# Advanced Algorithms and Computational Models (module A)

Giacomo Fiumara
gfiumara@unime.it

2023-2024

# Outline of the course (at large)

- Python basics
- Graph theory
- Network Science

# Outline of the course

## Python basics: some details

- Introduction to Python programming
- Functions
- Intro to Object Oriented Programming

# Outline of the course

## Graph Theory: some details

- Introduction and basic definitions
- Data structures
- Graph exploration
- Shortest paths
- Minimum spanning trees

# Outline of the course

## Network Science: some details

- Introduction and basic definitions
- Random network models
- Real networks
- Dynamic models
- Communities
- Epidemics spreading

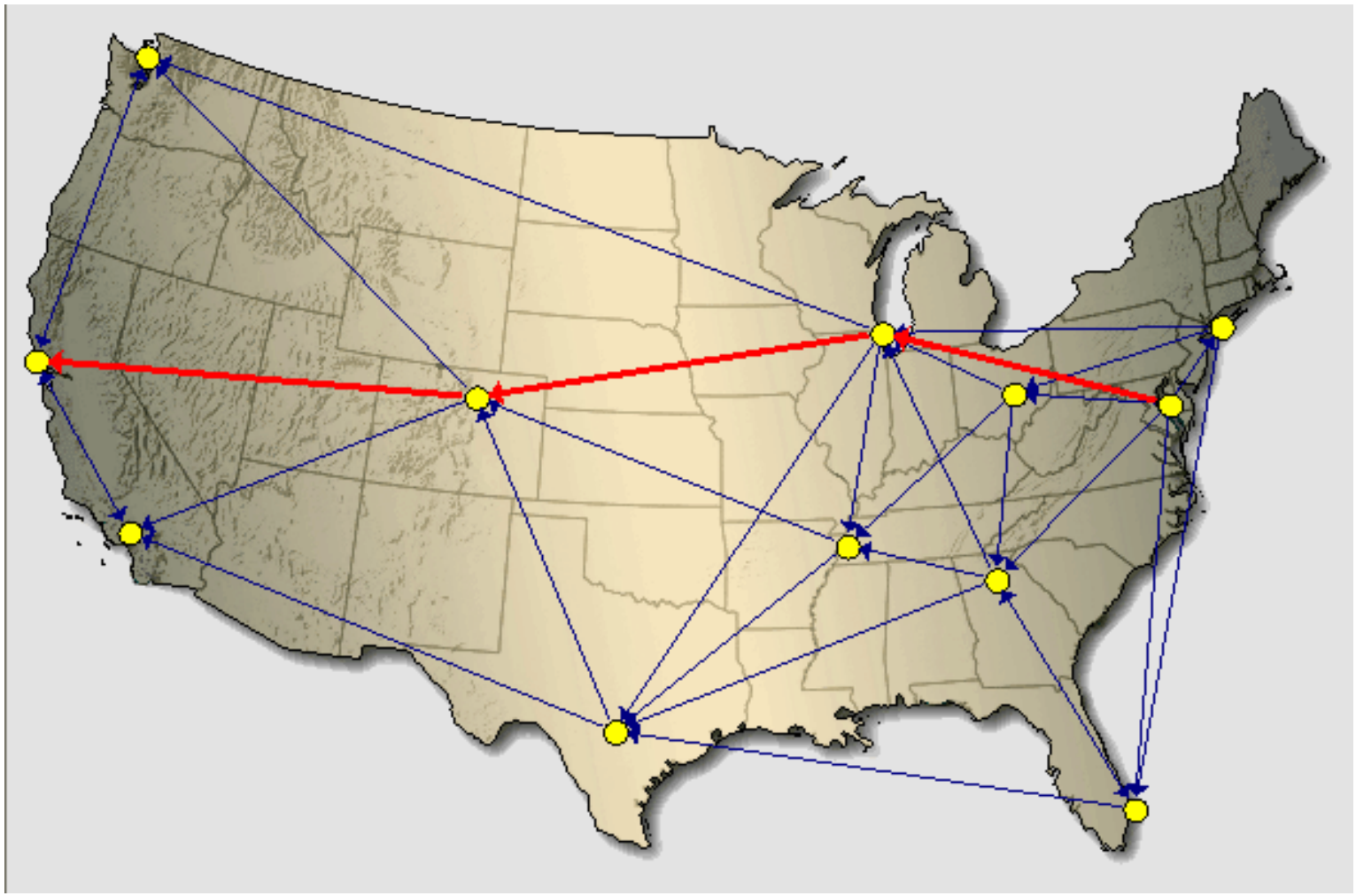
# Graph Theory

- Pairwise connections between items play a critical role in a vast number of computational applications
- The relationship implied by these connections lead to some natural questions:
  - Is there a way to connect one item to another?
  - How many other items are connected to a given item?
  - What is the shortest chain of connections between this item and this other item?

- A person who is planning a trip may need to answer questions such as "What is the shortest route from A to B?"
- An experienced traveler who has experimented traffic delays on the shortest route may ask the question "What is the **fastest** way from A to B?"

# Graph Theory

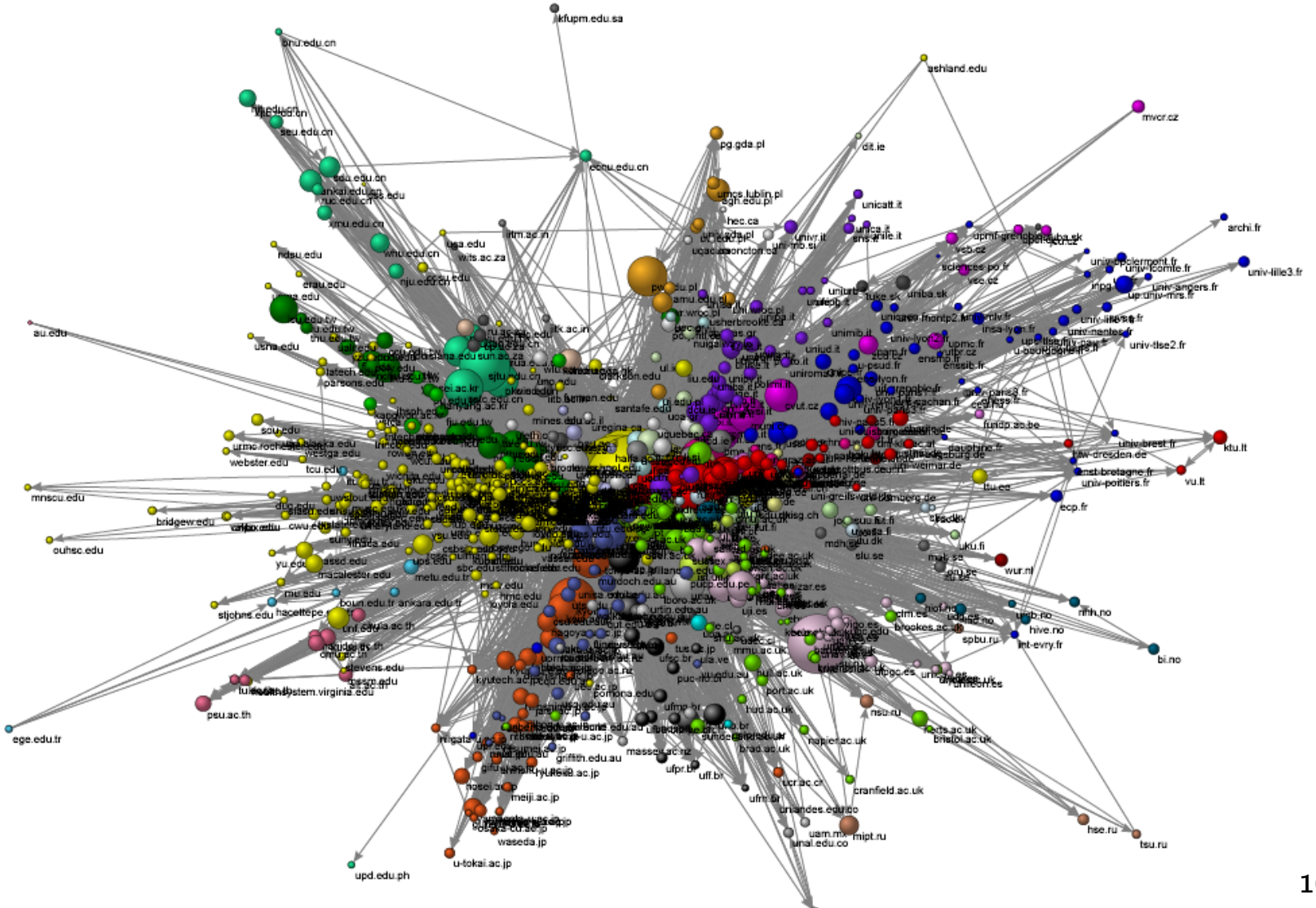
## An example: maps (shortest path)



- When one browses the web, it is common to encounter pages that contain links to other pages and it is common to move from one page to another by clicking on the links
- The entire web is a graph, where the items are pages and the connections are links

# Graph Theory

## An example: Web content (Connections among academic web sites)



- An electric circuit is composed of elements such as transistors, resistors and capacitors that are wired together
- It is necessary to answer questions such as "Is a short-circuit present?" or "It is possible to lay out this circuit on a chip without making any wires cross?"
- The first question depends on the property of the connections (wires)
- The answer to the second questions requires detailed information about the wires, the devices that those wires connect and the physical constraints of the chip

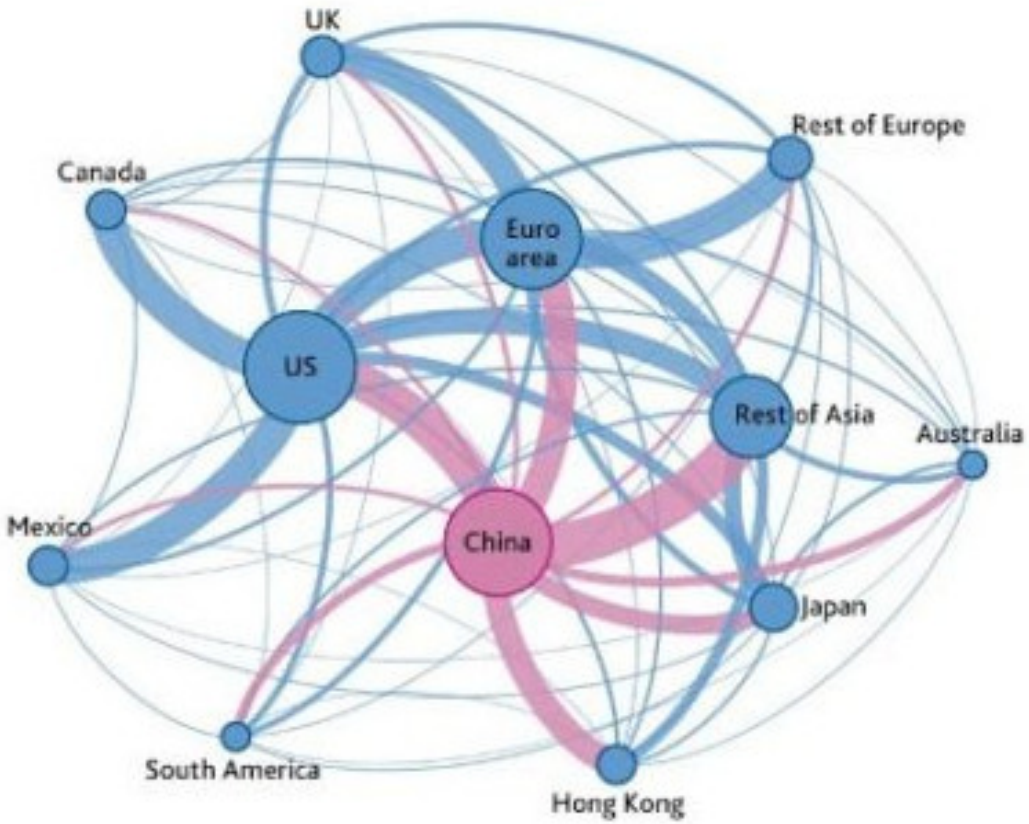
# Graph Theory

## An example: Commerce

- Retailers and financial institutions buy/sell in a market
- In this situation a connection represents the transfer of cash and goods between an institution and a customer
- Knowledge of the nature of the connection structure in this instance may enhance the understanding of the nature of the market

# An example: Commerce

## Network of global goods trade, 2018



# An example: Social Networks

- When a user uses a social network, he builds explicit connections with his friends
- Items correspond to people; connections are to friends or followers
- Understanding the properties of these networks is an application of great interest not just to companies that support such networks, but also in politics, diplomacy, entertainment, education, marketing and many other domains

# An example: Social Networks

## Spatial distribution of Facebook (as of 2010)

# An example: Social Networks

## Graph of scientific relations with Erdos

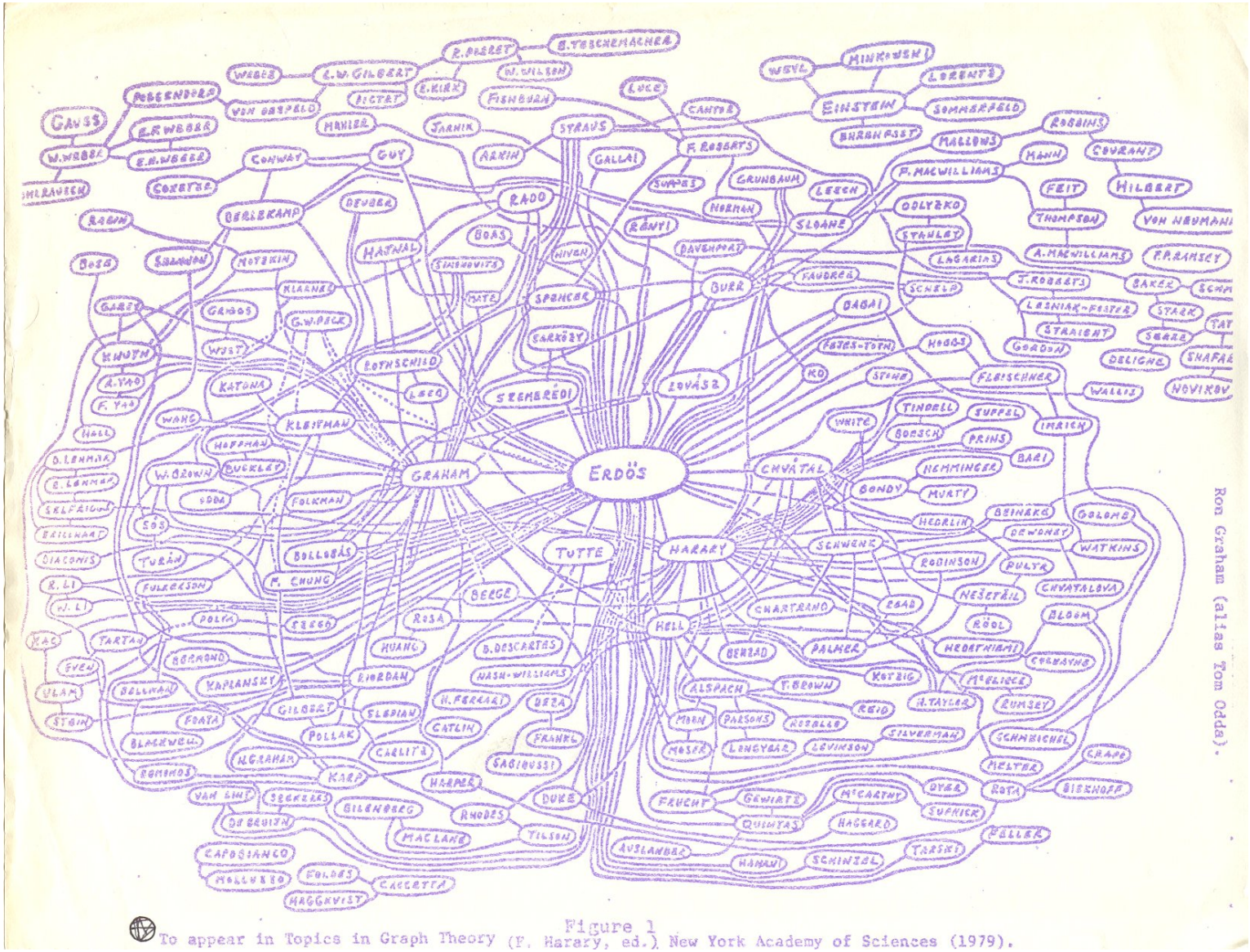


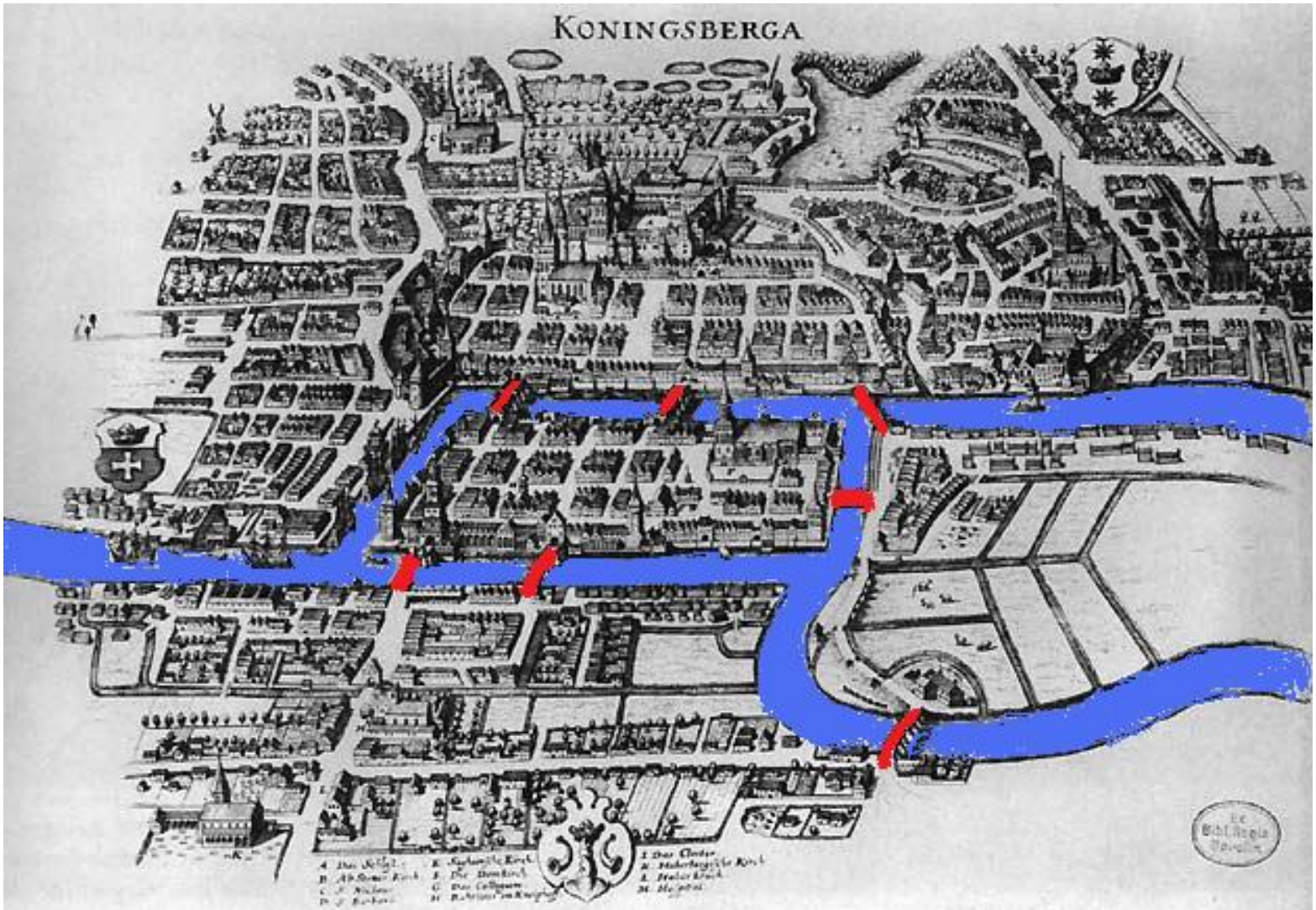

Figure 1

To appear in Topics in Graph Theory (F. Harary, ed.) New York Academy of Sciences (1979).

