgraph

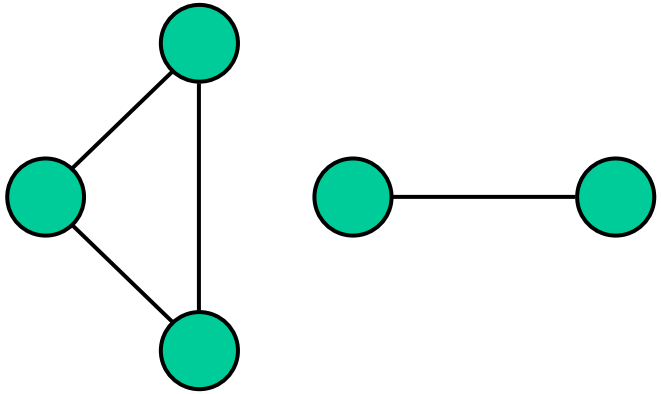
# Connectivity

- A graph is **connected** if there is a path between every pair of vertices



Connected graph

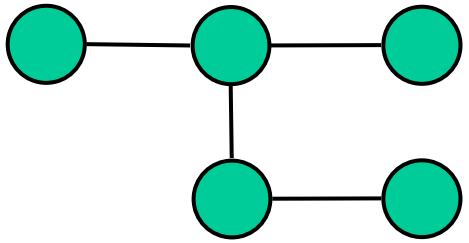
- A connected **component** of a graph G is a maximal connected subgraph of G



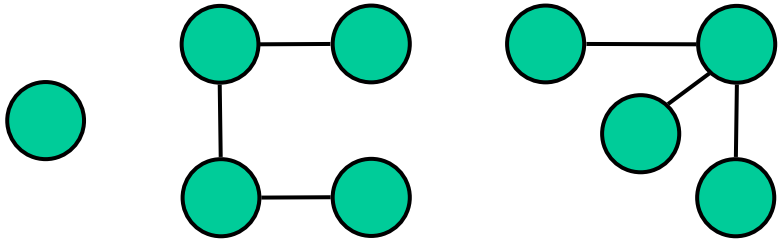
Non connected graph with two connected components

# Trees & Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles
  - This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



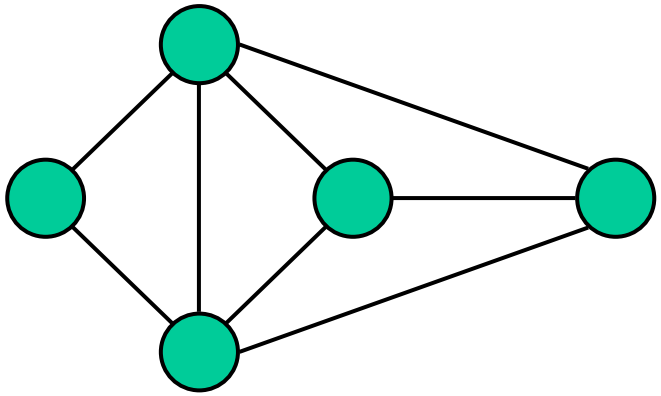
Tree



Forest

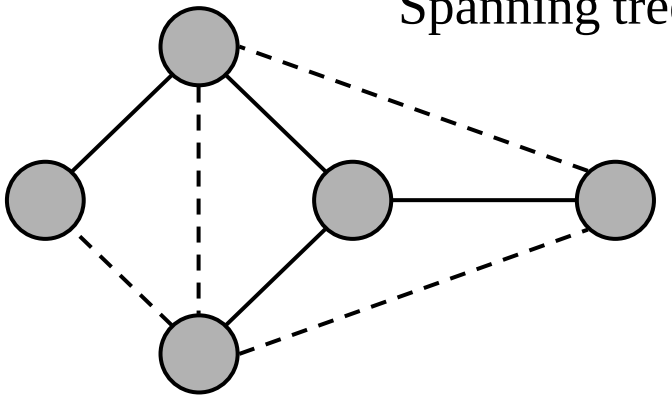
# Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph

Spanning tree



# Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph (next slide)
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find a path between two given vertices
  - Find a cycle in the graph

# Depth-First Search

**Algorithm** DFS($G$, $u$):

*Input:* A graph $G$ and a vertex $u$ of $G$

*Output:* A collection of vertices reachable from $u$, with their discovery edges

Mark vertex $u$ as visited.