- Citizens of Königsberg (now Kaliningrad, Russia, then in Prussia) on Sundays used to walk through the streets of the town and crossed the bridges over the river Pregel (and its tributaries)
- According to the legend they asked for a closed path that crossed once and only once every bridge
- The problem was submitted to the famous mathematician Leonhard Euler

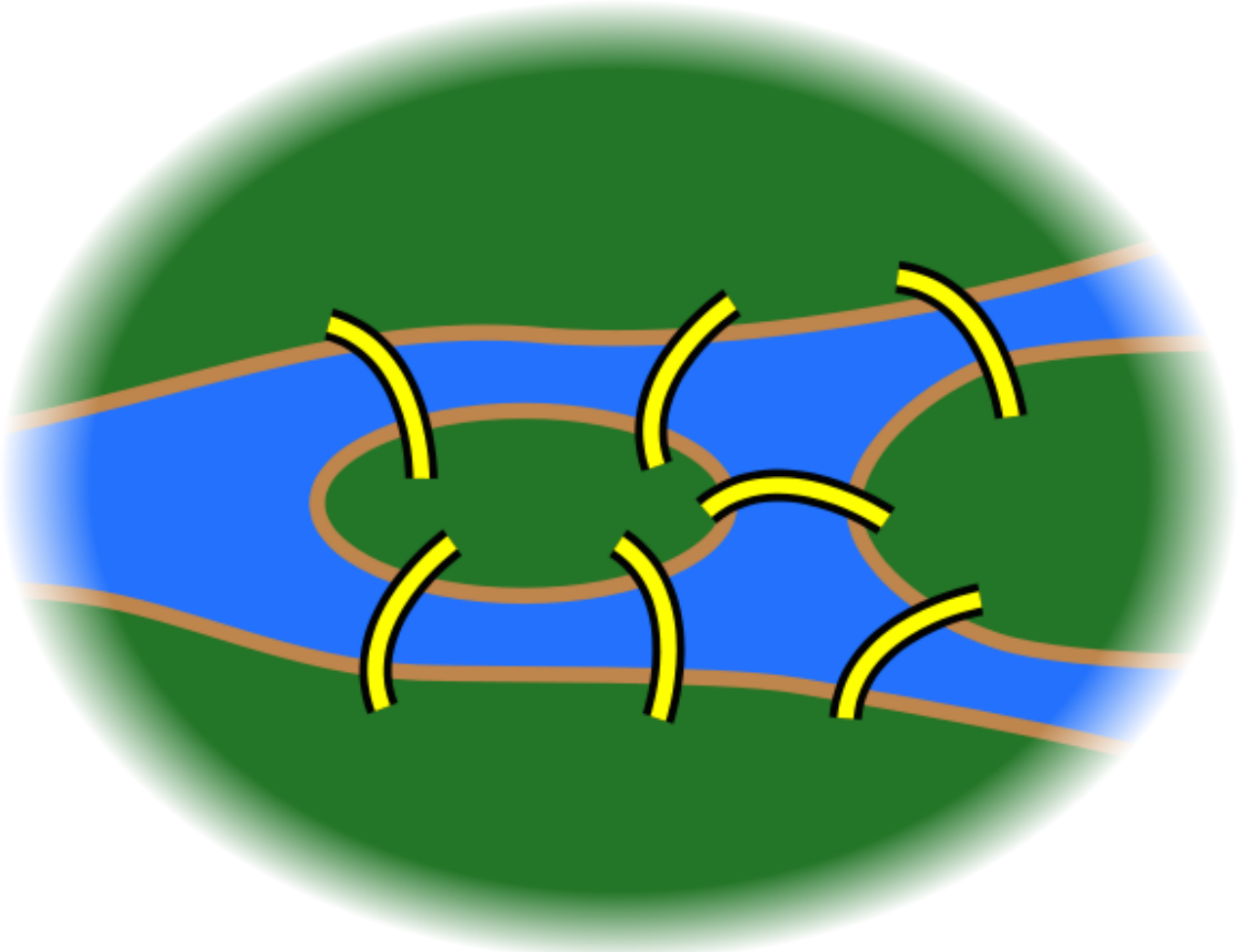
# Graph Theory

## The first problem in graph theory



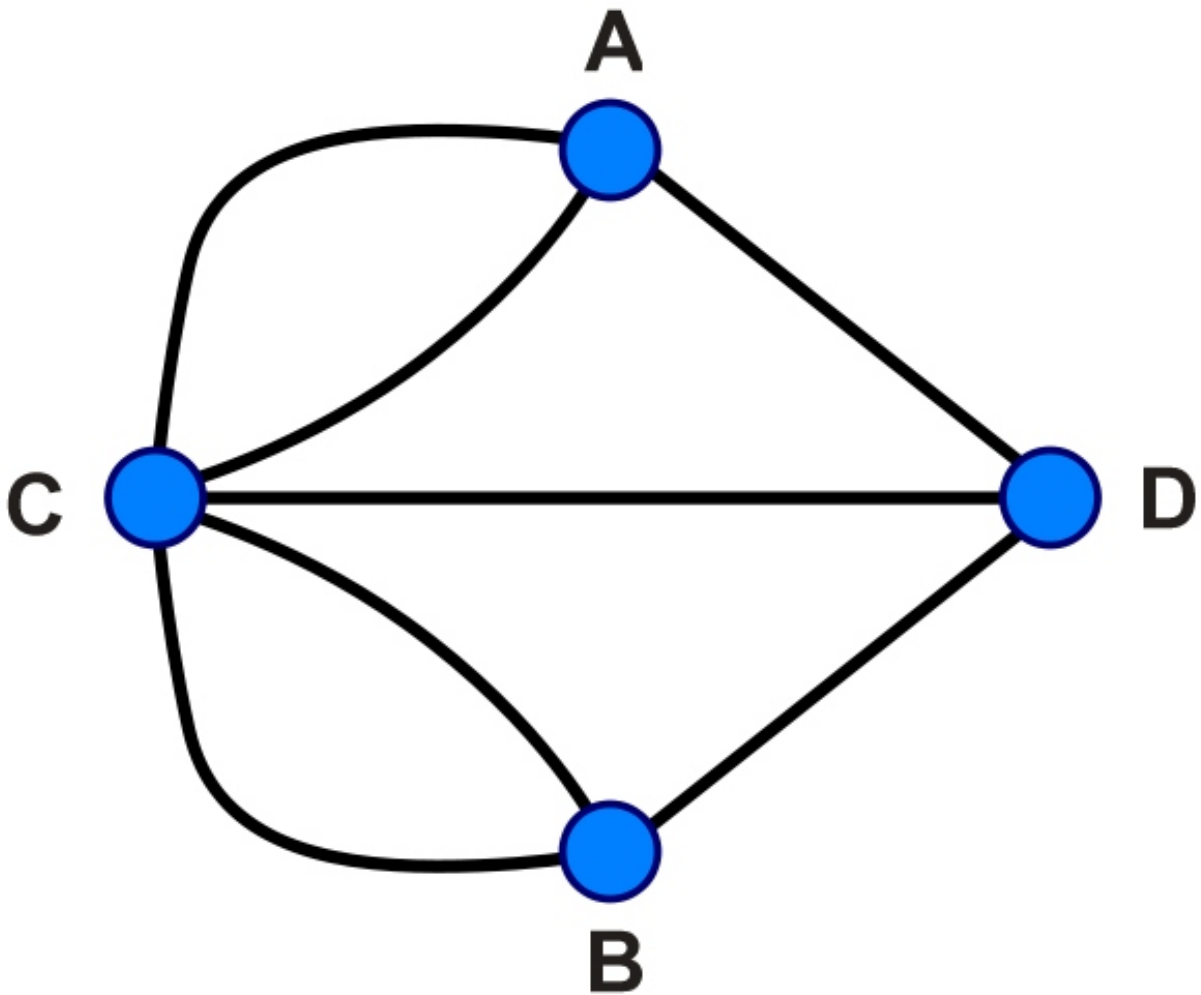
# Graph Theory

## The first problem in graph theory



# Graph Theory

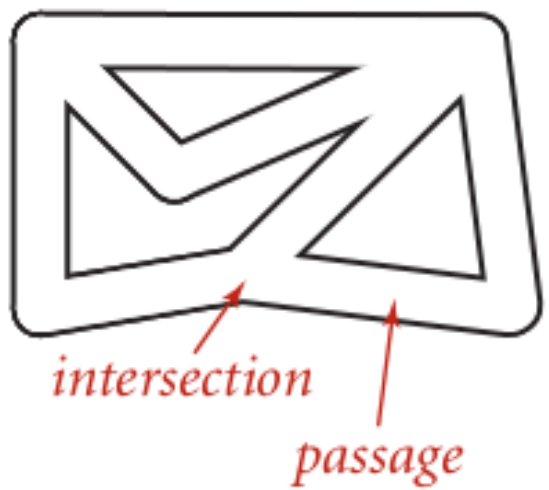
## The first problem in graph theory



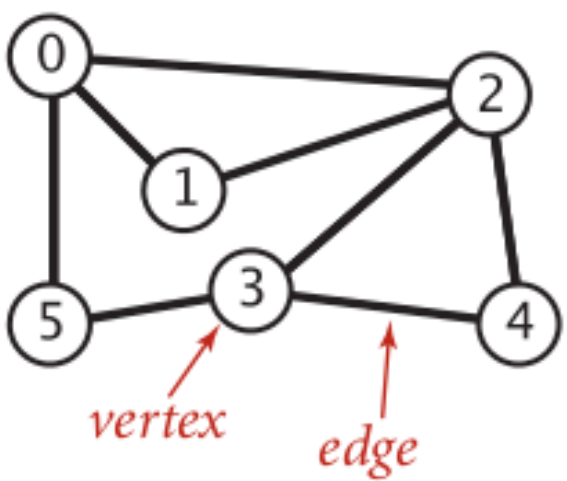
# Graph Theory

## The abstraction from routes to graph

**maze**




**graph**



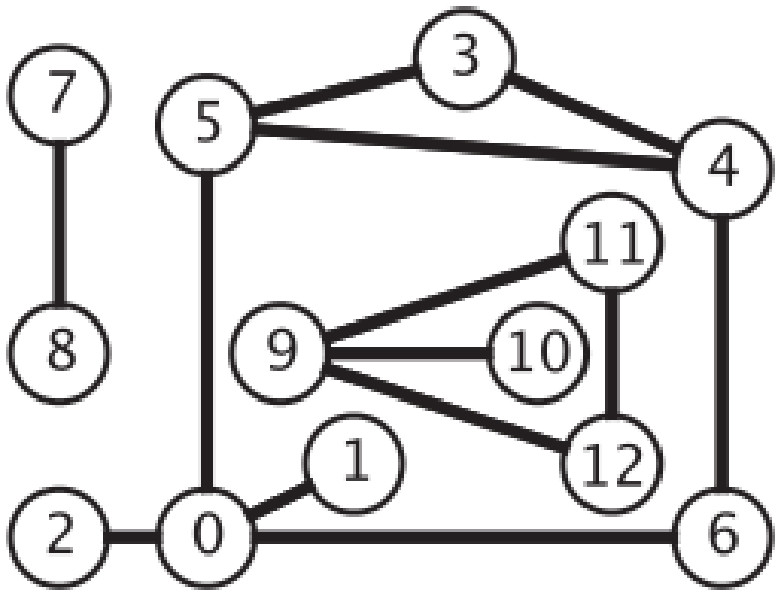
# Graph Theory

## Types of graph models

- Undirected graphs
- Directed graphs (or digraphs)
- Edge-weighted graphs
- Edge-weighted directed graphs

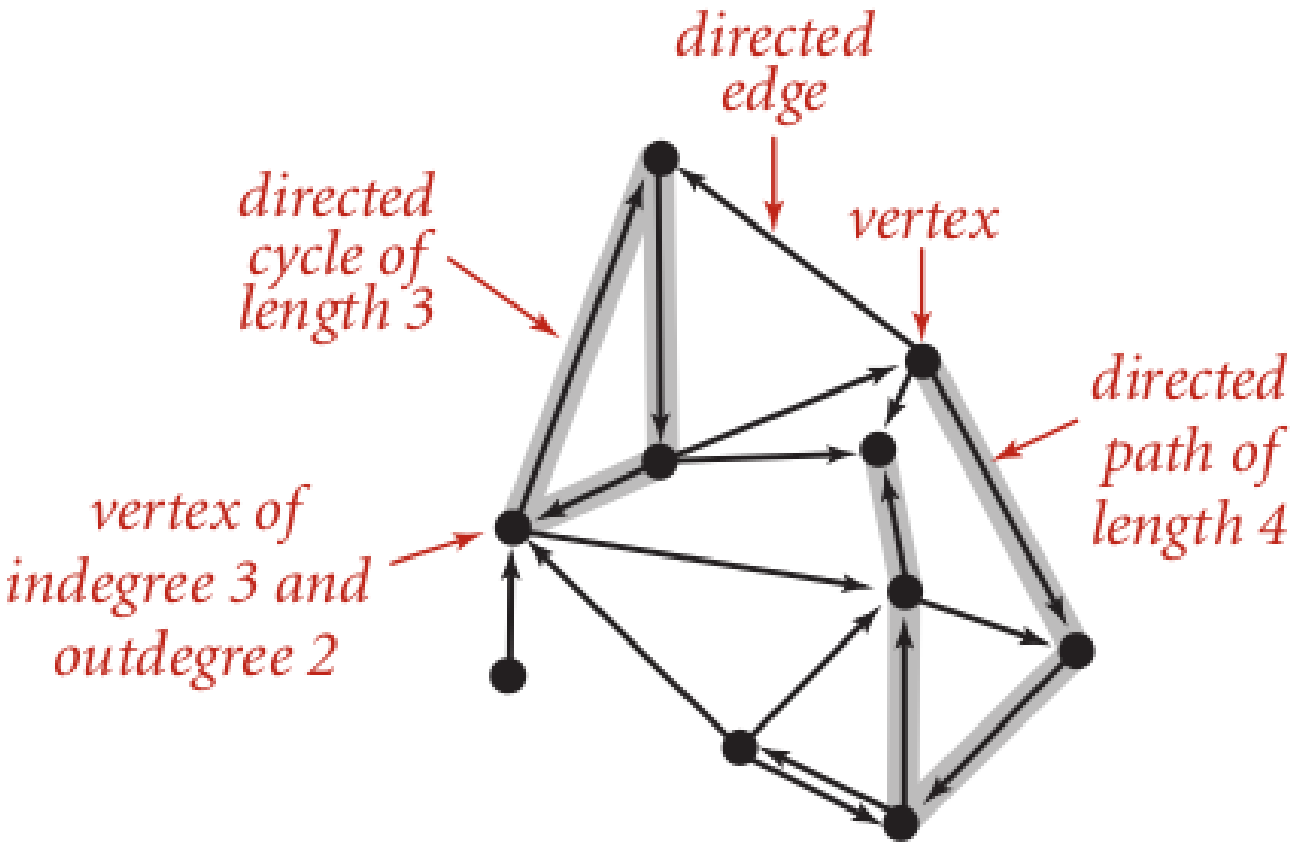
# Graph Theory

## An example of undirected graph



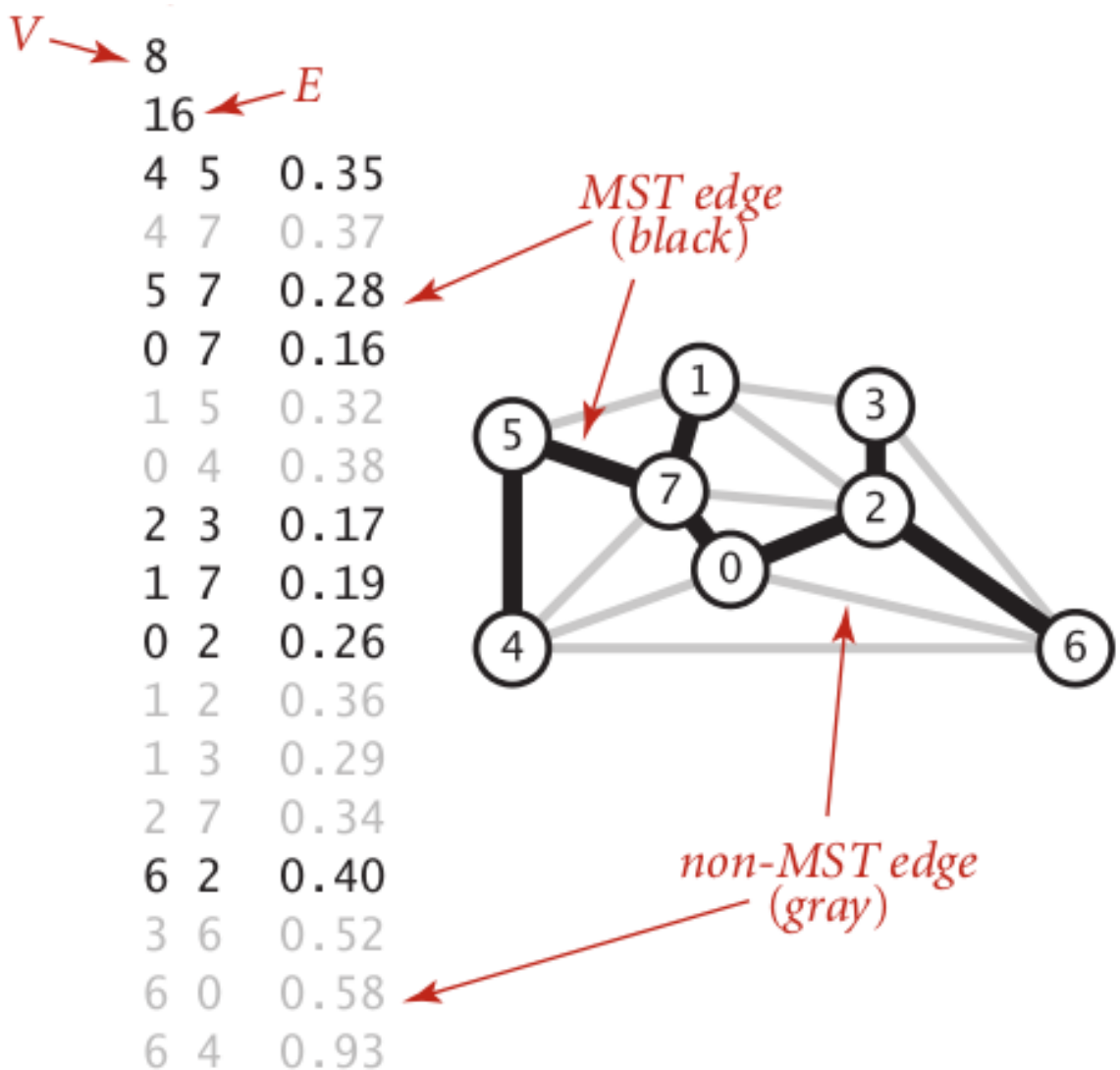
# Graph Theory

## An example of directed graph



# Graph Theory

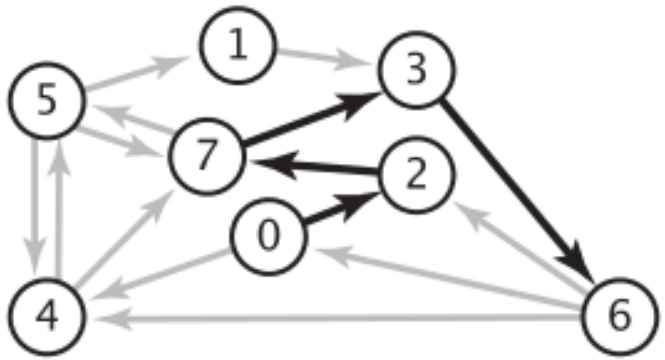
## An example of edge-weighted undirected graph



# Graph Theory

## An example of edge-weighted directed graph

```
4->5   0.35
5->4   0.35
4->7   0.37
5->7   0.28
7->5   0.28
5->1   0.32
0->4   0.38
0->2   0.26
7->3   0.39
1->3   0.29
2->7   0.34
6->2   0.40
3->6   0.52
6->0   0.58
6->4   0.93
```




**shortest path from 0 to 6**

```
0->2   0.26
2->7   0.34
7->3   0.39
3->6   0.52
```

# References

- Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, *Data Structures and Algorithms in Python*
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Third Edition, MIT Press
- Albert László Barabási, *Network Science*
- Additional readings

# Final exam howto (Module A only)

- Regularly and actively attend the lectures
- 3 assignments (10% each)
- Final project (40%)
- Project presentation (30%)

Final grade:

$$G = \frac{G_A + G_B}{2}$$

# Final Project (preliminaries)

Prepare a presentation (exactly 5 slides)

- What are your nodes and links
- How will you collect the data
- Expected size of the network (number of nodes, number of links)
- What questions you plan to ask (you may change them as the class moves on)
- Whey this network is important

# Tools

- Your notes and books
- Your laptop/desktop
- Python3
- Anaconda3 (a rich set of Python modules)
- NetworkX (a module for networks)
- In alternative: use Google Colab (no installation needed)

# Python primer

# Python primer

- The Python programming language was developed by Guido von Rossum in the early 1990s
- It has now become a prominently used language in industry and academics
- The second major version, Python 2, was released in 2000
- The third major version, Python 3, was released in 2008
- They are both maintained and freely available at `www.python.org`

# The Python Interpreter

- Python is formally an **interpreted** language
- This means that commands are executed through a software called **interpreter** which receives a command, evaluates it and reports the result of the command
- The interpreter can be used interactively (say, for debugging purposes), while a programmer typically defines a series of commands in advance and saves them in a plain text file (**source file**)
- Python source files are conventionally stored in a file named with the `.py` suffix
- The Python interpreter is invoked by typing `python` from the command line

# Objects in Python

- Python is an object-oriented language
- **Classes** form the basis for all data types
- In Python's object model, some built-in classes are the **`int`** class for integers, the **`float`** for floating-point values and the **`str`** for character strings

# The assignment statement

- The most important of all Python commands is the **assignment statement**
- For example:

```
temperature = 98.6
```

- This command establishes `temperature` as an **identifier** and then associates it with an object expressed on the rhs of the equal sign, in this case a floating-point with value 98.6.

# Identifiers

- Identifiers in Python are **case-sensitive**
- Can be composed of almost any combination of letters, numerals and underscore characters
- An identifier cannot begin with a numeral
- There are 33 specially reserved words:

**Reserved words**

| | | | | | |
|---|---|---|---|---|---|
| False | None | True | and | as | assert |
| break | class | continue | def | del | elif |
| else | except | finally | for | from | global |
| if | import | in | is | lambda | nonlocal |
| not | or | pass | raise | return | try |
| while | with | yeld | | | |

# Identifiers

- Each identifier is implicitly associated with the **memory address** of the object to which it refers
- Python is a *dynamically typed language*: there is no advance declaration associating an identifier with a particular data type
- **Caution**: although an identifier has no declared type, the object to which it refers has a definite type

# Identifiers

- A programmer can establish an **alias** by assigning a second identifier to an existing object
- For example:

```
original = temperature
```

- Once an alias has been established, either name can be used to access the underlying object
- If the object supports actions that affects its state, changes enacted through one alias will be apparent when using the other alias
- However, if one of the names is reassigned to a new value using a subsequent assignment statement, that does not affect the aliased object (the alias is broken)

# Identifiers

- For example:

```
temperature = temperature + 5.0
```

- The result of the expression on the rhs is stored as a new floating-point instance and the name on the lhs (`temperature`) is reassigned to the result
- The consequence is that this command has no effect on the value of the existing float instance that the identifier `original` continues to reference

# Identifiers

```
In [1]: temperature = 98.6

In [2]: original = temperature

In [3]: temperature = temperature + 5.0

In [4]: temperature
Out[4]: 103.6

In [5]: original
Out[5]: 98.6
```

# Instantiation

- The process of creating a new instance of a class is known as *instantiation*
- The syntax for instantiating an object is to invoke the *constructor* of the class
- Many of Python's built-in classes support what is known as a *literal* form for designating new instances
- For example, the assignment:

```
temperature = 98.6
```

results in the creation of a new instance of the **float** class, the term 98.6 being the literal form

# Calling methods

- Python supports traditional functions that are invoked with a syntax such as

```
sorted(data)
```

- in which `data` is a parameter sent to the function
- Python's classes may also define one or more **methods** (also known as **member functions**) which are invoked on a specific instance of a class using the dot operator
- For example:

```
data.sort()
```

# Calling methods

- Some methods return information about the state of an object, but do not change that state
- They are called **accessors**
- For example:

```
data.startswith('y')
```

- Other methods, such as the `sort` method of the `list` class, do change the state of the object
- They are called **mutators**

# Python's built-in classes

- A class is **immutable** if each object of that class has a fixed value upon instantiation that cannot subsequently be changed
- For example, the class `float` is immutable, since once an instance has been created, its value cannot be changed

| Class | Description | Immutable |
|---|---|---|
| `bool` | Boolean value | y |
| `int` | integer | y |
| `float` | floating-point number | y |
| `list` | mutable sequence of objects | n |
| `tuple` | immutable sequence of objects | y |
| `str` | character string | y |
| `set` | unordered set of distinct objects | n |
| `frozenset` | immutable form of set class | y |
| `dict` | associative mapping (aka dictionary) | n |

# The `bool` class

- The `bool` class is used to manipulate Boolean values
- The two only instances of the class are expressed as the literals `False` and `True`
- The default constructor, `bool()` returns `False`

```
goofy = bool()
pluto = False
```

- Python allows the creation of a Boolean value from a non-boolean type using the syntax `bool(huey)` for value `huey`
- The interpretation depends upon the type of the parameter
  - Numbers evaluate to `False` if zero, and `True` if nonzero
  - Strings and lists evalutate to `False` if empty, and `True` if nonempty

# The `int` class

- The `int` class is designed to represent integer values with arbitrary magnitude
- Unlike Java and C++ which support different integer types with different precision, Python automatically chooses the internal representation for an integer based upon the magnitude of its value
- In some contexts may be useful to express an integral value using binary, octal or hexadecimal. For example:

```
0b110101
0o755
0x9ab
```

# The `int` class

```
In [6]: a = 0b1101

In [7]: a
Out[7]: 13

In [8]: b = 0o755

In [9]: b
Out[9]: 493

In [10]: c = 0xff

In [11]: c
Out[11]: 255
```

# The `int` class

- The constructor `int()` returns value 0 by default
- However, it can be used only to construct an integer value starting from an existing value of another type
- For example

```
In [12]: int(goofy)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call la
st)
<ipython-input-12-798d88bb7518> in <module>()
----> 1 int(goofy)

NameError: name 'goofy' is not defined

In [13]: goofy = 12.4

In [14]: int(goofy)
Out[14]: 12

In [15]: int()
Out[15]: 0
```

# The `int` class

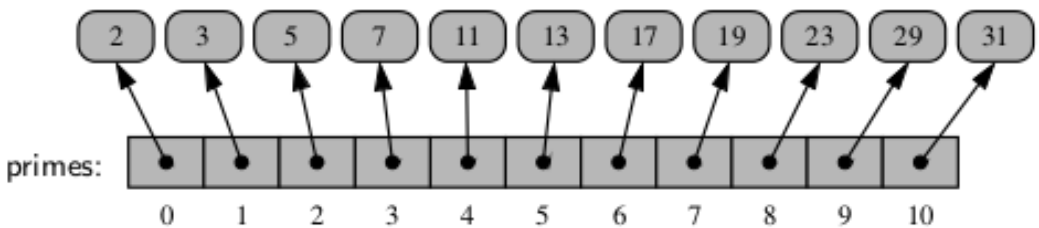
- If `goofy` represents a floating-point value, `int(goofy)` produces the *truncated* value of `goofy`
- The constructor can be used to parse a string which is presumed to represent an integral value: `int('150')` produces the integer value 150
- By default, the string must use base 10
- If conversion from another base is desired, that base can be indicated as a second optional parameter, for example: `int('7f', 16)` evaluates to the integer 127

# The `float` class

- The `float` class is the sole floating-point type in Python with a fixed-precision
- Literal forms are 98.6 and 10*e*3
- The constructor form of `float()` returns 0.0. When given a parameter, the constructor attempts to return the equivalent floating-point value
- For example, the call `float(2)` returns 2.0
- If the parameter to the constructor is a string (`float('14.3')`), it attempts to parse the string as a floating-point value

# The `list` class

- A `list` instance stores a sequence of objects
- A list is a referential structure, as it technically stores a sequence of *references* to its elements
- Lists are *array-based* sequenced and are *zero-indexed* (a list of length $n$ has elements indexed from 0 to $n-1$ inclusive)
- Lists are probably the most used container types in Python, thay have many valuable behaviors, including the ability to dynamically expand and contract their capacities as needed



# The `list` class

- Python uses the characters [ and ] as delimiters for a `list` literal, with [] being en empty list. For example:

```
['one', 'two', 'three']
```

- If identifiers *a*, *b*, *c* have been established, the following syntax is correct:

```
[a, b, c]
```

# The `list` class

```
In [16]: a = 1

In [17]: b = 2.89

In [18]: c = 'hello everybody'

In [19]: list1 = [a,b,c]

In [20]: list1
Out[20]: [1, 2.89, 'hello everybody']
```

# The `list` class

- The `list()` constructor produces an empty list by default
- However, the constructor will accept any parameter that is of *iterable* type (e.g., strings, lists, tuples, sets, dictionaries)
- For example:

```
In [21]: list('hello everybody')
Out[21]: ['h', 'e', 'l', 'l', 'o', ' ', 'e', 'v', 'e', 'r', 'y', 'b', 'o', 'd',
         'y']
```

# The `tuple` class

- The `tuple` class provides an immutable version of a sequence
- Python uses the characters ( and ) to delimit a tuple, with () being an empty tuple
- To express a tuple of length one as a literal, a comma must be placed after the element (within the parentheses). For example:

```
In [22]: aa = 22

In [23]: bb = (22,)

In [24]: aa
Out[24]: 22

In [26]: bb
Out[26]: (22,)
```

# The `str` class

- The `str` class is designed to represent an immutable sequence of characters based upon the Unicode international character set
- Strings have a more compact internal representation than lists and tuples

# The `str` class

- Strings can be enclosed in single quotes, as in *'ciao'* or double quotes, as in *"ciao"*
- The second choice is convenient, in particular when using another of the quotation characters as in *"L'attimo"*
- Alternatively, the quote delimiter can be designated using a backslash (the so-called *escape character*) as in *'L'attimo'*
- Of course, even the backslash must itself be escaped (if necessary)
- Python also supports the delimiter *' ' '* or *" " "*
- The main advantage of these delimiters is that strings naturally contain newlines

# The `set` class

- It represents the mathematical notion of a set, namely a collection of distinct elements without an inherent order
- It has a highly optimized method for checking whether a specific element is contained in the set
- Two main restrictions:
  - The set does not maintain the elements in any particular order
  - Only instances of *immutable* elements can be added to a Python set (integers, floating-point numbers, character strings)
- Python uses { and } as delimiters for a set, for example $\{1, 4, 2, 9\}$ or $'aa', 'b', 'ciao'$
- There is an important exception: {} represents an empty dictionary (not a set). To represent an empty set the constructor `set()` must be invoked

# The `set` class

```
In [27]: alist = [1,2,3,4,5]

In [28]: alist
Out[28]: [1, 2, 3, 4, 5]

In [29]: aset = {1,2,3,4,5}

In [30]: aset
Out[30]: {1, 2, 3, 4, 5}

In [31]: set('hello everybody')
Out[31]: {' ', 'b', 'd', 'e', 'h', 'l', 'o', 'r', 'v', 'y'}
```

# The `dict` class

- The `dict` class represents a **dictionary**, from a set of distinct *keys* to associated *values*
- For example, a dictionary might map from unique student ID numbers, to larger student records (name, address, course grades)
- The literal form {} represents an empty dictionary, while a nonempty dictionary is expressed using a comma-separated series of `key: value` pairs
- For example:

```
phone = {'john': 4040, 'joe': 4041, 'kate': 4042}
```

- Python supports the following keyword operators for Boolean values:

| Operator | Description |
|---|---|
| not | Unary negation |
| and | Conditional and |
| or | Conditional or |

- The `and` and `or` operators **short-circuit**, in that they do not evaluate the second operand if the result can be determined based on the value of the first operand

Python supports the following operators to test two notions of equality:

| Operator | Description |
|---|---|
| is | same identity |
| is not | different identity |
| == | equivalent |
| != | not equivalent |

# Expressions, Operators and Precedence

## Equality Operators

- The expression `a` **`is`** `b` evaluated to `True` precisely when identifiers `a` and `b` are aliases for the *same* object
- The expression `a == b` tests a more general notion of equivalence:
  - If identifiers `a` and `b` refer to the same object, then `a == b` evaluates to `True`
  - If identifiers refer to different objects that are equivalent, then `a == b` evaluates to `True`
- The precise notion of equivalence depends on the data type: two strings are considered equivalent if they match character for character, while two sets are equivalent if they have the same content, irrispective of order
- In general, use of `is` and `is not` should be reserved for situations in which it is necessary to detect true aliasing

Data types may define a natural order via the following operators

| Operator | Description |
|---|---|
| $<$ | less than |
| $<=$ | less than or equal to |
| $>$ | greater than |
| $>=$ | greater than or equal to |

- These operators have expected behavior for numeric types
- They are defined lexicographically, and case sensitively, for strings
- An exception is raised if operands have incomparable types

# Expressions, Operators and Precedence

## Comparison Operators

In [32]:
```
a = 1
```

In [33]:
```
b = 1
```

In [34]:
```
a == b
```

Out[34]: True

In [35]:
```
a != b
```

Out[35]: False

In [36]:
```
a is b
```

Out[36]: True

In [37]:
```
a <= b
```

Out[37]: True

In [38]:
```
a >= b
```

Out[38]: True

# Expressions, Operators and Precedence

## Comparison Operators

```
In [39]: x = 'Goofy'

In [40]: y = 'Goofy'

In [41]: x == y
Out[41]: True

In [42]: x < y
Out[42]: False

In [43]: x <= y
Out[43]: True

In [44]: z1 = 'aa'

In [45]: z2 = 'ab'

In [46]: z1 < z2
Out[46]: True
```

Python supports the following arithmetic operators:

| Operator | Description |
|---|---|
| + | addition |
| − | subtraction |
| * | multiplication |
| / | true division |
| // | integer division |
| % | modulo operator |

# Expressions, Operators and Precedence

## Arithmetic Operators

- The use of $+$, $-$ and $*$ is straightforward (if both operands are `int`, the result is `int`; if one or both operands are `float`, the result will be `float`)
- / denotes the **true division**, returns the quotient (float)
- // is used to perform the integral calculation
- % returns the remainder of the integer division

```
In [47]: 21 / 5
Out[47]: 4.2

In [48]: 21 // 5
Out[48]: 4

In [49]: 21 % 5
Out[49]: 1
```

- Python extends the semantics of // and % to cases where one or both operands are negative
- Let's assume that

$$n = q \cdot m + r$$

Where:

- $n$, is the dividend
- $m$, is the divisor
- $q$, is the quotient ($q = n//m$)
- $r$, is the remainder ($r = n\%m$)

$$n = q \cdot m + r$$

- When the divisor is positive, Python guarantees that $0 \leq r < m$
- As a consequence, we have that $-27//4 = -7$ and $-27\%4 = 1$
- We have indeed that $-7 \cdot 4 + 1 = -27$

- When the divisor is negative, Python guarantees that $m < r \leq 0$
- As a consequence, we have that $27//(-4) = -7$ and $27\% - 4 = -1$
- We have indeed that $-7 \cdot (-4) - 1 = 27$

Python supports the following bitwise operators for integers:

| Operator | Description |
|---|---|
| $\sim$ | bitwise complement (prefix unary operator) |
| & | bitwise and |
| \| | bitwise or |
| ^ | bitwise exclusive-or |
| << | shift bits left, filling in with zeros |
| >> | shift bits right, filling in with sign bit |

# Expressions, Operators and Precedence

## Sequence Operators

Each of Python's built-in sequence types (`str`, `tuple`, `list`) support the following operator syntaxes:

| **Operator** | **Description** |
|---|---|
| $s[j]$ | element at index $j$ |
| $s[start : stop]$ | slice including indices $[start, stop]$ |
| $s[start : stop : step]$ | slice including indices `start`, `start+step`, `start` $+2\cdot$`step`, ... up to but not equalling or `stop` |
| $s + t$ | concatenation of sequences |
| $k * s$ | shorthand for $s + s + s + \ldots$ (k times) |
| val **`in`** s | containment check |
| val **`not in`** s | non-containment check |

# Expressions, Operators and Precedence

## Sequence Operators

- Python relies on *zero-indexing* of sequences (a sequence has $n$ elements indexed from 0 to $n-1$ inclusive)
- *Negative indices* denote a distance from the end of the sequence: $index-1$ denotes the last element, $index-2$ the second to last and so on
- Python uses a *slicing* notation to describe subsequences of a sequence
  - Slices are described as half-open intervals, with a start index (included) and a stop index (excluded)
  - An optional *step* index, possibly negative, can be indicated as a third parameter of a slice
  - If a start or stop index is omitted in the slicing notation, it is presumed to designate the extreme of the original sequence

# Expressions, Operators and Precedence

## Sequence Operators

```
In [50]: a = ['a','b','c','d','e','f','g','h','i','j']

In [51]: a[0]
Out[51]: 'a'

In [52]: a[5]
Out[52]: 'f'

In [53]: a[0:3]
Out[53]: ['a', 'b', 'c']

In [54]: a[:3]
Out[54]: ['a', 'b', 'c']

In [55]: a[0:]
Out[55]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

In [56]: a[0::2]
Out[56]: ['a', 'c', 'e', 'g', 'i']
```

- Lists are mutable, therefore the following syntax is used to replace an element at a given index

```
s[j] = val
```

- Lists also support a syntax to remove the designated element from the list:

```
del s[j]
```

- Slice notation can also be used to replace or delete a sublist

The notation

```
val in s
```

can be used for any of the sequences to see if there is an element equivalent to `val` in the sequence
All sequences define comparison operators based on *lexicographic order*, performing an element by element comparison until the first difference is found. For example:

```
In [1]: ['a', 'b', 'c'] < ['a', 'B', 'c']
Out[1]: False

In [2]: ['a', 'b', 'c'] < ['a', 'B', 'c']
Out[2]: False

In [3]: ['a', 'b', 'c'] > ['a', 'B', 'c']
Out[3]: True
```

The following operations are supported by sequence types:

| Operation | Description |
|---|---|
| s == t | equivalent (element by element) |
| s != t | not equivalent |
| s < t | lexicographically less than |
| s <= t | lexicographically less than or equal to |
| s > t | lexicographically greater than |
| s >= t | lexicographically greater than or equal to |

Sets support the following operators:

| Operator | Description |
|---|---|
| key in s | containment check |
| key not in s | non-containment check |
| s1 == s2 | s1 is equivalent to s2 |
| s1 != s2 | s1 is not equivalent to s2 |
| s1 <= s2 | s1 is subset of s2 |
| s1 < s2 | s1 is proper subset of s2 |
| s1 >= s2 | s1 is superset of s2 |
| s1 > s2 | s1 is proper superset of s2 |
| s1 \| s2 | the union of s1 and s2 |
| s1 & s2 | the intersection of s1 and s2 |
| s1 - s2 | the set of elements in s1 but not in s2 |
| s1^s2 | the set of elements in precisely one of s1 or s2 |

- Sets do not guarantee a particular order of their elements, so the comparison operators (such as $<$) are not lexicographic; rather, they are based on the mathematical notion of subset
- The consequence is that the comparison operators define a partial order
- Dictionaries, like sets, do not maintain a well-defined order on their elements
- The concept of subset is not particularly meaningful for dictionaries, so the `dict` class does not support operators such as $<$
- Dictionaries support the notion of equivalence, with $d1 == d2$ if the two dictionaries contain the same set of key-value pairs

The most widely used behavior of dictionaries is accessing a value associated with a particular key $k$ with the indexing syntax, $d[k]$. The supported operators are the following:

| **Operator** | **Description** |
|---|---|
| d[key] | value associated with the given key |
| d[key] = value | set (or reset) the value associated with given key |
| del d[key] | remove key and its associated value from dictionary |
| key in d | containment check |
| key not in d | non-containment check |
| d1 == d2 | d1 is equivalent to d2 |
| d1 != d2 | d1 is not equivalent to d2 |

# Expressions, Operators and Precedence

## Compound Expressions and Operator Precedence

| | Type | Symbols |
|---|---|---|
| 1 | member access | expr.member |
| 2 | function/method calls | expr(...) |
| 3 | exponentiation | ** |
| 4 | unary operators | +expr, -expr |
| 5 | multiplication, division | *, /, //, % |
| 6 | addition, subtraction | +, - |
| 7 | bitwise shifting | «, » |
| 8 | bitwise and | & |
| 9 | bitwise xor | ^ |
| 10 | bitwise or | \| |
| 11 | comparisons (containment) | is, is not, ==, !=, <, <=, >, >=, in not in |
| 12 | logical not | not expr |
| 13 | logical and | and |
| 14 | logical or | or |
| 15 | conditional | val1 if cond else val2 |
| 16 | assignments | =, +=, -=, *=, etc. |

# Control flow

## Conditionals

- Conditional constructs provide a way to execute a chosen block of code based on the run-time evaluation of one or more Boolean expressions

```
if first_condition:
  first_body
elif second_condition:
  second_body
elif third_condition:
  third_body
else:
  fourth_body
```

# Control flow

## Conditionals

- Each condition is a Boolean expression, and each body contains one or more commands that are executed conditionally
- There may be any number of `elif` clauses; the final `else` clause is optional
- Nonboolean types may be evaluated as Boolean with intuitive meanings, for example

```
if response:
```

- is equivalent to:

```
if response != '':
```

For example:

```
if door_is_closed:
  if door_is_locked:
    unlock_door()
  open_door()
advance()
```

The final command, `advance()`, is not indented and therefore not part of the conditional body. It will be executed unconditionally

The syntax for a `while` loop in Python is as follows:

```
while condition:
  body
```

- *condition* can be an arbitrary Boolean expression
- *body* can be an arbitrary block of code (including nested control structures)

# Control flow

## Loops: while loops

- The execution of a `while` loop begins with a test of the Boolean condition: if that condition evaluates to `True`, the body of the loop is performed
- After each iteration of the body, the loop condition is retested, and if it evaluates to `True`, another iteration of the body is performed
- When the test condition evaluates to `False` (if ever), the loop is exited and the flow of control continues just beyond the body of the loop

An example:

```
j = 0
while j < len(data) and data[j] != 'X':
  j += 1
```

- The correctness of this loop relies on the behavior of the `and` operator
- In this code fragment, first is tested `j < len(data)` to ensure that *j* is a valid index and only after the relative element of `data` is accessed
- If the compound condition was written in the opposite order, the evaluation of `data[j]` would eventually raise an `IndexError` when 'X' is not found.