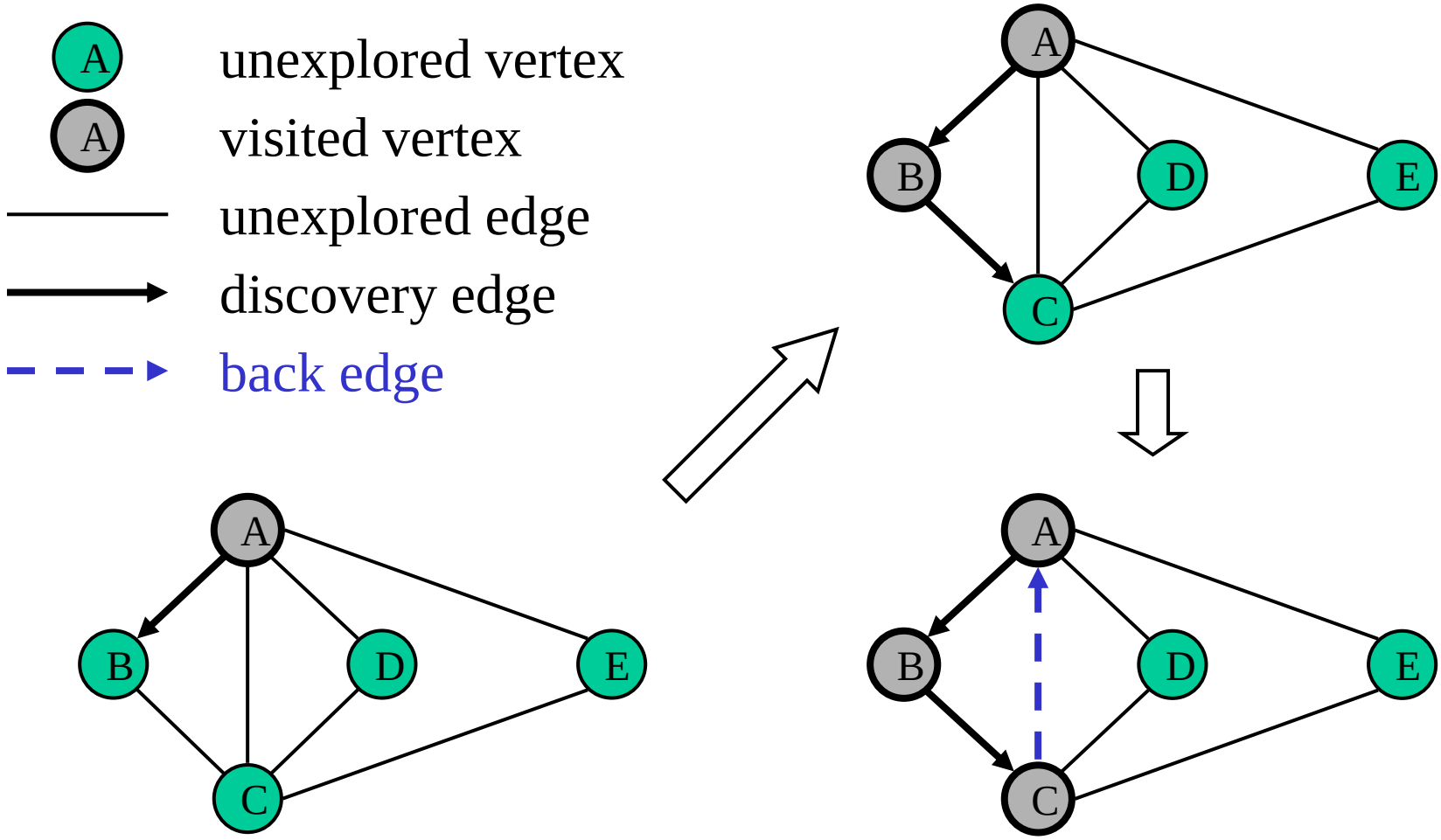
**for** each of $u$'s outgoing edges, $e = (u, v)$ **do**