The syntax for a `for-loop` in Python is as follows:

```
for element in iterable:
  body
```

The for-loop syntax can be used on any type of iterable structure, such as `list`, `set`, `dict` or `file`
An example:

```
biggest = data[0]
for val in data:
  if val > biggest:
    biggest = val
```

- In some applications is needed the index of an element within the sequence, for example the index of the biggest element in a list
- In this case it is possible to loop over the all possible indices of the list
- To this purpose, Python provides a built-in class named `range` that generates integer sequences. The syntax `range(n)` generates the series of $n$ values from 0 to $n-1$
- For example:

```
bigindex = 0
for j in range(len(data)):
  if data[j] > data[bigindex]:
    bigindex = j
```

# Control flow

## Loops: for loops

- Python supports a `break` statement that immediately terminates a while or for loop when executed within its body
- For example:

```
found = False
for item in data:
  if item == target:
    found = True
    break
```

- Python also supports a `continue` statement that causes the current iteration of a loop body to stop, but with subsequently passes of the loop proceeding as expected

# Functions

- Python supports both **functions** and **methods**
- The term *function* refers to a traditional, stateless function that is invoked without the context of a particular class or an instance of that class, such as `sorted(data)`
- The term *method* is used to describe a member function that is invoked upon a specific object using an object-oriented message passing syntax, such as `data.sort()`

# Functions

```
def count(data, target):
  n = 0
  for item in data:
    if item == target:
      n += 1
  return n
```

- The first line establishes a new identifier as the name of the function and the number of parameters that it expects
- Each time a function is called, Python creates a dedicated **activation record** that stores information relevant to the current call
- The activation record includes the **namespace**, to manage all the identifiers having a local scope within the current call

# Functions

## Return Statement

- A `return` statement indicates that the function should immediately cease execution, and that an expressed value should be return to the caller
- If a return statement is executed without an explicit argument, the `None` value is automatically returned
- Often, a `return` statement is the final command within the body of the function
- However, multiple `return` statements are admitted in the same function, with conditional logic controlling which command will be executed

```
def contains(data, target):
  for item in data:
    if item == target:
      return True
  return False
```

- The identifiers used to describe the expected parameters are known as **formal parameters**, and the objects sent by the caller when invoking the functions are knwon as **actual parameters**
- Parameter passing in Python follows the semantics of the standard *assignement statement*: when a function is invoked, each identifier (formal parameter) is assigned, in the function's local scope, to the respective actual parameter provided by the caller of the function

# Functions

## Information Passing

- For example:

```
def count(data, target):
. . .
prizes = count(grades, 'A')
```

- Just before the function body is executed, the actual parameter, `grades` and `'A'`, are implicitly assigned to the formal parameter `data` and `target`
- The communication of a return value from the function back to the caller is similarly implemented as an assignment
- An advantage to Python's mechanism for passing information to and from a function is that objects are not copied
- As a consequence, the invocation of a function is efficient, even in the case where a parameter or return value is a complex object

# Functions

## Default Parameter Values

- Python provides means for functions to support more than one possible calling signature (polimorphism)
- Most notably, functions can declare one or more default values for parameters, thereby allowing the caller to invoke a function with varying numbers of actual parameters
- Suppose a function is declared with signature `def foo(a, b=15, c=27)`
- There are three parameters, two of which offer default values
- A caller can send three actual parameters, e.g. `foo(5, 20, 25)`, in which case the default values are not used
- If the caller sends one parameter, `foo(4)`, the function will execute with parameter values $a = 4, b = 15, c = 27$

| Calling Syntax | Description |
|---|---|
| `abs` | Return the absolute value of a number |
| `all(iterable)` | Return `True` if `bool(e)` is `True` for each element e |
| `any(iterable)` | Return `True` if `bool(e)` is `True` for at least one element e |
| `chr(integer)` | Return a one-character string with the given Unicode code point |
| `divmod(x, y)` | Return $(x//y, x\%y)$ as tuple, if `x` and `y` are integers |
| `hash(obj)` | Return an integer hash value for the object |

| Calling Syntax | Description |
|---|---|
| `id(obj)` | Return the unique integer serving as an "identity" for the object |
| `input(prompt)` | Return a string from stdin; the prompt is optional |
| `isinstance(obj, cls)` | Determine if `obj` is an instance of the class |
| `iter(iterable)` | Return a new iterator object for the parameter |
| `len(iterable)` | Return the number of elements in the given iteration |

| Calling Syntax | Description |
|---|---|
| `map(f, iter1, iter2, ...)` | Return an iterator yelding the result of function calls `f(e1, e2, ...)` for respective elements $e1 \in iter1, e2 \in iter2, ...$ |
| `max(iterable)` | Return the largest element of the given iteration |
| `max(a, b, c, ...)` | Return the largest of the arguments |
| `min(a, b, c, ...)` | Return the smallest of the arguments |
| `next(iterator)` | Return the next element reported by the iterator |

| Calling Syntax | Description |
|---|---|
| `open(filename, mode)` | Open a file with the given name and access mode |
| `ord(char)` | Return the Unicode code point of the given character |
| `pow(x, y)` | Return the value $x^y$ (as an integer if $x$ and $y$ are integers) |
| `pow(x, y, z)` | Return the value $x^y$ mod z as an integer |
| `print(obj1, obj2, ...)` | Print the arguments, with separating spaces and trailing newline |

| Calling Syntax | Description |
|---|---|
| `range(stop)` | Construct an iteration of values $0, 1, \ldots, stop - 1$ |
| `range(start, stop)` | Construct an iteration of values $start, start + 1, \ldots, stop - 1$ |
| `range(start, stop, step)` | Construct an iteration of values $start + step, start + 2 \cdot step, \ldots$ |
| `reversed(sequence)` | Return an iteration of the sequence in reverse |
| `round(x)` | Return the nearest `int` value |
| `round(x, k)` | Return the value rounded to the nearest $10^{-k}$ |
| `sorted(iterable)` | Return a list containing elements of the iterable in sorted order |

| Calling Syntax | Description |
|---|---|
| `sum(iterable)` | Return the sum of the elements in the iterable (must be numeric) |
| `type(obj)` | Return the class to which the instance `obj` belongs |

# Recursive functions

- A recursive function is a function that calls itself

```
def countdown(n):
    if n <= 0:
        print("Go!")
    else:
        print(n)
        countdown(n-1)
```

- How many times does it invoke itself?
- Until the base condition is reached

# Recursive functions

## Factorial function

- The factorial of a positive number $n$, usually denoted as $n!$, is defined as:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{se } n \geq 1 \end{cases}$$

# Recursive functions

## Factorial function

- The factorial can be defined as a recursive function
- Given that, by definition, $n! = n \cdot (n-1)!$
- We may write:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n \geq 1 \end{cases}$$

- Note that this kind of definition is common to other recursive funtions
- Usually, in the definition of a recursive functions we have one or more base conditions
- In this case the only base condition is that $n = 0$

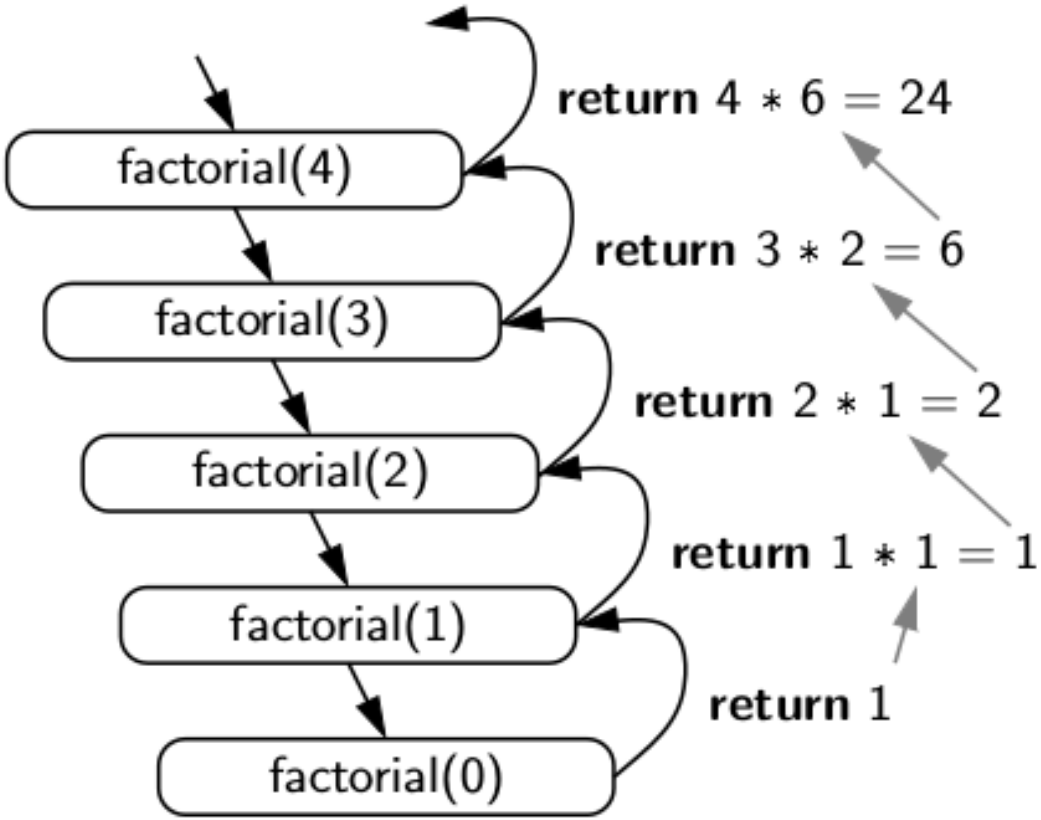
# Recursive functions

## Factorial function

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

# Recursive functions

## Factorial function



# Simple Input and Output

## The `print` Function

- The built-in function `print` is used to generate standard output to the console
- It prints an arbitrary sequence of arguments, separated by space and followed by a trailing newline character. For example: `print(x, y)`
- The `print` function can be customized as follows:
  - By default, a separating space is inserted into the output between each pair of arguments. The separator can be customized by providing a desired separating string. For example: `print(x, y, sep='--')`
  - By default, a trailing newline is output after the final argument. An alternative trailing string can be designated using the keyword end. For example: `print(x, y, sep=':', end='.')`
  - By default, the output is sent to the standard console. Output can be directed to a file by indicating an output file stream using `file` as a keyword parameter.

# Simple Input and Output

## The `input` Function

- The built-in function `input` is used to display a prompt (if given as an optional parameter) and wait until the user enters some sequence of characters
- The formal return value of the function is the string of characters that were entered strictly before the return key
- When reading a numeric input, it must be used the `input` function to get the string of characters and then the `int` or `float` syntax to construct the numeric value that the character string represents

```
age = int(input("Enter your age in years: "))
```

# Simple Input and Output

## The `input` Function

- Because `input` returns a string as its result, use of that function can be combined with the functionality of the string class
- For example, if multiple pieces of information is entered on the same line, the `split` method can be invoked:

```
reply = input("Enter x and y, sep. by spaces: ")
pieces = reply.split()
x = float(pieces[0])
y = float(pieces[1])
```

# Simple Input and Output

## Files

- Files are typically accessed in Python beginning with a call to a built-in function, named `open`, that returns a proxy for future interactions with the underlying file
- For example, the command

```
fp = open('sample.txt')
```

- attempts to open a file named `sample.txt` returning a proxy that allows read-only access to the file

# Simple Input and Output

## Files

- The `open` function accepts a second parameter that determines the access mode
- The default mode is ‘r’ (for reading); other common modes are ‘w’ (writing) or ‘a’ (append)
- It is also possible to work with binary files using access modes such as ‘rb’ or ‘wb’
- When processing a file, the proxy maintains a current position within the file as an offse from the beginning, measured in number of bytes
- When opening a file with mode ‘r’ or ‘w’, the position is initially 0, if opened in append mode, the position is initially at the end of the file
- The syntax `fp.close()` closes the file associated with proxy `fp` ensuring that any written contents are saved

| Calling Syntax | Description |
|---|---|
| `fp.read()` | Return the (remaining) content of a readable file as a string |
| `fp.read(k)` | Return the next $k$ bytes of a readable file as a string |
| `fp.readline()` | Return the current line of a readable file as a string |
| `fp.readlines()` | Return all lines of a readable file as a string |
| `for line in fp:` | Iterate all lines of a readable file |
| `fp.seek(k)` | Change the current position to be at the $k^{th}$ byte of the file |
| `fp.tell()` | Return the current position, measured as byte-offset from the start |

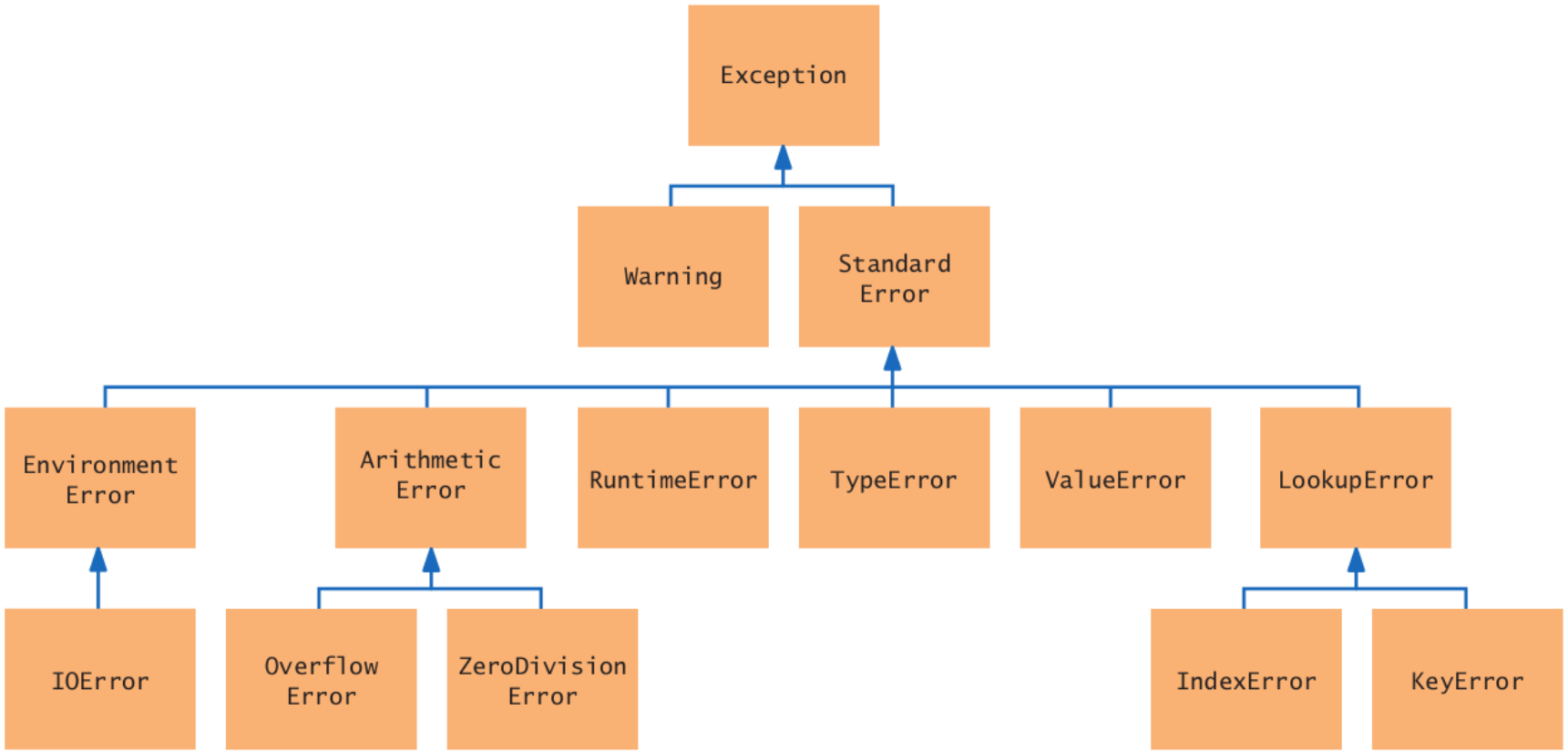
| Calling Syntax | Description |
|---|---|
| `fp.tell()` | Return the current position, measured as byte-offset from the start |
| `fp.write(string)` | Write given string at current position of the writable file |
| `fp.writelines(seq)` | Write each of the strings of the given sequence at the current position of the writable file. This command does not insert *newlines*, beyond those embedded in the strings |
| `print(..., file=fp)` | Redirect output of `print` function to the file |

# Exception Handling

- Exceptions are unexpected events that occur during the execution of a program
- An exception may result from a logical error or an unanticipated situation
- In Python **exceptions** are objects that are **raised** by code that encounters an unexpected circumstance
- The Python interpreter can also raise an exception should it encounter an unexpected condition (e.g., running out of memory)
- A raised error may be **caught** by a surrounding context that *handles* the exception in an appropriate fashion
- If uncaught, an exception causes the interpreter to stop executing the program and to report an appropriate message to the console

# Exception Handling

## Common Exception Types



# Exception Handling

## Common Exception Types

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
```

# Exception Handling

## Common Exception Types

```
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
```

# Exception Handling

## Common Exception Types

```
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

# Exception Handling

## Common Exception Types

| Class | Description |
|---|---|
| `Exception` | A base class for most error types |
| `AttributeError` | Raised by syntax `obj.foo` if `obj` has no member named `foo` |
| `EOFError` | Raised if "end of file" reached for console or file input |
| `IOError` | Raised upon failure of I/O operation |
| `IndexError` | Raised if index to sequence is out of bounds |
| `KeyError` | Raised if nonexistent key requested for se or dictionary |
| `KeyboardInterrupt` | Raised if CTRL-C is typed while program is executing |

# Exception Handling

## Common Exception Types

| Class | Description |
|---|---|
| `NameError` | Raised if nonexistent identifier used |
| `StopIteration` | Raised by `next(iterator)` if no element |
| `TypeError` | Raised when wrong type of parameter is sent to a function |
| `ValueError` | Raised when parameter has invalid value (e.g., `sqrt(-1)`) |
| `ZeroDivisionError` | Raised when any division operator used with 0 as divisor |

# Exception Handling

## Raising an Exception

An exception is thrown by executing the **`raise`** statement, for example:

```
raise ValueError('x cannot be negative')
```

- A newly created instance of the `ValueError` class is raised, with the error message serving as a parameter of the constructor
- If this exception is not caught within the body of the function, the execution of the function immediately ceases and the exception is propagated to the calling context

# Exception Handling

## Raising an Exception

When checking the validity of parameters sent to a function, it should be first verified that a parameter is of an appropriate type. For example:

```
def sqrt(x):
  if not isinstance(x, (int, float)):
    raise TypeError('x must be numeric')
  elif x < 0:
    raise ValueError('x cannot be negative')
    . . .
```

`isinstance(obj, cls)` returns `True` if the object `obj` is an instance of class `cls`, or any subclass of that type

# Exception Handling

## Raising an Exception

A rigorous error-checking could be written as follows:

```
def sum(values):
  if not isinstance(values, collection.Iterable):
    raise TypeError('parameter must be an iterable type'
        )
    total = 0
    for v in values:
      if not isinstance(v, (int, float)):
        raise TypeEror('elements must be numeric')
        total = total + v
    return total
```

A more direct and clear implementation can be written as follows:

```
def sum(values):
  total = 0
  for v in values:
    total = total + v
  return total
```

It should be noted that this simple implementation performs exactly like the Python built-in version of the function.

# Exception Handling

## Raising an Exception

- Even without the explicit checks, appropriate exceptions are raised naturally by the code
- In particular, if `values` is not an iterable type, the attempt to use a for-loop syntax raises a `TypeError` reporting that the object is not iterable
- In the case when a user send an iterable type that includes a nonnumerical element, such as `sum([3.14, 'ciao'])`, a `TypeError` is naturally raised and the following error message is returned:

```
unsupported operand type(s) for +: 'float' and 'str'
```

# Exception Handling

## Catching an Exception

First philosophy: *Look before you leap*
or:
avoid the possibility of an exception being raised through the use of proactive conditional test

```
if y != 0:
  ratio = x / y
else:
  . . .
```

# Exception Handling

## Catching an Exception

Second philosophy: *it is easier to ask for forgiveness than it is to get permission*
or:
we need not spend extra execution time safeguarding every possible exception

```
try:
  ratio = x / y
except ZeroDivisionError
  . . .
```

# Exception Handling

## Catching an Exception

Another form of `try ... except` syntax:

```
try:
  fp = open('sample.txt')
except IOError as e:
  print('Unable to open the file: ', e)
```

In this case the name `e` denotes the instance of the exception that was thrown and printing it causes a detailed error message to be displayed

# Exception Handling

## Catching an Exception

The `try ... except` syntax supports more than one type of exception, for example:

```
age = -1
while age <= 0:
  try:
    age = int(input('Enter your age in years: '))
    if age <= 0:
      print('Your age must be positive')
  except (ValueError, EOFError):
    print('Invalid response')
```

The tuple `(ValueError, EOFError)` is used to designate the types of error that we want to catch with the `except` clause

# Exception Handling

## Catching an Exception

It is possible to provide different responses to different types of errors, by using two or more `except` clauses:

```
age = -1
while age <= 0:
  try:
    age = int(input('Enter your age in years: '))
    if age <= 0:
      print('Your age must be positive')
  except ValueError:
    print('That is an invalid age specification')
  except EOFError:
    print('There was an unexpected error reading input')
    raise
```

# Exception Handling

## Catching an Exception

- In this implementation there are two separate `except` clauses for `ValueError` and `EOFError` situations
- The clause for handling the `EOFError` case contains the `raise` statement to re-raise the same exception that is currently being handled
- The `raise` statement allows to interrupt the `while` loop and propagate the exception upward

# Exception Handling

## Catching an Exception

- It is possible to have a final `except` clause without any identified error types, to catch any other exceptions that occurred
- This possibility should be used rarely, since it is difficult to suggest how to handle an error of an unknown type
- A `try` statement can have a `finally` clause, with a body of code that will always be executed in the standard or exceptional cases
- This block is used for cleanup work, for example closing open files

# Iterators and Generators

The `for` syntax

```
for element in iterable:
```

applies for a number of basic container types that qualify as iterable types (list, tuple, set)

A string can produce an iteration of its characters, a dictionary can produce an iteration of its keys, and a file can produce an iteration of its lines

# Iterators and Generators

In Python, the mechanism for iteration is based upon the following conventions:

- An **iterator** is an object that manages an iteration through a series of values. If `i` identifies an iterator object, then each call to the built-in function, `next(i)`, produces a subsequent element from the underlying series with a `StopIteration` exception raised to indicate that there are no firther elements
- An **iterable** is an object `obj` that produces an *iterator* via the syntax `iter(obj)`

# Iterators and Generators

According to these definitions, an instance of a list is an iterable, but not an iterator

With `data=[1, 2, 4, 8]` it is not legal to call `next(data)`

```
In [4]: data = [1,2,4,8]

In [5]: i = iter(data)

In [6]: i
Out[6]: <list_iterator at 0x7f5298b27240>

In [7]: next(i)
Out[7]: 1

In [8]: next(i)
Out[8]: 2

In [9]: next(i)
Out[9]: 4

In [10]: next(i)
Out[10]: 8

In [11]: next(i)
---------------------------------------------------------------------
---
```

# Iterators and Generators

- The for-loop syntax in Python simply automates this process, creating an iterator for the given iterable, and then repeatedly calling for the next element until catching the `StopIteration` exception
- More generally, it is possible to create multiple iterators based upon the same iterable object, with each iterator maintaining its own state of progress
- Each iterator will maintain a current index into the original list, representing the next element to be reported. Therefore, if the contents of the original list are modified after the creation of the iterator (but before the iteration is complete), the iterator will report the *updated* contents of the list

# Iterators and Generators

```
class PowTwo:
  def __init__(self, max = 0):
    self.max = max

  def __iter__(self):
    self.n = 0
    return self

  def __next__(self):
    if self.n <= self.max:
      result = 2**self.n
      self.n += 1
      return result
    else:
      raise StopIteration
```

# Iterators and Generators

- The most convenient technique for creating iterators in Python is through the use of **generators**
- A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a `yield` statement is executed to indicate each element of the series

# Iterators and Generators

— A traditional function:

```
def factors(n):
  results = []
  for k in range(1, n+1):
    if n%k == 0:
      results.append(k)
  return results
```

— A generator:

```
def factors(n):
  for k in range(1, n+1):
    if n%k == 0:
      yield k
```

# Iterators and Generators

— An example:

```
def factors(n):
  for k in range(1,n+1):
    if n % k == 0:
      yield k

n = 100

for value in factors(n):
  print(value)
```

# Iterators and Generators

```
|giacomo : ~/Dropbox/Teach/AACM/Lectures/examples >more generator.py
def factors(n):
  for k in range(1,n+1):
    if n % k == 0:
      yield k

n = 100

for value in factors(n):
  print(value)
|giacomo : ~/Dropbox/Teach/AACM/Lectures/examples >python3 generator.py
1
2
4
5
10
20
25
50
100
```

# Iterators and Generators

— Another example:

```
def gen():
  print("Starting...")
  yield "A"
  print("continuing ...")
  yield "B"
  print("Ending")

for element in gen():
  print("Element in gen: ", element)
```

— Python supports a *conditional expression* syntax that can replace a simple control structure:

```
expr1 if condition else expr2
```

This compound expression evaluates to `expr1` if `condition` is true, otherwise evaluates to `expr2`

# Additional Conveniences

## Conditional Expressions

For example:

```
if n >= 0:
  param = n
else:
  param = -n
result = foo(param)
```

can be written as:

```
param = n if n >= 0 else -n
result = foo(param)
```

Or, even more concisely:

```
result = foo(n if n >= 0 else -n)
```

- The mere shortening of source code is advantageous because it avoids the distraction of cumbersome control structure
- Conditional expressions, however, should be used only to improve the readability of the code

# Additional Conveniences

## Comprehension Syntax

- A very common programming task is to produce a series of values based upon the processing of another series
- Often this task can be accomplished using what is known as *comprehension syntax*
- The *list comprehension* has the following general form:

```
expression for value in iterable if condition
```

- Both `expression` and `condition` may depend on `value`
- The if-clause is optional

# Additional Conveniences

## Comprehension Syntax

The list comprehension:

```
expression for value in iterable if condition
```

is logically equivalent to:

```
result = []
for value in iterable:
  if condition:
    result.append(expression)
```

# Additional Conveniences

## Comprehension Syntax

For example, the traditional fragment of code:

```
squares = []
for k in range(1,n+1):
  square.append(k*k)
```

can be replaced with:

```
squares = [k*k for k in range(1,n+1)]
```

# Additional Conveniences

## Comprehension Syntax

— List comprehension:

```
[k*k for k in range(1,n+1)]
```

— Set comprehension:

```
{k*k for k in range(1,n+1)}
```

— Generator comprehension:

```
(k*k for k in range(1,n+1))
```

— Dictionary comprehension:

```
{k: k*k for k in range(1,n+1)}
```

# Additional Conveniences

## Packing and Unpacking of Sequences

— Automatic packing

If a series of comma-separated expressions are given, they will be treated as a single tuple, even if no enclosing parentheses are provided. For example:

```
data = 2, 4, 6, 8
```

implies that the identifier `data` is assigned to the tuple $(2, 4, 6, 8)$

Example:

```
return x, y
```

— Automatic unpacking

As a dual to the packing behavior, Python can automatically **unpack** a sequence. For example:

```
x, y, z, t = range(5, 9)
```

```
x,y,z,t = range(5,9)
```

```
z
```

```
7
```

# Additional Conveniences

## Packing and Unpacking of Sequences

```
x,y,z,t,u = range(5,9)
```

```
---------------------------------------------------
---------------------------------
ValueError                                Trac
eback (most recent call last)
<ipython-input-1-374f81002118> in <module>()
----> 1 x,y,z,t,u = range(5,9)

ValueError: not enough values to unpack (expec
ted 5, got 4)
```

# Additional Conveniences

## Packing and Unpacking of Sequences

This technique can be used to unpack tuples returned by a function. For example

```
quotient, remainder = divmod(a, b)
```

This syntax can also be used in the context of a for loop, when iterating over a sequence of iterables

```
for x, y in [(7, 2), (5, 8), (6, 4)]:
```

# Additional Conveniences

## Simultaneous Assignments

The combination of automatic packing and unpacking forms a technique known as **simultaneous assignment**:

```
x, y, z = 6, 2, 5
```

Actually, the rhs of this assignment is automatically packed into a tuple, and then automatically unpacked with its elements assigned to three identifiers on the lhs

When using a simultaneous assignment, all of the expression are evaluated on the rhs before any of the assignments are made to the lhs variables. For this reason, it provides a convenient means for swapping the values associated with two variables:

```
i, j = j, i
```

# Additional Conveniences

## Simultaneous Assignments

For example:

```
def fibonacci():
  a, b = 0, 1
  while True:
    yield a
    a, b = b, a + b
```

Instead of the more traditional generator:

```
def fibonacci():
  a = 0
  b = 1
  while True:
    yield a
    future = a + b
    a = b
    b = future
```

# Scopes and Namespaces

- When computing a sum with the syntax `x + y`, the names `x` and `y` must have been previously associated with objects that serve as values, otherwise a `NameError` will be raised if no such definitions are found. Determining the value associated with an identifier is known as **name resolution**
- Whenever an identifier is assigned to a value, that definition is made with a specific **scope**. Top-level assignments are made in what is known as **global scope**, whereas assignments made within the body of a function typically have **local scope** to that function call
- Each distinct scope in Python is represented using an abstraction known as **namespace**, which manages all the identifiers currently defined in a given scope

# Modules

The standard Python distribution includes values, functions and classes that are organized in additional libraries known as **modules** that can be imported from within a program

`import` statement loads definitions from a module into the current namespace. One form uses the following syntax:

```
from math import pi, sqrt
```

The effect of this statement is to import `pi` and `sqrt` into the current namespace, allowing direct use of the identifier `pi` or a call of the function `sqrt(x)`

# Modules

If there are many definitions from the same module, an asterisk may be used as a wild card:

```
from math import *
```

Alternatively, it is possible to import the module itself, using the following syntax:

```
import math
```

Formally, this adds the identifier `math` to the current namespace, with the module as its value. Once imported, individual definitions from the module can be accessed using a fully-qualified name, such as `math.pi` or `math.sqrt(x)`

# Modules

## Creating a New Module

- To create a new module, it is necessary to put the relevant definitions in a file named with a `.py` suffix
- Those definitions can be imported from any other `.py` file within the same project directory
- If, for example, one has to call the function `count`, which is contained into a file named `myfunctions.py`, the function can be imported using the syntax

```
from myfunctions import count
```

| Module Name | Description |
|---|---|
| array | Provides compact array storage for primitive types |
| collections | Defines additional data structures and abstract base classes involving collections of objects |
| copy | Defines general functions for making copies of objects |
| heapq | Provides heap-based priority queue functions |
| math | Defines common mathematical constants and functions |

(to be continued)

(continued)

| Module Name | Description |
|---|---|
| os | Provides support for interactions with the operating system |
| random | Provides random number generation |
| re | Provides support for processing regular expressions |
| sys | Provides additional level of interaction with the Python interpreter |
| time | Provides support for measuring time, or delaying a program |

The running time of an algorithm (without performing experiments) can be analyzed performing a high-level description of the algorithm. To this purpose, a set of **primitive operations** can be defined as follows:

- Assigning an identifier to an object
- Determining the object associated with an identifier
- Performing an arithmetic operation
- Comparing two numbers
- Accessing a single element of a Python list by index
- Calling a function
- Returning from a function

# Algorithm Analysis

## Primitive Operations

- A primitive operation corresponds to a low-level instruction with an execution time that is constant
- Instead of trying to determine the specific execution time of each primitive operation, the number of primitive operations will be counted and this number will be used as a measure of the running time of the algorithm
- To capture the order of growth of an algorithm's running time, a function $f(n)$ will be associated, that characterizes the number of primitive operations that are performed as a function of the input size $n$

- In algorithm analysis, one has to focus on the growth rate of the running time as a function of the input size $n$
- The running times of algorithms are characterized using functions that map the size of the input $n$ to values that correspond to the main factor that determines the growth rate in terms of $n$
- In other words, one can perform an analysis of an algorithm by estimating the number of primitive operations executed up to a constant factor, rather than getting involved in language-specific or hardware-specific analysis of the exact number of operations

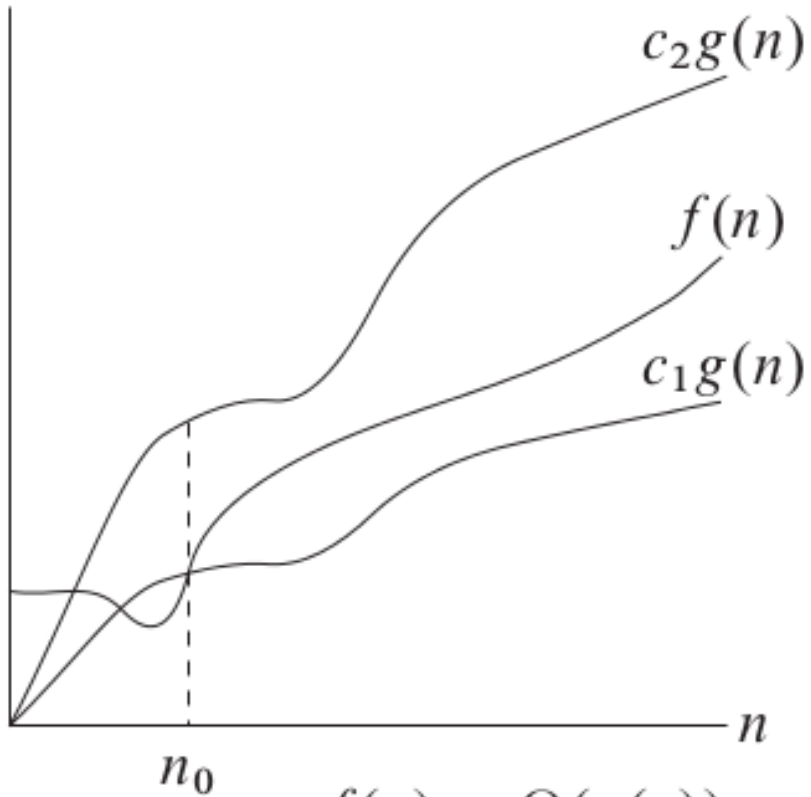
For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \exists \text{ positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0\}$$

A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants $c_1$ and $c_2$ such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large $n$

# Algorithm Analysis

## Θ-notation




$f(n) = \Theta(g(n))$

From the definition of $\Theta(g(n))$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0\}$$

follows that every $f(n) = \Theta(g(n))$ must be aymptotically nonnegative, that is, $f(n)$ be nonnegative whenever $n$ is sufficiently large.

Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty.

As an example, consider any quadratic function

$$f(n) = an^2 + bn + c$$

where $a, b, c$ are constants and $a > 0$.
The lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large $n$.

Disregarding the lower-order terms and ignoring the constants yelds $f(n) = \Theta(n^2)$

More generally, for any polynomial

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

where the $a_i$ are constants and $a_d > 0$, we have $p(n) = \Theta(n^d)$

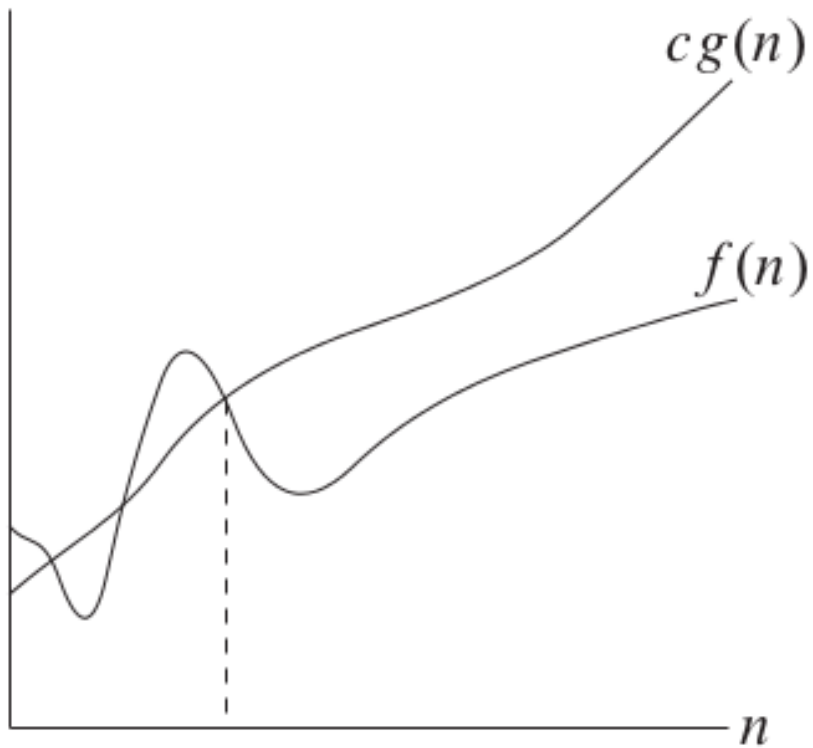
Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$ or $\Theta(1)$

- The $\Theta$-notation asymptotically bounds a function from above and below
- When we have only an asymptotic upper bound, we use $O$-notation
- For a given function $g(n)$ we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0\}$$

# Algorithm Analysis

## $O$-notation




$f(n) = O(g(n))$

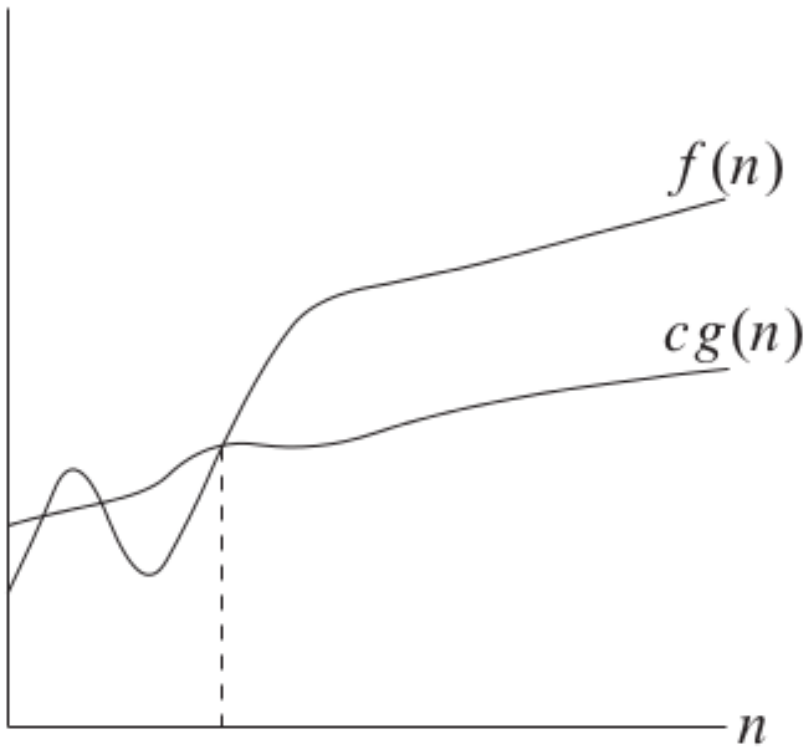
- We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is member of the set $O(g(n))$
- Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since $\Theta$-notation is a stronger notion than $o$-notation
- Since $O$-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on every input

- Just as $O$-notation provides an asymptotic **upper** bound on a function, Ω-notation provides an **asymptotic lower bound**
- For a given function $g(n)$ we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0\}$$

# Algorithm Analysis

## Ω-notation




$$f(n) = \Omega(g(n))$$

From the definition of the asymptotic notations, this important theorem follows

**Theorem**

*For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Given a sequence $S$ consisting of $n$ numbers, we want to compute a sequence $A$ such that $A[j]$ is the average of elements $S[0], S[1], \ldots, S[j]$ for $n = 0, 1, \ldots, n-1$, that is

$$A[j] = \frac{\sum_{i=0}^{j} S[i]}{j+1}$$

# Algorithm Analysis

## An example: prefix averages (first implementation)

```
def prefix_average1(S):
  n = len(S)
  A = [0] * n
  for j in range(n):
    total = 0
    for i in range(j+1):
      total += S[i]
    A[j] = total / (j + 1)
  return A
```

# Algorithm Analysis

## An example: prefix averages (first implementation)

- The statement, `n = len(S)` is executed in constant time (the `list` class maintains an instance variable that records the current length of the list)
- The statement, `A = [0] * n`, causes the creation and initialization of a Python list having length $n$, and with all entries equal to zero. This uses a constant number of primitive operations per element, and thus runs in $O(n)$ time
- The body of the outer loop, controlled by $j$ is executed $n$ times. Therefore statements `total = 0` and `A[j] = total / (j + 1)` are executed $n$ times each. These two statements contribute a number of primitive operations proportional to $n$, that is, $O(n)$ time

- The body of the inner loop, controlled by $i$ is executed $j + 1$ times, depending on the current value of the outer loop counter $j$. Thus, statement `total += S[i]` is executed $1 + 2 + 3 + \cdots + n = n(n+1)/2$ times, which implies that the statement contributes $O(n^2)$ time

The running time of this implementation is given by the sum of three terms: the first and the second terms are $O(n)$, and the third term is $O(n^2)$. Therefore the running time is $O(n^2)$.

# Algorithm Analysis

## An example: prefix averages (second implementation)

```
def prefix_average2(S):
  n = len(S)
  A = [0] * n
  for j in range(n):
    A[j] = sum(S[0:j+1]) / (j + 1)
  return A
```

- The expression `sum(S[0:j+1])` is a function call and its evaluation takes $O(j+1)$ time
- The computation of the slice `S[0:j+1]` also uses $O(j+1)$ time, as it constructs a new list instance for storage

The running time of this implementation is dominated by a series of steps that take time proportional to
$1+2+3+\cdots+n = n(n+1)/2$, and thus $O(n^2)$.