  **if** vertex $v$ has not been visited **then**

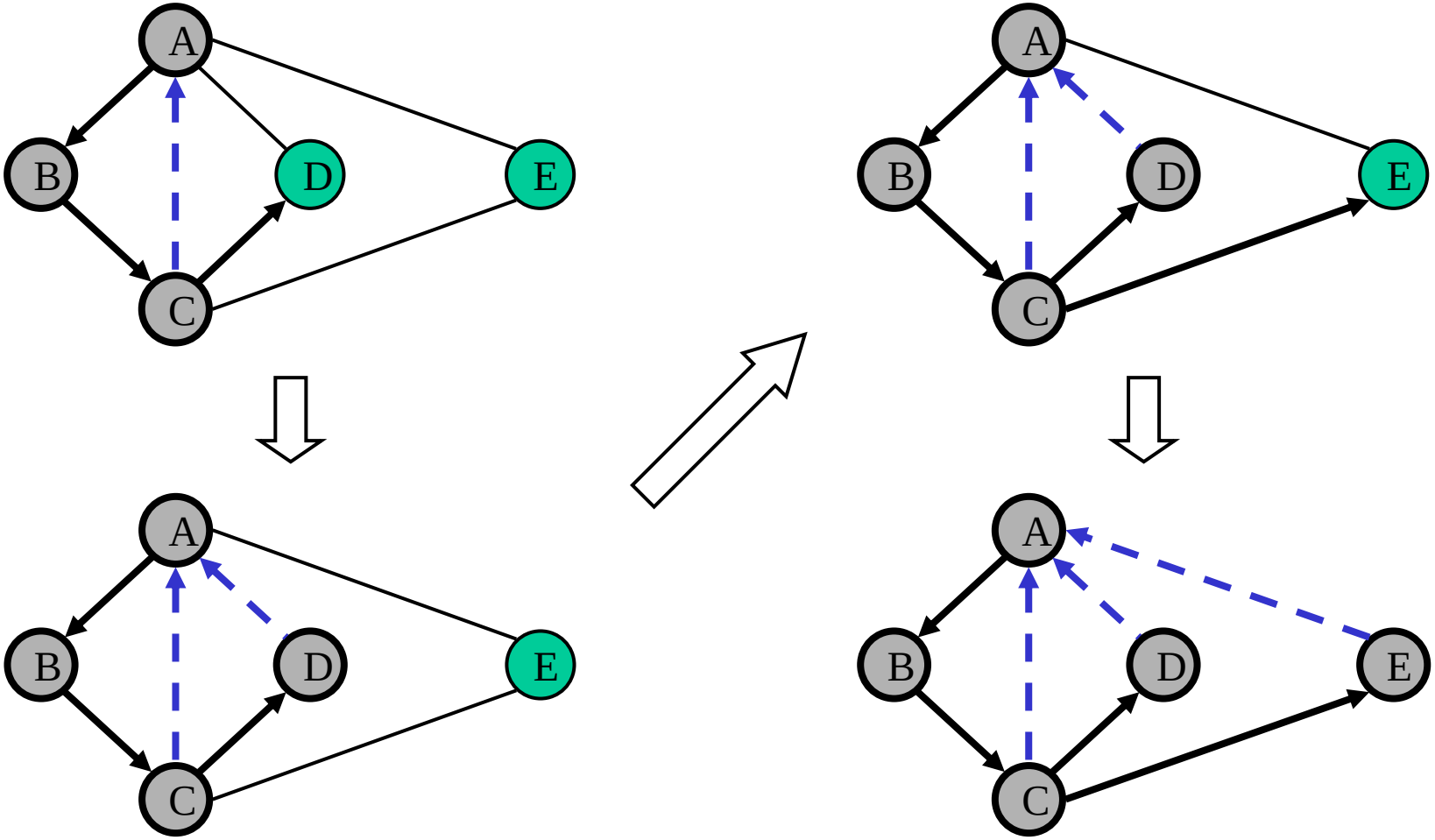
    Record edge $e$ as the discovery edge for vertex $v$.

    Recursively call DFS($G$, $v$).