# Algorithm Analysis

## An example: prefix averages (third implementation)

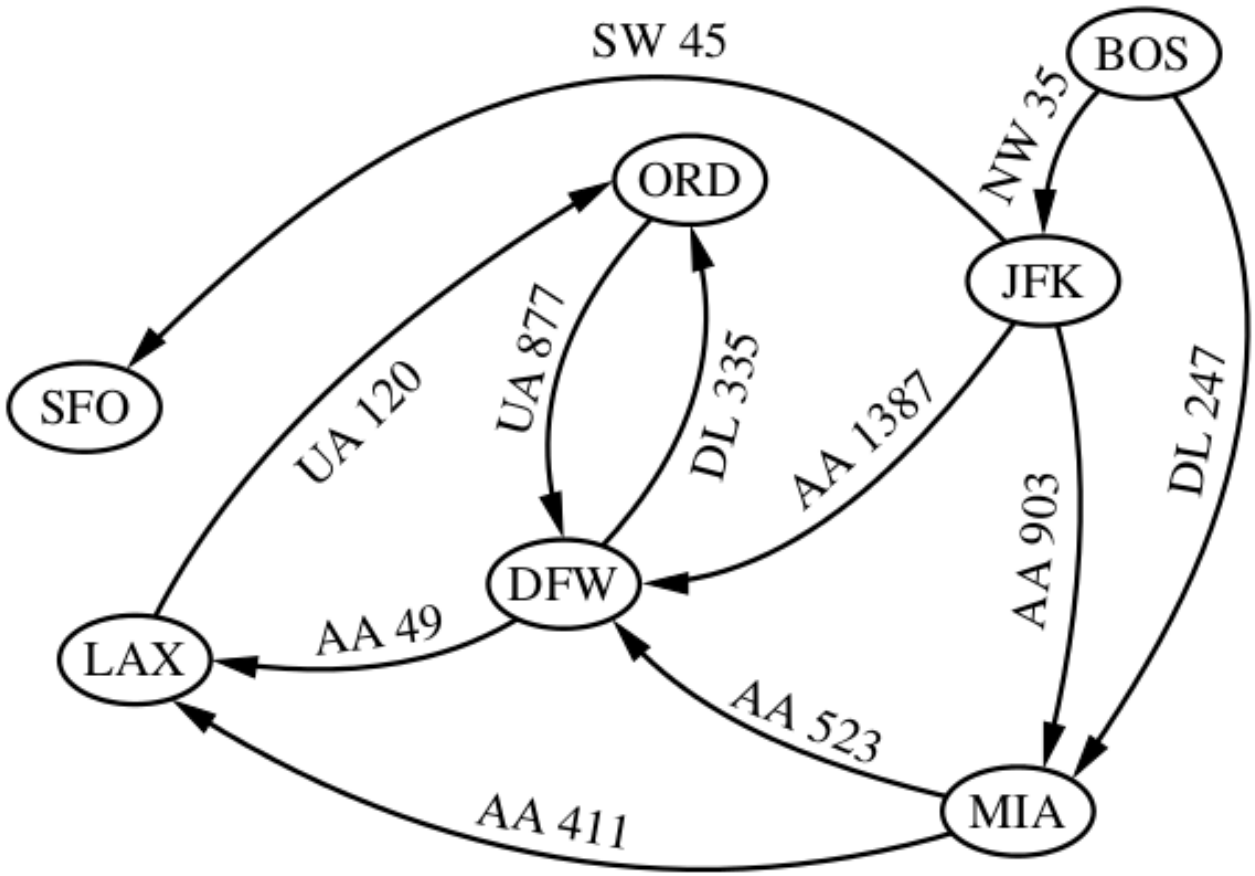
```
def prefix_average3(S):
  n = len(S)
  A = [0] * n
  total = 0
  for j in range(n):
    total += S[j]
    A[j] = total / (j + 1)
  return A
```

- The initialization of variables $n$ and *total* uses $O(1)$ time
- The initialization of the list `A` uses $O(n)$ time
- There is a single loop, controlled by $j$. This contributes a total of $O(n)$ time
- The body of the loop is executed $n$ times, for $j = 0, 1, \ldots, n-1$. Thus, statements `total += S[j]` and `A[j] = total / (j + 1)` are executed $n$ times each: their contribution is $O(n)$

The running time of this implementation is given by the sum of the four terms: the first is $O(1)$ and the remaining are $O(n)$.
Therefore, the running time is $O(n)$ .

- A **graph** is a way of representing relationships that exist between pairs of objects
- In other words, a graph is a set of objects, called *vertices* (or nodes), together with a collection of pairwise connections between them, called *edges*
- A graph $G$ is a set $V$ of vertices and a collection $E$ of edges
- Edges in a graph are either *directed* or *undirected*
  - An edge $(u, v)$ is said directed from $u$ to $v$ if the pair $(u, v)$ is ordered, with $u$ preceding $v$
  - An edge $(u, v)$ is said undirected if the pair $(u, v)$ is not ordered

# Graphs


SW 45
BOS
NW 35
ORD
JFK
SFO
UA 120
UA 877
DL 335
AA 1387
DL 247
AA 903
DFW
AA 49
LAX
AA 523
AA 411
MIA

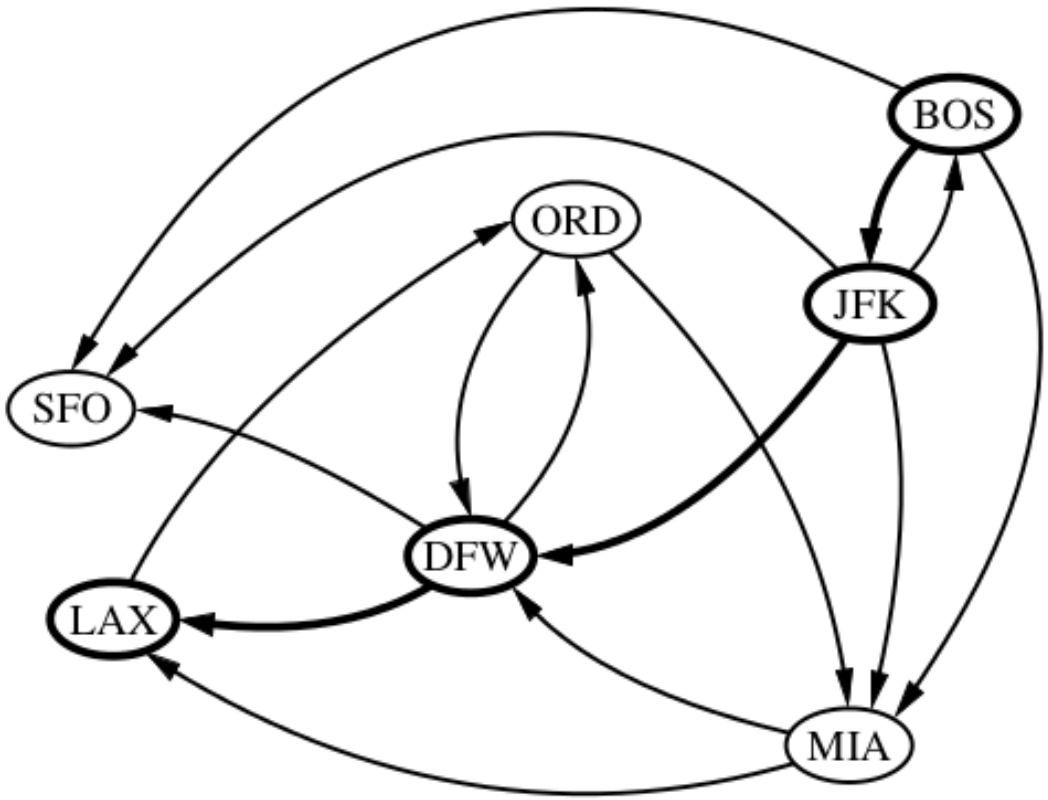
- If all the edges in a graph are undirected, the graph is an *undirected graph*
- Likewise, a *directed graph* (or *digraph*) is a graph whose edges are all directed
- A graph that has both directed and undirected edges is called a *mixed graph*
- The two vertices joined by an edge are called the *end vertices* (or *endpoints*) of the edge
- If an edge is directed, its first endpoint is called *origin* and the other is the *destination* of the edge
- Two edges are called *adjacent* if there is an edge connecting them

- An edge is said to be *incident* to a vertex if the vertex is one of the endpoints of the edge
- The *outgoing edges* of a vertex are the directed edges whose origin is in the vertex
- The *ingoing edges* of a vertex are the directed edges whose destination is in the vertex
- The *degree* of a vertex $v$, denoted $deg(v)$, is the number of incident edges of $v$
- The *in-degree* and *out-degree* of a vertex $v$ is the number of the incoming and outgoing edges of $v$, and are denoted $indeg(v)$ and $outdeg(v)$ respectively

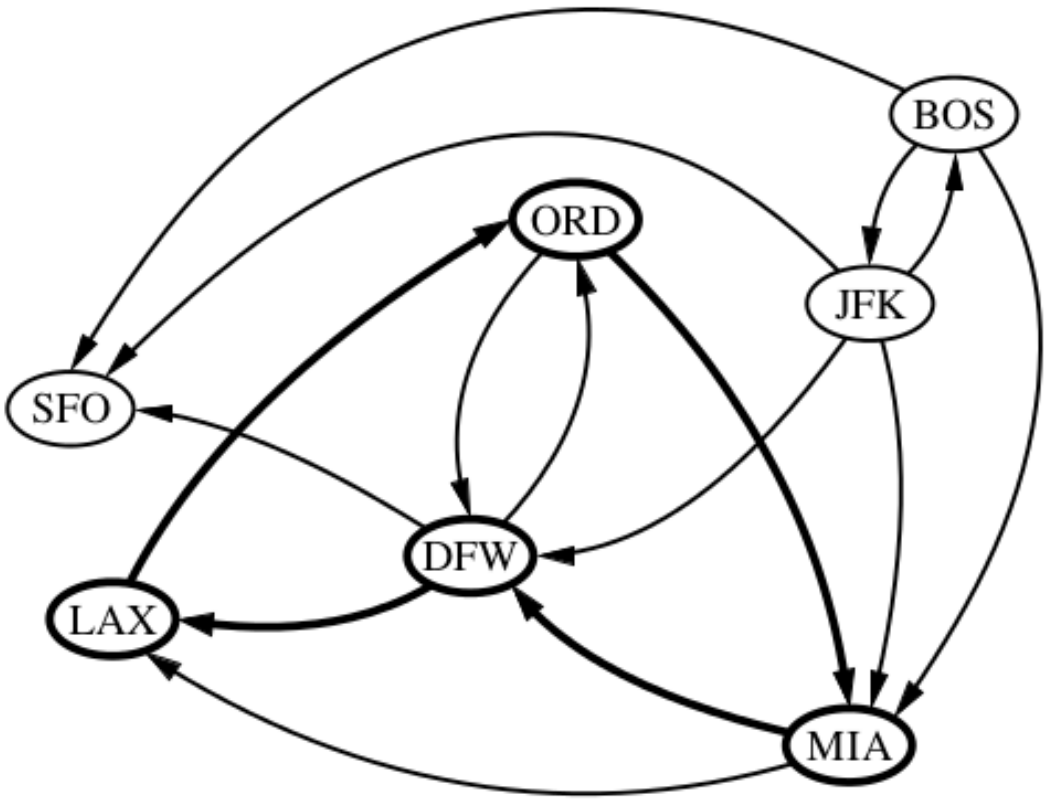
- According to the definition, the group of edges of a graph is referred to as a *collection* (not a *set*), thus allowing two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination
- Such edges are called *parallel edges* or *multiple edges*
- Another special type of edge if one that connects a vertex to itself. In other words, an edge is a *self-loop* if its two endpoints coincide
- Graphs that do not have parallel edges or self-loops are called *simple*

- A *path* is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex
- If the starting and the ending vertices coincide, the path is called *cycle*
- A cycle must include at least one vertex
- A path is called *simple* if each vertex in the path is distinct
- A *directed path* is a path such that all edges are directed are traversed along their direction
- A directed graph is *acyclic* if it has no directed cycles

- Given vertices $u$ and $v$ of a (directed) graph $G$, we say that $u$ *reaches* $v$ if $G$ has a (directed) path from $u$ to $v$
- In an undirected graph the notion of reachability is symmetric
- A graph is *connected* if, for any two vertices, there is a path between them
- A directed graph is *strongly connected* is for any two vertices $u$ and $v$, $u$ reaches $v$ and $v$ reaches $u$




*A directed path from BOS to LAX*




*A directed cycle (ORD; MIA; DFW; LAX; ORD)*

- A *subgraph* of a graph $G$ is a graph $H$ whose vertices and edges are subsets of the vertices and edges of $G$
- A *spanning subgraph* of $G$ is a subgraph of $G$ that contains all the vertices of the graph $G$
- If a graph $G$ is not connected, its maximal connected subgraphs are called the *connected components* of $G$

**Proposition 1**
If $G$ is a graph with $m$ edges and vertex set $V$, then

$$\sum_{v \text{ in } V} deg(v) = 2m$$

**Justification:** An edge $(u, v)$ is counted twice in the summation above; once by its endpoint $u$ and once by its endpoint $v$. Thus, the total contribution of the edges to the degrees of the vertices is twice the number of edges.

**Proposition 2**
If $G$ is a directed graph with $m$ edges and vertex set $V$, then

$$\sum_{v \text{ in } V} indeg(V) = \sum_{v \text{ in } V} outdeg(V) = m$$

**Justification:** In a directed graph, an edge $(u, v)$ contributes one unit to the out-degree of its origin $u$ and one unit to the in-degree of its destination $v$. Thus, the total contribution of the edges to the out-degree (in-degree) of the vertices is equal to the number of edges.

**Proposition 3**
Let $G$ be a simple graph with $n$ vertices and $m$ edges. If $G$ is undirected, then

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

and if $G$ is directed, then

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

**Justification:** Suppose $G$ is undirected. Since no two edges can have the same endpoints and there are no self-loops, the maximum degree of a vertex in $G$ is $n-1$ in this case. Thus, by Proposition 1, $2m \leq n(n-1)$.
Now suppose that $G$ is directed. Since no two edges can have the same origin and destination, and there are no self-loops, the maximum in-degree of a vertex in $G$ is $n-1$ in this case. Thus, by Proposition 2, $m \leq n(n-1)$.

# Graphs Abstract Data Type (ADT)

A graph is a collection of vertices and edges. It can be modeled as a combination of three data types: `Vertex`, `Edge` and `Graph`

- A `Vertex` is an object that stores an arbitrary element provided by the user, and supports a method, `element()`
- An `Edge` stores an associated object, retrieved with the `element()` method
- It also supports the following methods:
  - `endpoints()`, which returns a tuple $(u, v)$ such that vertex $u$ is the origin of the edge and vertex $v$ is the destination. For an undirected graph, the orientation is arbitrary
  - `opposite(v)`, which returns the other endpoint if vertex $v$ is one endpoint of the edge

# Graphs Abstract Data Type (ADT)

A graph can be either directed or undirected. A mixed graph can be represented as a directed graph, modeling edge $\{u, v\}$ as a pair of directed edges $(u, v)$ and $(v, u)$. The `Graph` ADT includes the following methods:

`vertex_count()`
Return the number of vertices of the graph

`vertices()`
Return an iteration of all the vertices of the graph

`edge_count()`
Return the number of edges of the graph

`edges()`
Return an iteration of all the edges of the graph

# Graphs Abstract Data Type (ADT)

`get_edge(u,v)`
Return the edge from vertex $u$ to vertex $v$, if one exists, otherwise return None

`degree(v, out=True)`
For un undirected graph, return the number of edges incident to vertex $v$. For a directed graph, return the number of outgoing (incoming) edges incident to vertex $v$

`incident_edges(v, out=True)`
Return an iteration of all edges incident to vertex $v$. In the case of a directed graph, report outgoing edges by default

# Graphs Abstract Data Type (ADT)

`insert_vertex(x=None)`
Create and return a new `Vertex` storing element $x$

`insert_edge(u, v, x=None)`
Create and return a new `Edge` from vertex $u$ to vertex $v$, storing element $x$

`remove_vertex(v)`
Remove vertex $v$ and all its incident edges from the graph

`remove_edge(e)`
Remove edge $e$ from the graph

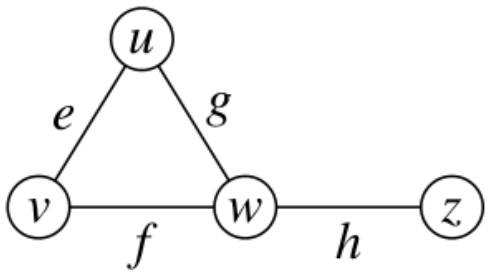
# Data Structures for Graphs

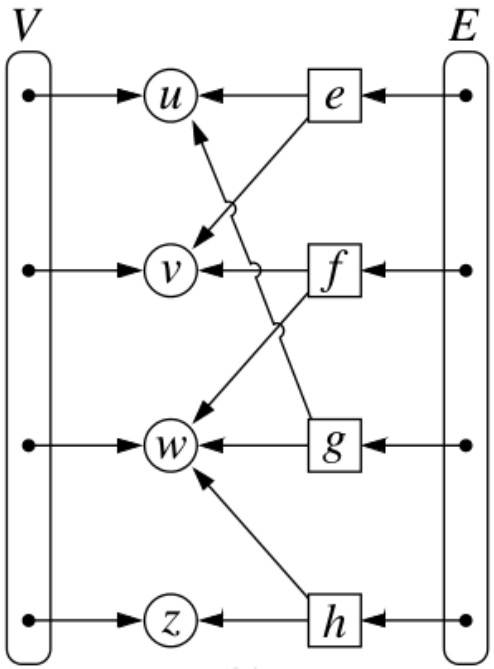
- *Edge list*
- *Adjacency list*
- *Adjacency map*
- *Adjacency matrix*

# Data Structures for Graphs

## Edge List

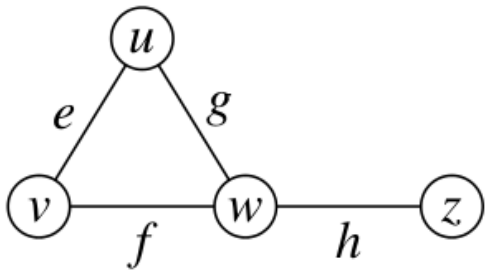
Unordered list of all edges. This is the minimum, since there is no efficient way to locate a particular edge $(u, v)$, or the set of all edges incident to a vertex $v$



# Data Structures for Graphs

## Edge List




In a simpler way:

```
e ==> u - v
f ==> v - w
g ==> w - u
h ==> w - z
```

# Data Structures for Graphs

## Edge List

| Operation | Running Time |
|---|---|
| `vertex_count()` | $O(1)$ |
| `edge_count()` | $O(1)$ |
| `vertices()` | $O(n)$ |
| `edges()` | $O(m)$ |
| `get_edge(u,v)` | $O(m)$ |
| `degree(v)` | $O(m)$ |
| `incident_edges(v)` | $O(m)$ |
| `insert_vertex(v)` | $O(1)$ |
| `remove_vertex(v)` | $O(m)$ |
| `insert_edge(u,v,x)` | $O(1)$ |
| `remove_edge(e)` | $O(1)$ |

# Data Structures for Graphs

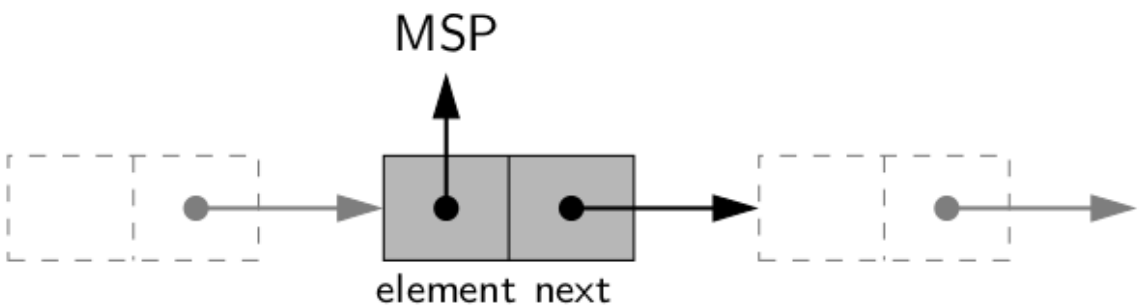
## Edge List: performances

- The edge list structure performs well in terms of reporting the number of vertices or edges, or in producing an iteration of those vertices or edges
- The most significant limitations of an edge list structure are the $O(m)$ running times of methods `get_edge(u,v)`, `degree(v)` and `incident_edges(v)`. This is due to the arrangement of the graph in an unordered list, so that the only way to answer those queries is to inspect all edges
- The update of the graph (insertion of a vertex, or an edge, removal of an edge) requires $O(1)$ time
- The removal of a vertex, on the contrary, requires $O(m)$ time, because all edges incident to the vertex $v$ must also be removed. To locate the incident edges to the vertex, all edges must be examined

# Digression: Linked Lists

## Singly Linked Lists

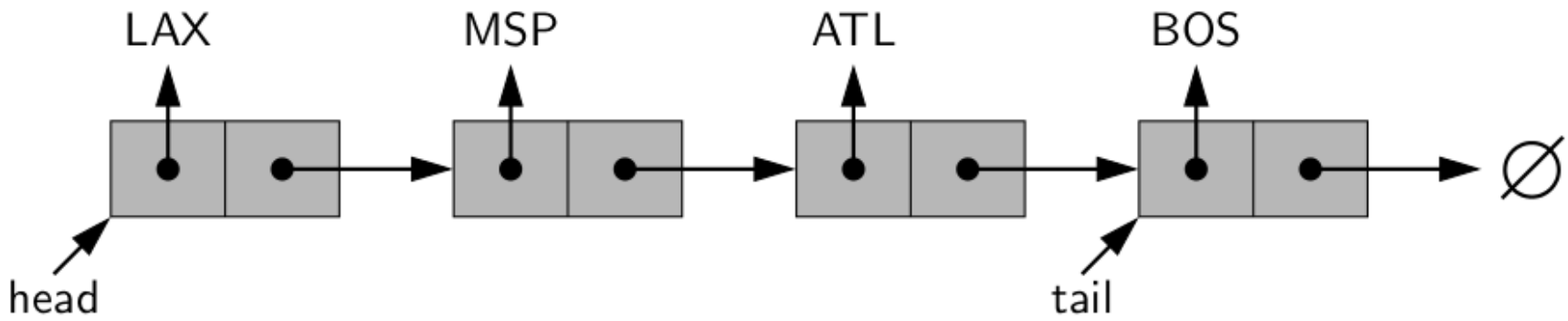
A *singly linked list* is a collection of **nodes** that collectively form a linear sequence. Each node contains a reference to an object that is an element of the sequence as well as a reference to the next node of the list



# Digression: Linked Lists

## Singly Linked Lists

The first and the second node of a linked list are known as the **head** and the **tail** of the list



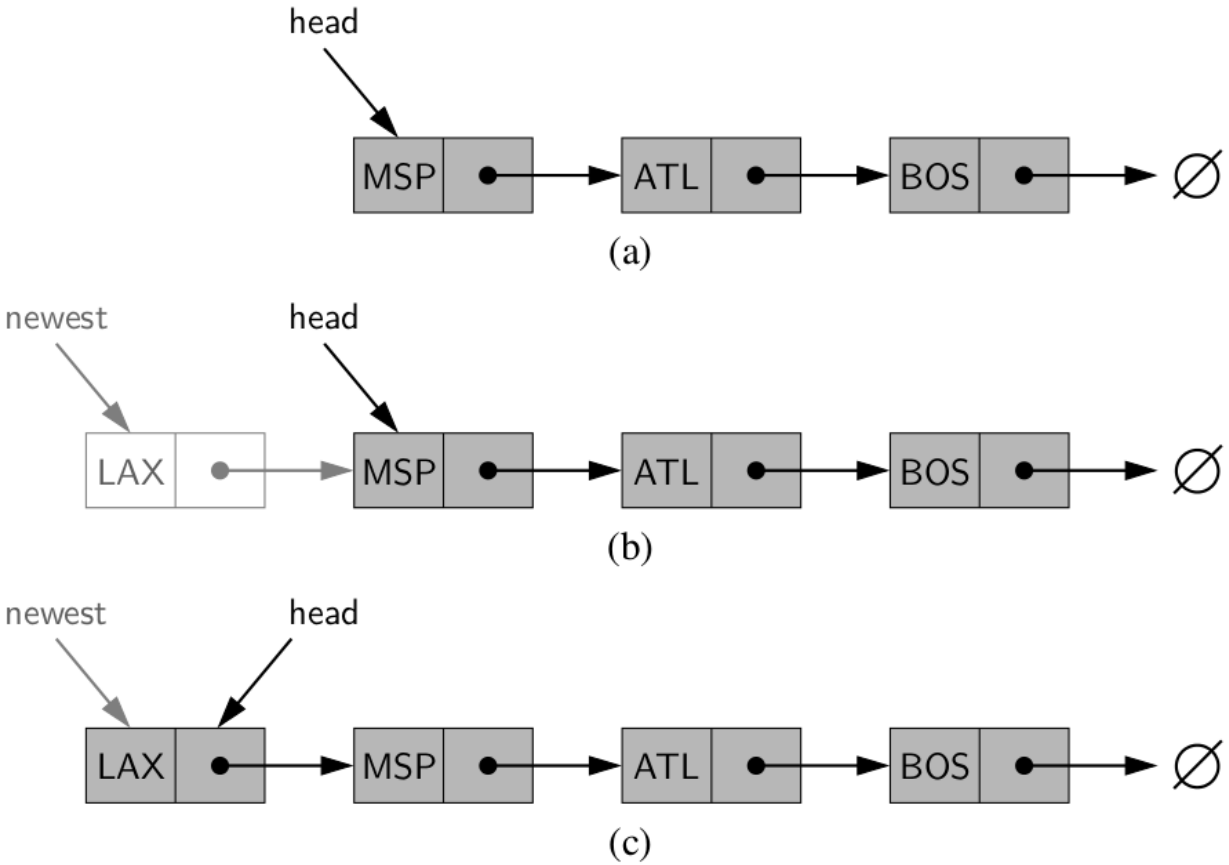

A representation in memory of a linked list relies on the collaboration of many objects:

— Each node is represented as a unique object, with that instance storing a reference to its element and a reference to the next node (or None)

— Another object represents the list as a whole

# Digression: Linked Lists

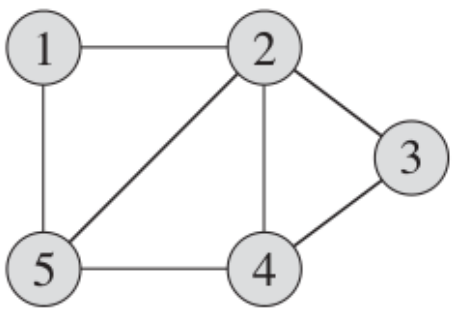
## Inserting an Element at the Head of a Singly Linked Lists

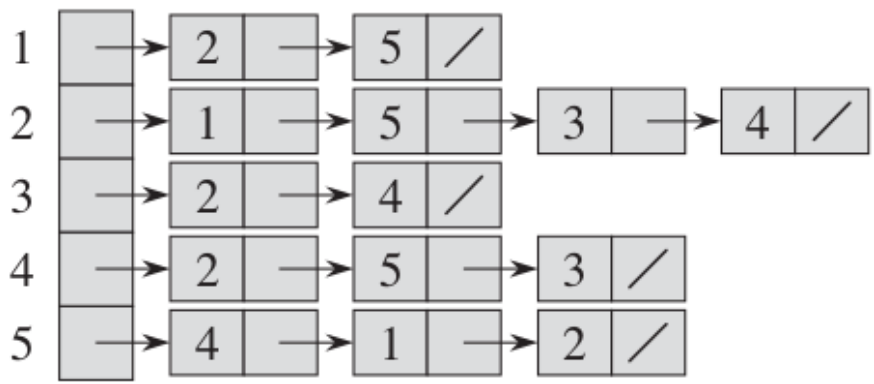


# Data Structures for Graphs

## Adjacency List

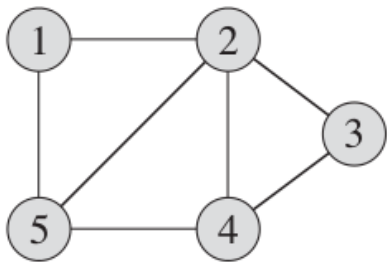
For each vertex is maintained a separate list containing those edges that are incident to the vertex. The complete set of edges can be determined by taking the union of the smaller sets

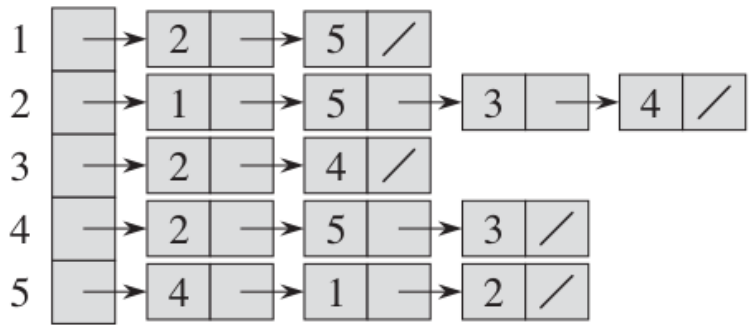


# Data Structures for Graphs

## Adjacency List

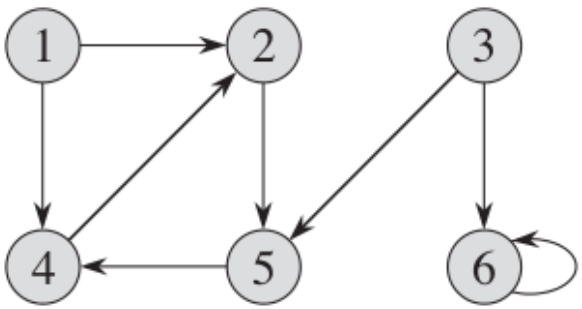


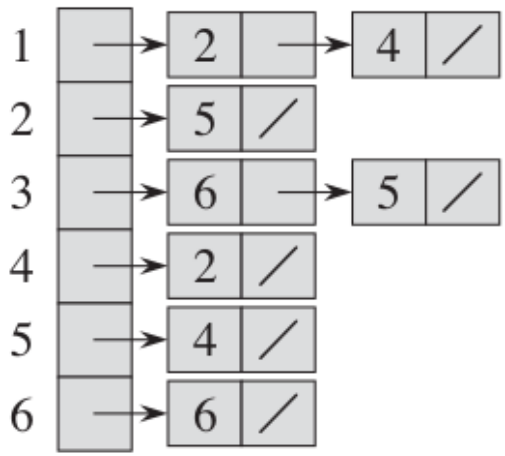




```
graph = {1: set([2, 5]),
         2: set([1, 3, 4, 5]),
         3: set([2, 4]),
         4: set([2, 3, 5]),
         5: set([1, 2, 4])}
```

# Data Structures for Graphs

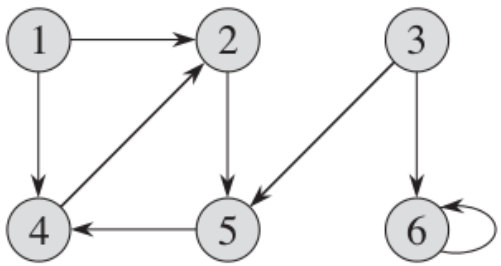
## Adjacency List

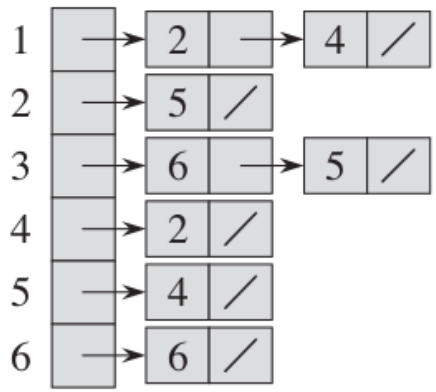


# Data Structures for Graphs

## Adjacency List




```
graph = {1: set([2, 4]),
         2: set([5]),
         3: set([5, 6]),
         4: set([2]),
         5: set([4]),
         6: set([6])}
```

- The **adjacency list** structure groups the edges of a graph by storing them in secondary containers that are associated with each individual vertex
- For each vertex $v$, we maintain a collection $I(v)$, called the *incidence collection* of $v$, whose entries are edges incidento to the vertex $v$.
- In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, $I_{out}(v)$ and $I_{in}(v)$.

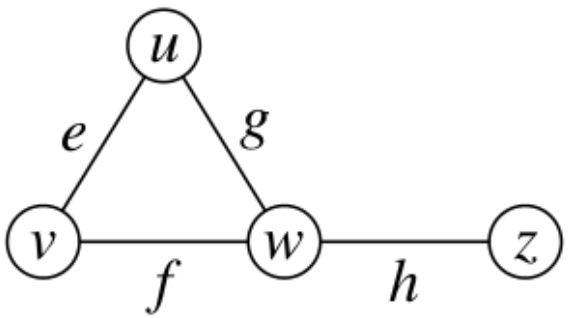
# Data Structures for Graphs

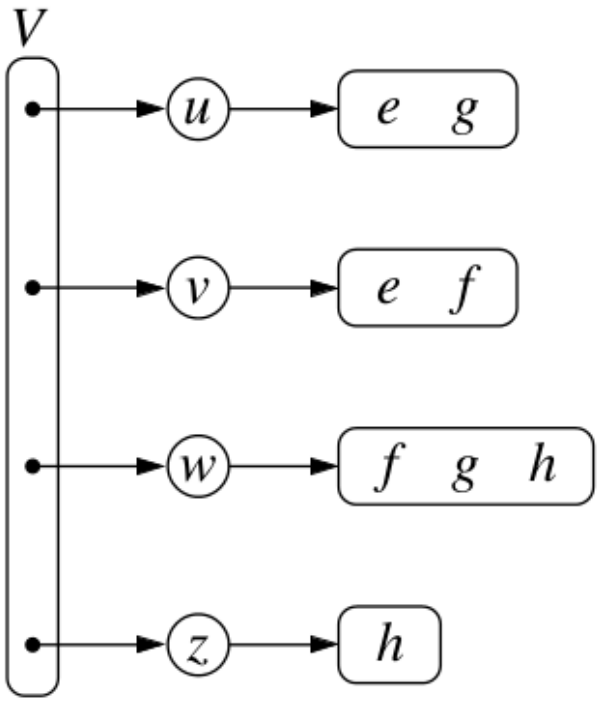
## Adjacency List Structure

- The primary structure for an adjacency list must maintain the collection $V$ of vertices in a way so that we can allocate the secondary structure $I(v)$ for a given vertex $v$ in $O(1)$ time.
- This can be done using a positional list to represent $V$, with each `Vertex` instance maintaining a direct reference to its $I(v)$ incidence collection
- If vertices can be uniquely numbered from 0 to $n-1$, we could use a primary array-based structure to access the appropriate secondary lists
- The benefit of an adjacency list is that the collection $I(v)$ contains exactly those edges that should be reported by the method `incident_edges(v)`. Therefore, we can implement this method by iterating the edges of $I(v)$ in $O(deg(v))$, where $deg(v)$ is the degree of vertex $v$

# Data Structures for Graphs

## Adjacency List Structure



# Data Structures for Graphs

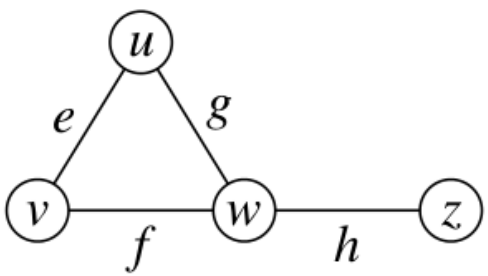
## Adjacency List

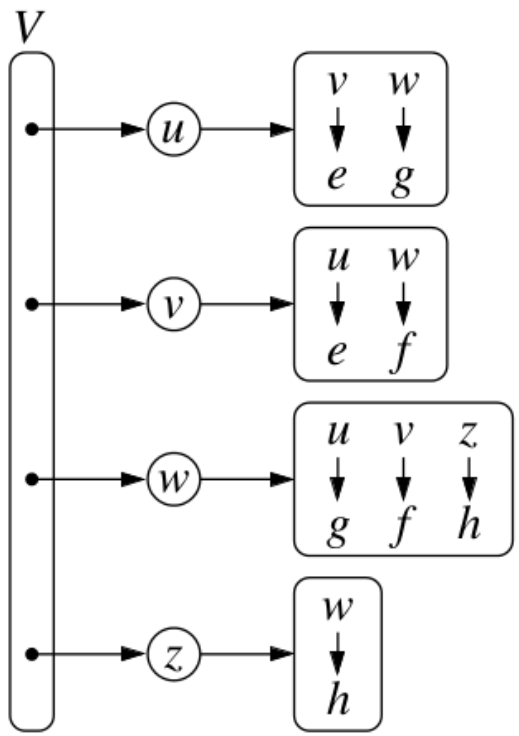
| Operation | Running Time |
|---|---|
| `vertex_count()` | $O(1)$ |
| `edge_count()` | $O(1)$ |
| `vertices()` | $O(n)$ |
| `edges()` | $O(m)$ |
| `get_edge(u,v)` | $O(min(d_u, d_v))$ |
| `degree(v)` | $O(1)$ |
| `incident_edges(v)` | $O(d_v)$ |
| `insert_vertex(v)` | $O(1)$ |
| `remove_vertex(v)` | $O(d_v)$ |
| `insert_edge(u,v,x)` | $O(1)$ |
| `remove_edge(e)` | $O(1)$ |

# Data Structures for Graphs

## Adjacency Map

Very similar to an adjacent list, but the secondary container of all edges incident to a vertex is organized as a map, rather than a list, with the adjacent vertex serving as a key. The access to a specific edge $(u, v)$ is performed in $O(1)$ time



# Data Structures for Graphs

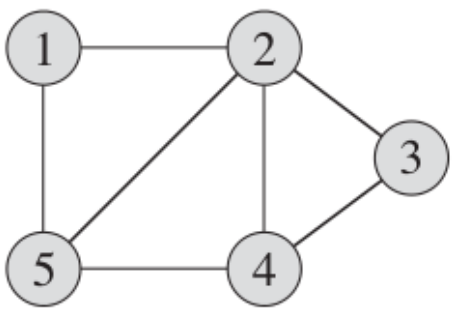
## Adjacency Map

| Operation | Running Time |
|---|---|
| `vertex_count()` | $O(1)$ |
| `edge_count()` | $O(1)$ |
| `vertices()` | $O(n)$ |
| `edges()` | $O(m)$ |
| `get_edge(u,v)` | $O(1)$ exp. |
| `degree(v)` | $O(1)$ |
| `incident_edges(v)` | $O(d_v)$ |
| `insert_vertex(v)` | $O(1)$ |
| `remove_vertex(v)` | $O(d_v)$ |
| `insert_edge(u,v,x)` | $O(1)$ exp. |
| `remove_edge(e)` | $O(1)$ exp. |

# Data Structures for Graphs

## Adjacency Matrix

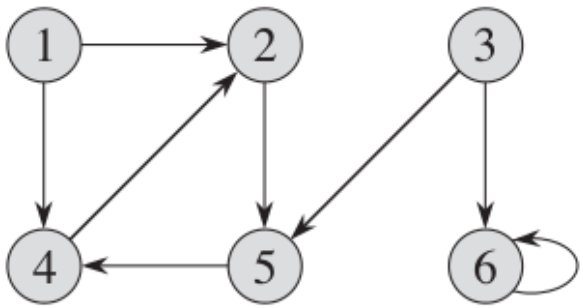
Provides worst-case $O(1)$ access to a specific edge $(u, v)$ by maintaining an $n \times n$ matrix for a graph with $n$ vertices. Each entry is dedicated to storing a reference to the edge $(u, v)$ for a particular pair of vertices $u$ and $v$. If no such edge exists, the entry will be `None`




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

# Data Structures for Graphs

## Adjacency Matrix




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

# Data Structures for Graphs

## Adjacency Matrix

| Operation | Running Time |
|---|---|
| `vertex_count()` | $O(1)$ |
| `edge_count()` | $O(1)$ |
| `vertices()` | $O(n)$ |
| `edges()` | $O(m)$ |
| `get_edge(u,v)` | $O(1)$ |
| `degree(v)` | $O(n)$ |
| `incident_edges(v)` | $O(n)$ |
| `insert_vertex(v)` | $O(n^2)$ |
| `remove_vertex(v)` | $O(n^2)$ |
| `insert_edge(u,v,x)` | $O(1)$ |
| `remove_edge(e)` | $O(1)$ |

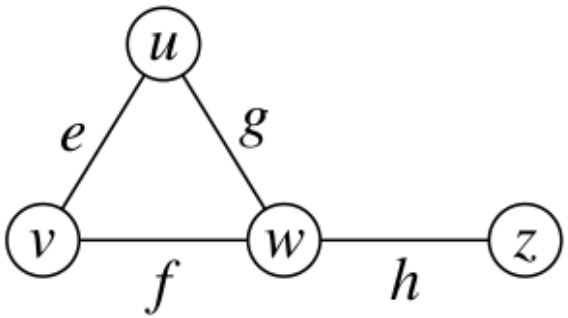
# Data Structures for Graphs

## Adjacency Matrix Structure

- The **adjacency matrix** structure for a graph $G$ uses a matrix $A$, which allows us to locate an edge between a given pair of vertices in *worst-case* constant time
- In this representation, the vertices are the integers in the set $\{0, 1, \ldots n-1\}$ and the edges are pairs of such integers
- Specifically, the cell $A[i, j]$ holds a reference to the edge $(u, v)$ if it exists, where $u$ is the vertex with index $i$ and $v$ is the vertex with index $j$. Is there is no such edge, then $A[i, j] = \mathit{None}$
- Array $A$ is symmetric is graph $G$ is undirected, as $A[i, j] = A[j, i]$ for all pairs $i$ and $j$

# Data Structures for Graphs

## Ajacency Matrix Structure




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

— Advantages

- The most significant advantage of an adjacency matrix is that any edge $(u, v)$ can be accessed in worst-case $O(1)$ time

— Disadvantages

- To find edges incident to vertex $v$ we must examine all $n$ entries in the row associated with $v$. An adjacency list 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
- The $O(n^2)$ space usage of an adjacency matrix is usually worse than the $O(n + m)$ space required for adjacency lists, although the number of edges in a **dense** graph is proportional to $n^2$

# Data Structures for Graphs

| **Operation** | **Edge List** | **Adj. List** | **Adj. Map** | **Adj. Matrix** |
|---|---|---|---|---|
| `vertex_count()` | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| `edge_count()` | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| `vertices()` | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| `edges()` | $O(m)$ | $O(m)$ | $O(m)$ | $O(m)$ |
| `get_edge(u,v)` | $O(m)$ | $O(min(d_u, d_v))$ | $O(1)$ exp. | $O(1)$ |
| `degree(v)` | $O(m)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| `incident_edges(v)` | $O(m)$ | $O(d_v)$ | $O(d_v)$ | $O(n)$ |
| `insert_vertex(v)` | $O(1)$ | $O(1)$ | $O(1)$ | $O(n^2)$ |
| `remove_vertex(v)` | $O(m)$ | $O(d_v)$ | $O(d_v)$ | $O(n^2)$ |
| `insert_edge(u,v,x)` | $O(1)$ | $O(1)$ | $O(1)$ exp. | $O(1)$ |
| `remove_edge(e)` | $O(1)$ | $O(1)$ | $O(1)$ exp. | $O(1)$ |

# Data Structures for Graphs

## Implementation

```
class Vertex:
  def __init__(self, x):
    self._element = x

  def element(x):
    return self._element

  def __hash__(self):
    return hash(id(self))
```

# Data Structures for Graphs

## Implementation

```
class Edge:
  def __init__(self, u, v, x):
    self._origin = u
    self._destination = v
    self._element = x

  def endpoints(self):
    return (self._origin, self._destination)

  def opposite(self, v):
    return self._destination if v is self._origin else
        self._origin

  def element(self):
    return self._element

  def __hash__(self):
    return hash(self._origin, self._destination)
```

# Data Structures for Graphs

## Implementation

```
class Graph:
  def __init__(self, directed=False):
    self._outgoing = {}
    self._incoming = {} if directed else self._outgoing

  def is_directed(self):
    return self._incoming is not self._outgoing

  def vertex_count(self):
    return len(self._outgoing)

  def vertices(self):
    return self._outgoing.keys()
```

To be continued ...

# Data Structures for Graphs

## Implementation

```
def edge_count(self):
  total = sum(len(self._outgoing[v]) for v in self.
      _outgoing)
  return total if self.id_directed() else total // 2

def edges(self):
  result = set()
  for secondary_map in self._outgoing.values():
    result.update(secondary_map.values())
  return result

def get_edge(self, u, v):
  return self._outgoing[v].get(v)
```

To be continued ...