# Depth-First Search: Example

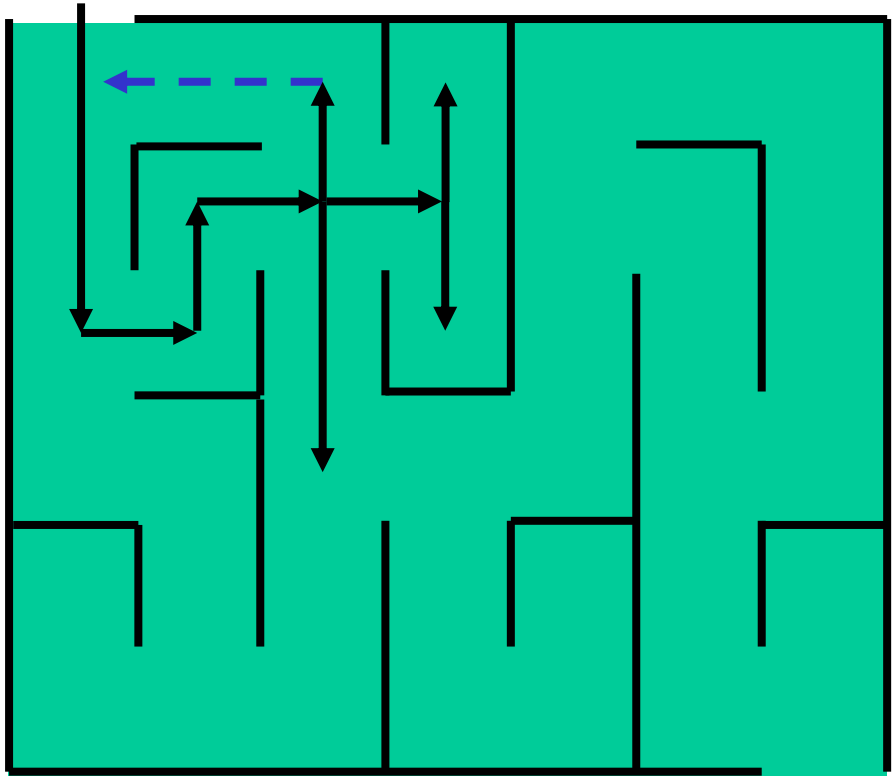


# Depth-First Search: Example



# DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (**start vertex**) by means of a rope (**recursion stack**)



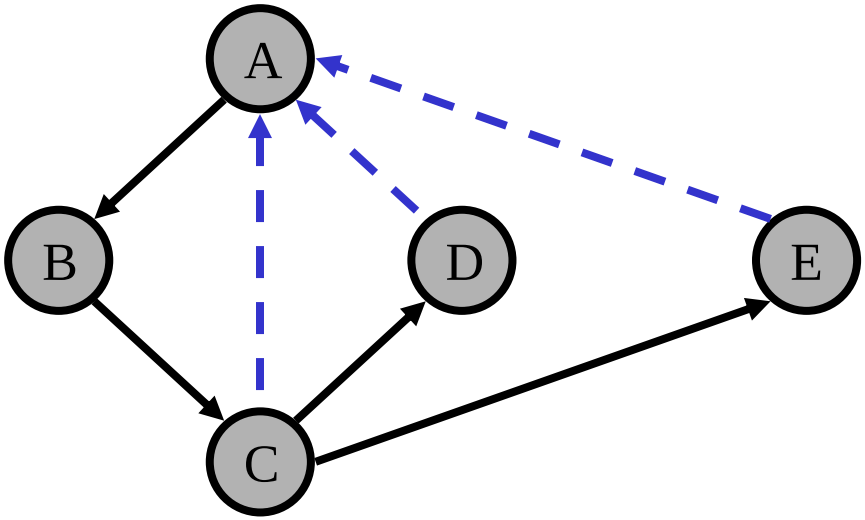
# Properties of DFS

Property 1

***DFS*(*G*, *v*)** visits all the vertices and edges in the connected component of ***v***

Property 2

The discovery edges labeled by ***DFS*(*G*, *v*)** form a spanning tree of the connected component of ***v***



# Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice: $O(n)$
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice: $O(m)$
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges/**outgoingEdges** is called once for each vertex: $O(m)$
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$

# Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices ***v*** and ***z.***
- We call ***DFS*(*G, v*)** with ***v*** as the start vertex
- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex ***z*** is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```

# Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle.
- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (***v***, ***w***) is encountered, we return the cycle as the portion of the stack from the top to vertex ***w***

```
Algorithm cycleDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                pathDFS(G, w, z)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements()
    S.pop(v)
```

# DFS for an Entire Graph

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

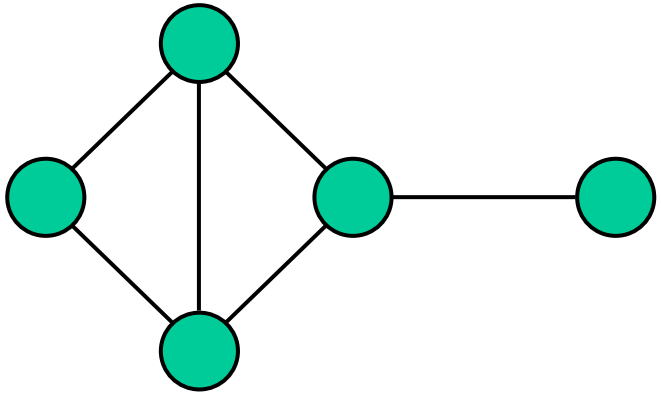
```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

```
Algorithm DFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the edges of G
      in the connected component of v
      as discovery edges and back edges
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

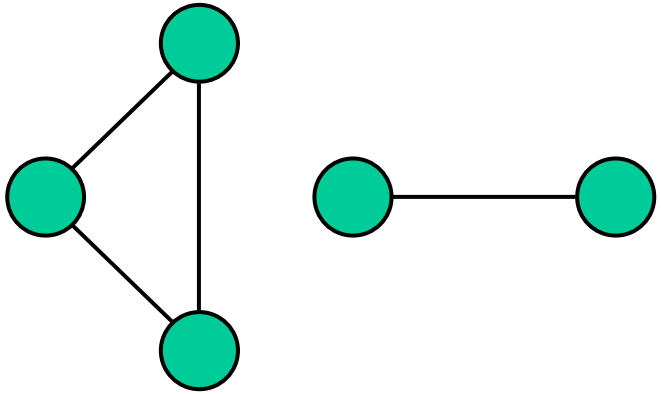
# Connectivity (Revisited)

- A graph is connected if there is a path between every pair of vertices

- A connected component of a graph G is a maximal connected subgraph of G



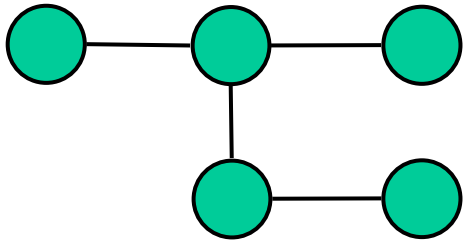
Connected graph



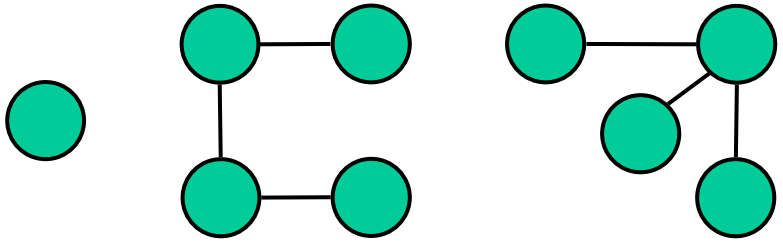
Non connected graph with two connected components

# Trees & Forests (Revisited)

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles
  - This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees

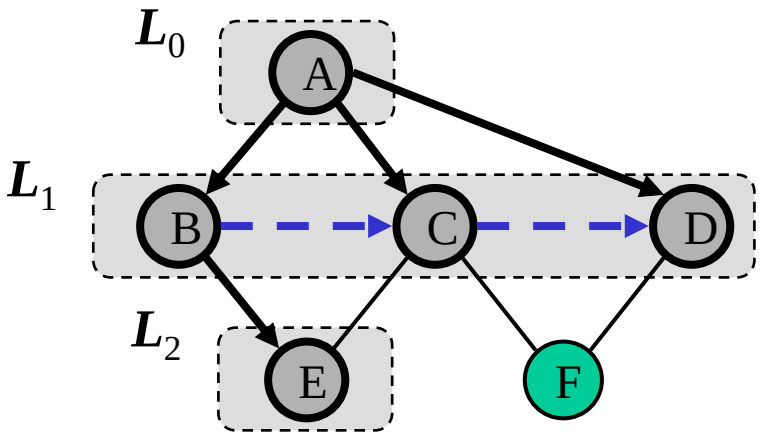


Tree



Forest

# Breadth-First Search



# Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph (next slide)

- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G

- BFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time

- BFS can be further extended to solve other graph problems
  - Find a path with the minimum number of edges between two given vertices (shortest path)
  - Find a simple cycle, if there is one

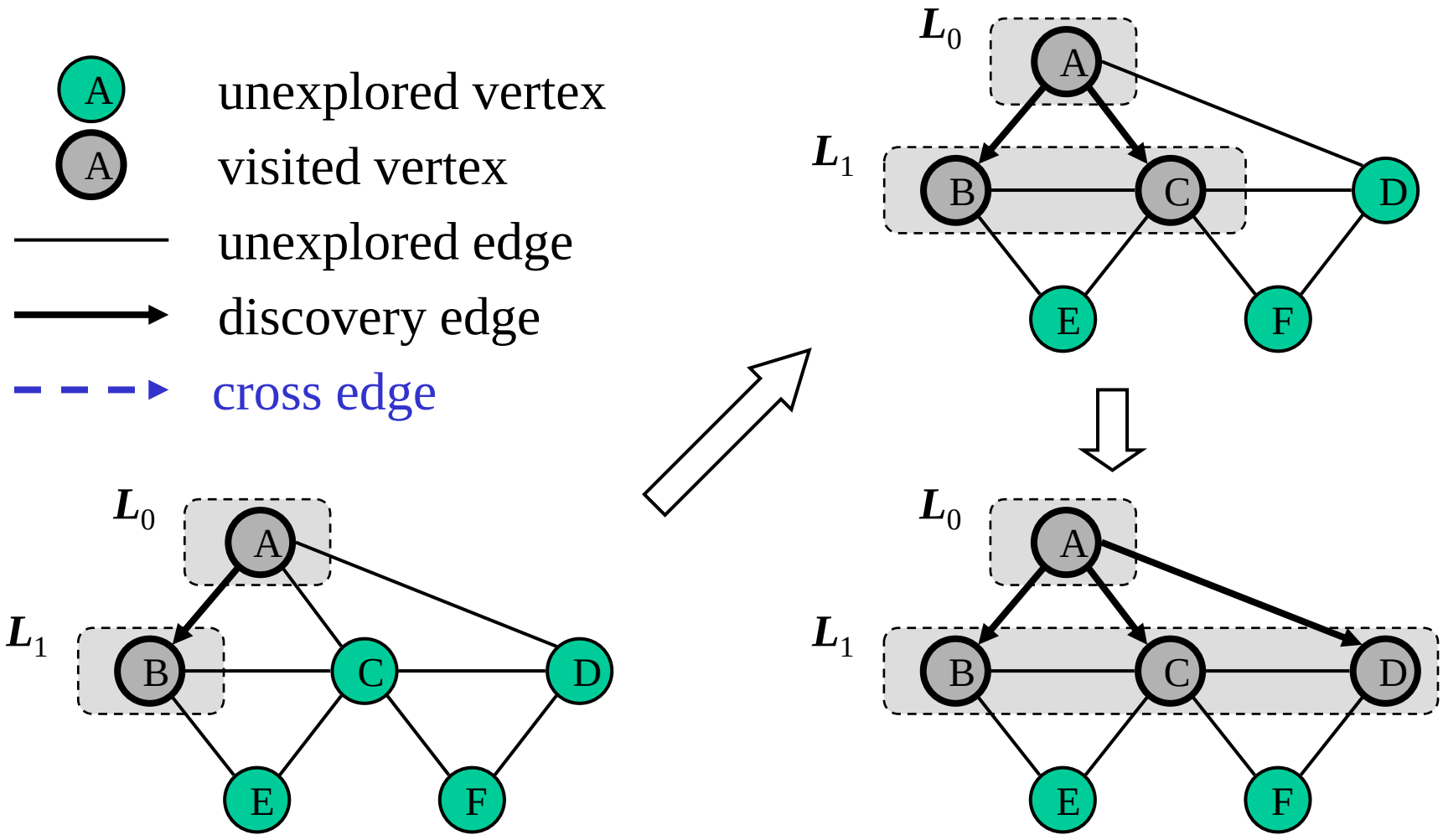
# BFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

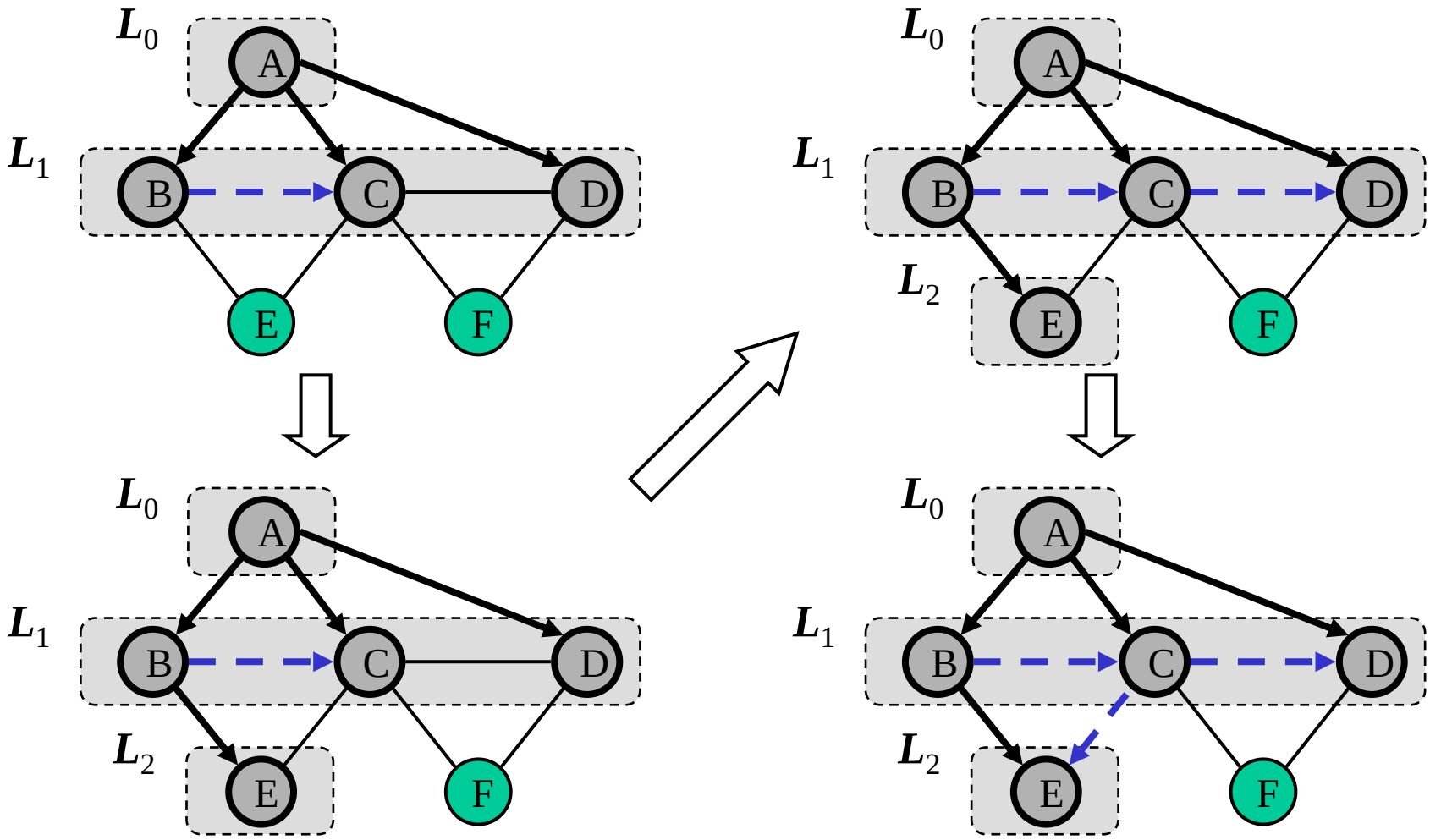
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G, v)
```

```
Algorithm BFS(G, s)
    L0 ← new empty sequence
    L0.addLast(s)
    setLabel(s, VISITED)
    i ← 0
    while ¬Li.isEmpty()
      Li+1 ← new empty sequence
      for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
          if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
              setLabel(e, DISCOVERY)
              setLabel(w, VISITED)
              Li+1.addLast(w)
            else
              setLabel(e, CROSS)
      i ← i +1
```

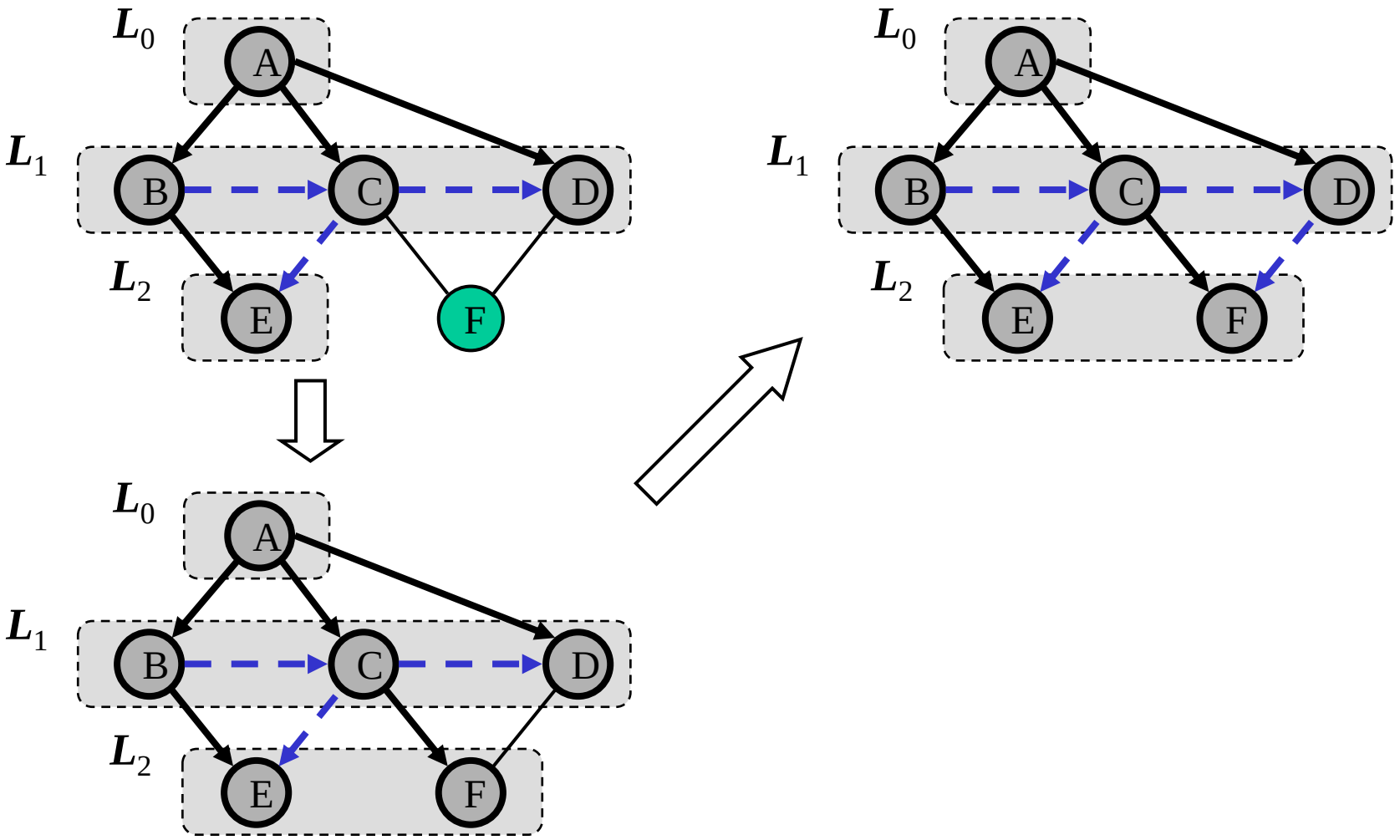
# BFS: Example


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
cross edge
$L_0$
A
$L_1$
B
C
D
E
F
$L_0$
A
$L_1$
B
C
D
E
F
$L_0$
A
$L_1$
B
C
D
E
F

# BFS: Example



# BFS: Example



# Properties

Notation

$G_s$: connected component of $s$

Property 1

***BFS***(***G, s***) visits all the vertices and edges of $G_s$

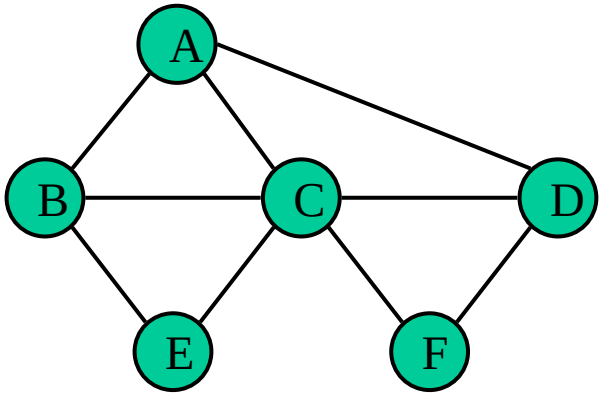
Property 2

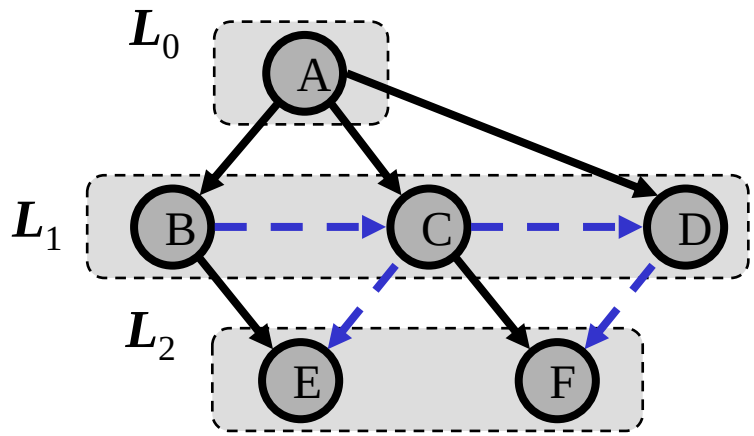
The discovery edges labeled by ***BFS***(***G, s***) form a spanning tree $T_s$ of $G_s$

Property 3

For each vertex $v$ in $L_i$

- The path of $T_s$ from $s$ to $v$ has $i$ edges
- Every path from $s$ to $v$ in $G_s$ has at least $i$ edges



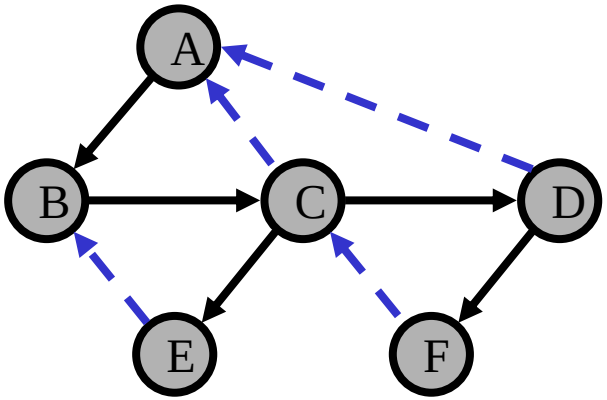



# Analysis

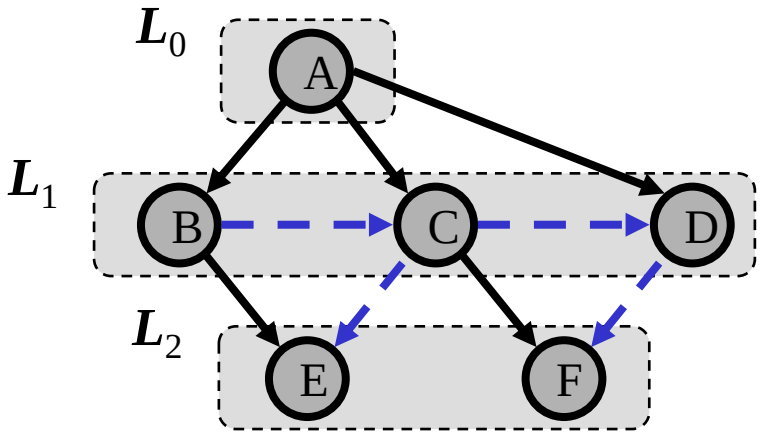
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice: $O(n)$
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice: $O(m)$
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence $L_i$ : $O(n)$
- Method **incidentEdges/outgoingEdges** is called once for each vertex: $O(m)$
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$

# DFS vs BFS

| Applications | DFS | BFS |
|---|---|---|
| Spanning forest, connected components, paths, cycles | √ | √ |
| Shortest paths | | √ |
| Biconnected components | √ | |



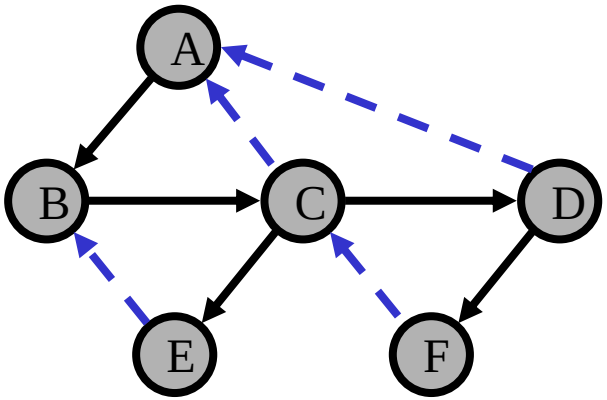

DFS



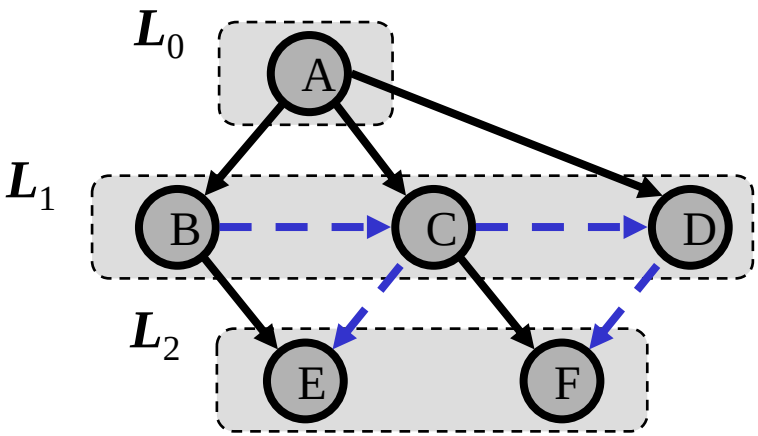

BFS

# DFS vs BFS

- Back edge (*v*,*w*)
  - *w* is an ancestor of *v* in the tree of discovery edges
- Cross edge (*v*,*w*)
  - *w* is in the same level as *v* or in the next level




DFS




BFS