# Data Structures for Graphs

## Implementation

```
  def degree(self,v, outgoing=True):
    adj = self._outgoing if outgoing else self._incoming
    return len(adj[v])

  def incident_edges(self, v, outgoing=True):
    adj = self._outgoing if outgoing else self._incoming
    for edge in adj[v].values():
      yield edge
```

To be continued ...

# Data Structures for Graphs

## Implementation

```
def insert_vertex(self, x=None):
  v = self.Vertex(x)
  self._outgoing[v] = {}
  if self.is_directed():
    self._incoming[v] = {}
  return

def insert_edge(self, u, v, x=None):
  e = self.Edge(u, v, x)
  self._outgoing[u][v] = e
  self._incoming[v][u] = e
```

Not to be continued :-)

# Graph Traversals

A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time

# Graph Traversals

Interesting problems that deal with reachability in an **undirected graph** $G$ include the following:

- Computing a path from vertex $u$ to vertex $v$, or reporting that no such path exists
- Given a start vertex $s$ of $G$, computing for every vertex $v$ of $G$ a path with the minimum number of edges between $s$ and $v$, or reporting that no such path exists
- Testing whether $G$ is connected
- Computing a spanning tree of $G$, is $G$ is connected
- Computing a cycle in $G$, or reporting that $G$ has no cycles

# Graph Traversals

Interesting problems that deal with reachability in an **directed graph** $G$ include the following:

- Computing a directed path from vertex $u$ to vertex $v$, or reporting that no such path exists
- Finding all the vertices of $G$ that are reachable from a given vertex $v$
- Determine whether $G$ is acyclic
- Determine whether $G$ is strongly connected

# Depth-First Search

Depth-first search (DFS) in a graph $G$ is useful for testing a number of properties of graphs, including whether there is a path from one vertex to another and whether or not a graph is connected

- We begin at a specific starting vertex $s$ is $G$
- Suppose that a given moment our current vertex is $u$
- We then traverse $G$ by considering an arbitrary edge $(u, v)$ incident to the current vertex $u$
- If the edge $(u, v)$ leads to a vertex $v$ that is already visited, that edge is ignored
- If, on the other hand, $(u, v)$ leads to an unvisited vertex $v$, the next current vertex will be $v$
- The vertex $v$ is marked as "visited" and is considered as the current vertex

# Depth-First Search

- Eventually, there is the possibility to get to a "dead end", that is a current vertex $v$ such that all the edges incident to $v$ lead to vertices already visited
- To get out of this impasse, we backtrack along the edge that brought to $v$, going back to a previously visited vertex $u$
- Now $u$ is the current vertex and the computation is repeated for any edges incident to $u$ that has not yet been considered
- If all incident edges to $u$ have been visited, it is necessary to backtrack again to the vertex that brought to $u$, and repeat the procedure at that vertex
- The process terminates when the backtrack leads back to the start vertex $s$, and there are no more unexplored edges incident to $s$

# 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

  for each outgoing edge e = (u,v) of u do
    if vertex v has not been visited then
      Mark vertex v as visited (via edge e)
      Recursively call DFS(G, v)
```

# Depth-First Search

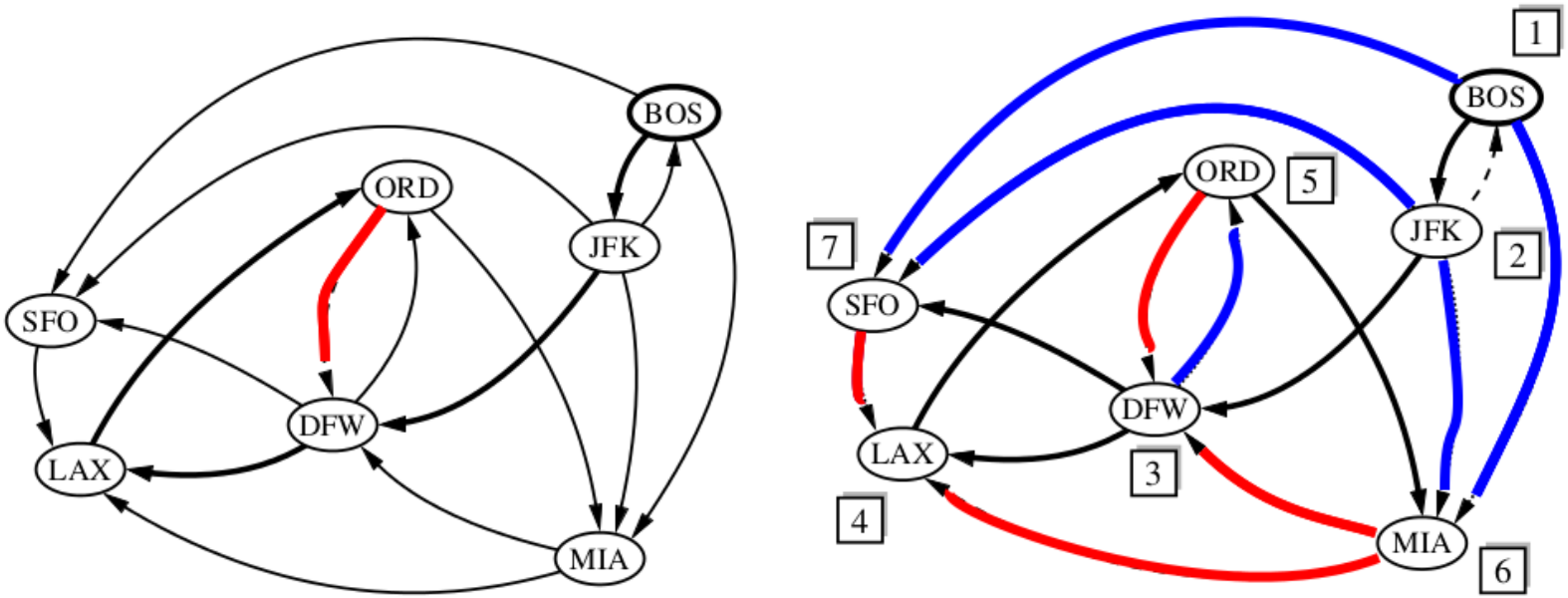
- An execution of DFS can be used to analyze the structure of a graph
- The DFS process naturally identifies the **depth-first search tree** rooted at a starting vertex $s$
- Whenever an edge $e = (u, v)$ is used to discover a new vertex $v$ during the DFS algorithm, that edge is known as **discovery edge** or **tree edge**
- All other edges considered during the execution of DFS are known as **nontree edges**, and takes us to a previously explored vertex
- In the case of an undirected graph, all nontree edges that are explored connect the current vertex to one that is an ancestor of it in the DFS tree (**back tree**)

# Depth-First Search

When performing a DFS on a directed graph, there are three possible kinds of nontree edges:

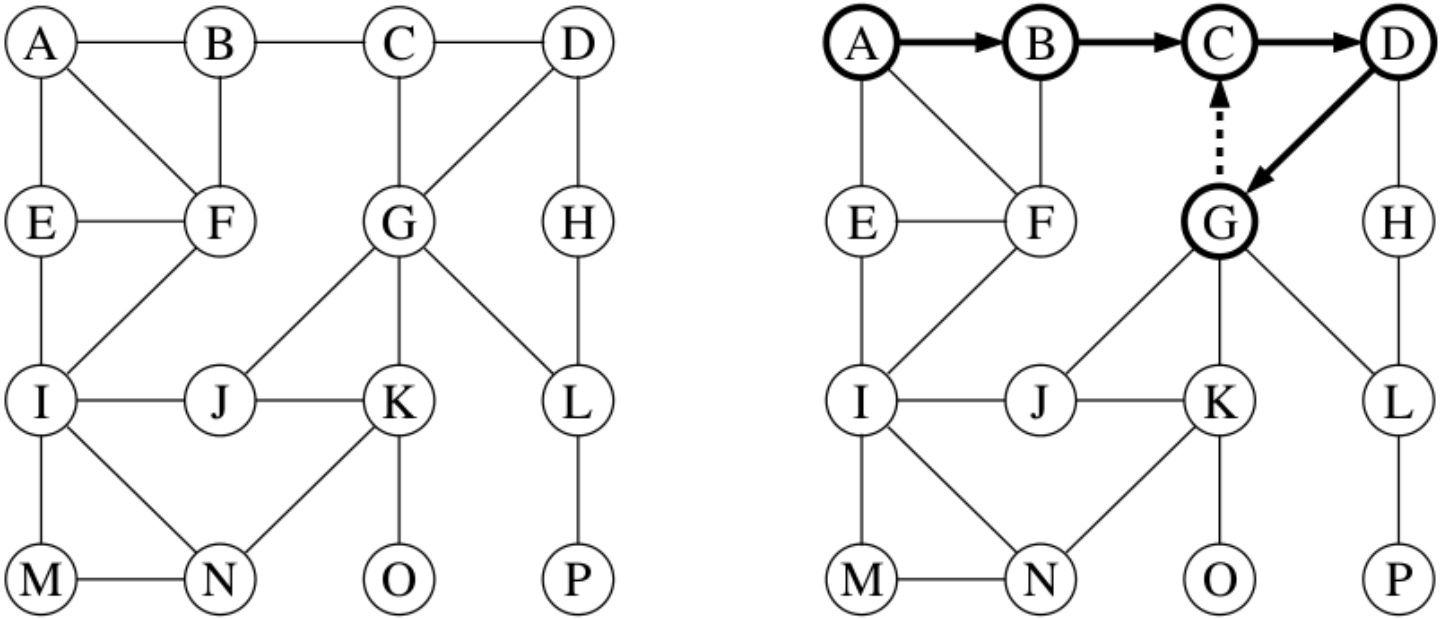
- **back edges**, which connect a vertex to its ancestor in the DFS tree
- **forward edges**, which connect a vertex to its descendant in the DFS tree
- **cross edges**, which connect a vertex to a vertex that is neither its ancestor nor its descendant

# Depth-First Search



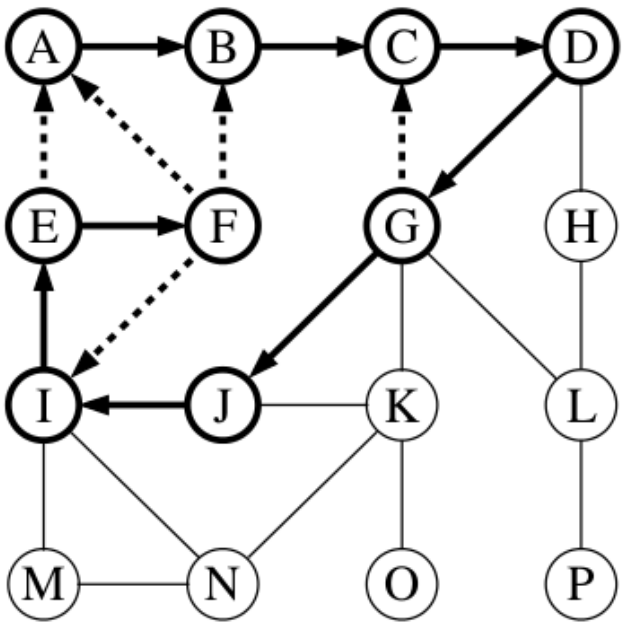

- **tree edges**, thick lines
- **back edges**, red lines
- **forward edges** and **cross edges**, blue lines

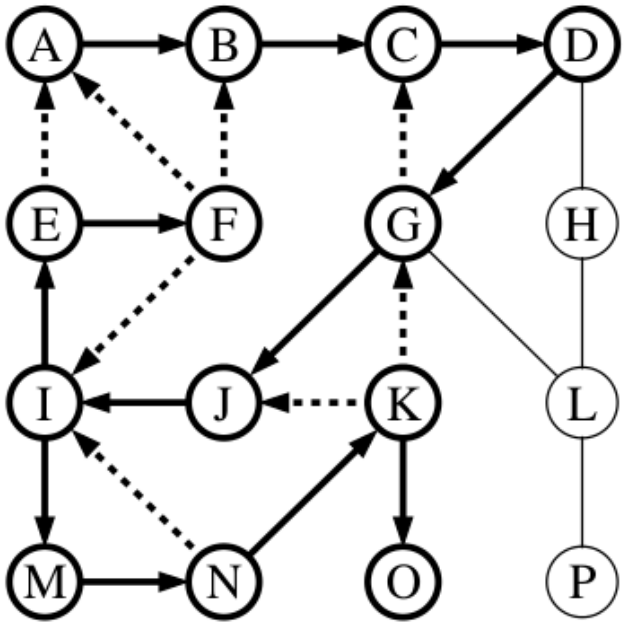
# Depth-First Search




- Visited vertices and explored edges are highlighted
- Nontree edges are drawn as dashed lines
- Left: input graph
- Right: DFS traversal starting at vertex A and proceding in alphabetical order (the edge GC is a nontree edge)

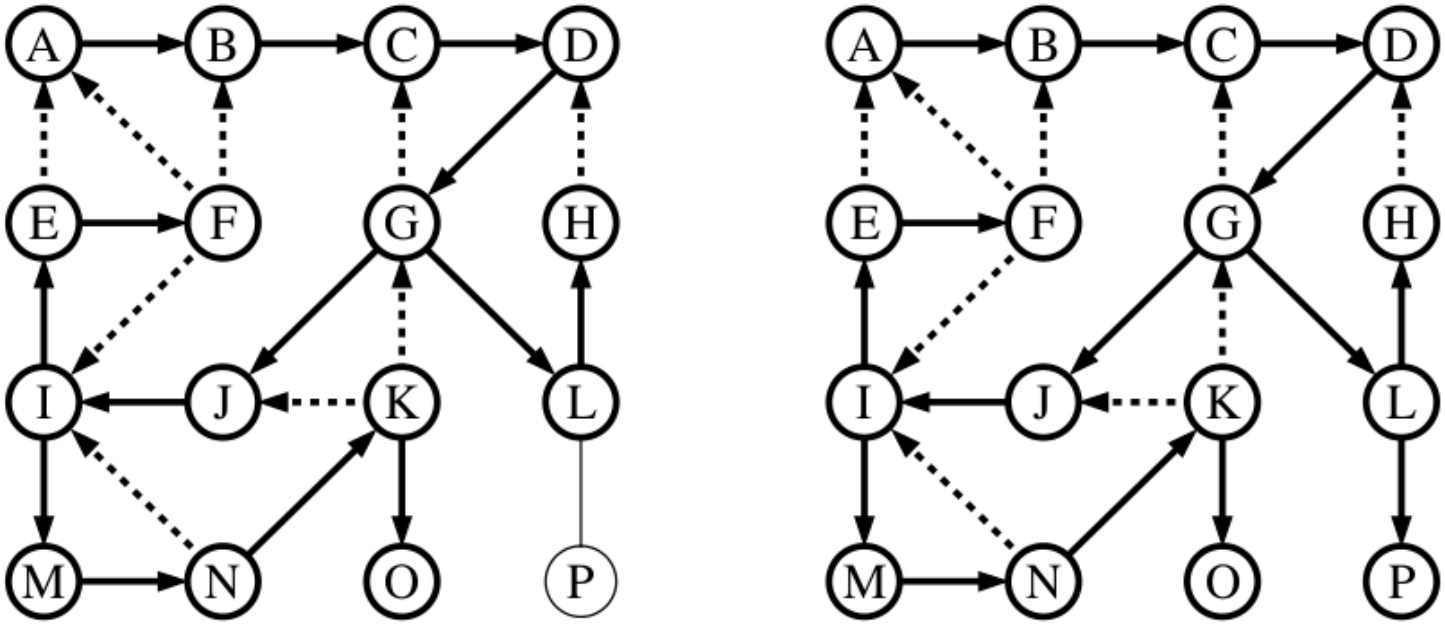
# Depth-First Search




- Left: Vertex F is a dead end
- Right: DFS is backtracked up to vertex I. Vertex O is another dead end

# Depth-First Search




- Visited vertices and explored edges are highlighted
- Nontree edges are drawn as dashed lines
- Left: After backtracking to G, DFS continues with edge $(G, L)$ and hits another dead end at H
- Right: final result

**Proposition**
*Let $G$ be an undirected graph on which a DFS traversal starting at a vertex $s$ has been performed. Then the traversal visits all vertices in the connected component of $s$, and the discovery edges form a spanning tree of the connected component of $s$.*

*Justification*
Suppose there is at least one vertex $w$ not visited in the connected component of $s$, and let $v$ be the first unvisited vertex on some path from $s$ to $w$.
Since $v$ is the first unvisited vertex on this path, it has a neighbor $u$ that was visited. But when $u$ was visited, the edge $(u, v)$ must have been considered, hence it cannot be correct that $v$ is unvisited. Therefore, there are no unvisited vertices in the connected component of $s$.

**Proposition**

*Let $\overrightarrow{G}$ be a directed graph. A DFS on $\overrightarrow{G}$ starting at a vertex $s$ visits all the vertices of $\overrightarrow{G}$ that are reachable from $s$. Also, the DFS tree contains directed paths from $s$ to every vertex reachable from $s$.*

*Justification (first part)*

Let $V_s$ be the subset of vertices of $\overrightarrow{G}$ visited by DFS starting at vertex $s$. We want to show that $V_s$ contains $s$, and every vertex reachable from $s$ belongs to $V_s$.

Suppose now that there is a vertex $w$ reachable from $s$ that is not in $V_s$. Consider a directed path from $s$ to $w$, and let $(u, v)$ be the first edge on such a path taking us out of $V_s$, that is, $u$ is in $V_s$ but $v$ is not in $V_s$.

*To be continued*

**Proposition**

*Let $\overrightarrow{G}$ be a directed graph. A DFS on $\overrightarrow{G}$ starting at a vertex $s$ visits all the vertices of $\overrightarrow{G}$ that are reachable from $s$. Also, the DFS tree contains directed paths from $s$ to every vertex reachable from $s$.*

*Continued*

When DFS reaches $u$, it explores all the outgoing edges of $u$, and thus must reach also vertex $v$ via edge $(u, v)$. Hence $v$ should be in $V_s$, and we have obtained a contradiction. Therefore, $V_s$ must contain every vertex reachable from $s$.

**Proposition**

*Let $\overrightarrow{G}$ be a directed graph. A DFS on $\overrightarrow{G}$ starting at a vertex $s$ visits all the vertices of $\overrightarrow{G}$ that are reachable from $s$. Also, the DFS tree contains directed paths from $s$ to every vertex reachable from $s$.*

*Justification (second part)*

The second part is proved by induction on the steps of the algorithm. Each time a discovery edge $(u, v)$ is identified, there exists a directed path from $s$ to $v$ in the DFS tree. Since $u$ must have previously been discovered, there exists a path from $s$ to $u$, so by appending the edge $(u, v)$ to that path, we have a directed path from $s$ to $v$.

# Depth-First Search

## Running Time of DFS

- DFS is an efficient method for traversing a graph
- DFS is called at most once on each vertex (since it gets marked as visited), and therefore every edge is examined at most twice for an undirected graph (once for each of its vertices), and at most once for a directed graph (from its origin vertex)
- If we let $n_s \leq n$ be the number of vertices reachable from a vertex $s$, and $m_s \leq m$ be the number of incident edges to those vertices, a DFS starting at $s$ runs in $O(n_s + m_s)$ time, provided that ...

...

- The graph is represented by a data structure such that creating and iterating through the `incident_edges(v)` takes $O(deg(v))$ time, and the `e.opposite(v)` method takes $O(1)$ time. The adjacency list is one such structure (not the adjacency matrix)
- There is a way to mark a vertex or edge as explored, and to test if a vertex or edge has been explored in $O(1)$ time

Given these assumptions, a number of problems can be solved

**Proposition**

*Let $G$ be an undirected graph with $n$ vertices and $m$ edges. A DFS traversal of $G$ can be performed in $O(n+m)$ time, and can be used to solve the following problems in $O(n+m)$ time:*

- *Computing a path between two given vertices of $G$, if one exists*
- *Testing whether $G$ is connected*
- *Computing a spanning tree of $G$, if $G$ is connected*
- *Computing the connected components of $G$*
- *Computing a cycle in $G$, or reporting that $G$ has no cycles*

**Proposition**

*Let $\overrightarrow{G}$ be a directed graph. A DFS traversal on $\overrightarrow{G}$ starting can be performed in $O(n+m)$ time, and can be used to solve the following problems in $O(n+m)$ time:*

- *Computing a directed path between two given vertices of $\overrightarrow{G}$, if one exists*
- *Computing the set of vertices of $\overrightarrow{G}$ that are reachable from a given vertex $s$*
- *Testing whether $\overrightarrow{G}$ is strongly connected*
- *Computing a directed cycle in $\overrightarrow{G}$, or reporting that $\overrightarrow{G}$ is acyclic*
- *Computing the* **transitive closure** *of $\overrightarrow{G}$*

# Depth-First Search

## Implementation

```
def DFS(g, u, discovered):
""" g          : graph
    u          : starting node
    discovered: dictionary
"""
  for e in g.incident_edges(u):
    v = e.opposite(u)
    if v not in discovered:
      discovered[v] = e
      DFS(g, v, discovered)
```

# Depth-First Search

## Reconstructing a Path from $u$ to $v$

To reconstruct the path that connects $u$ to $v$:

1. Go at the end of the path (namely, consider vertex $v$)
2. Name vertex $v$ *walk*
3. Examine the dictionary to determine which edge was used to reach *walk*
4. Determine the other endpoint of the edge, say vertex $w$
5. Name vertex $w$ *parent*
6. Add vertex *parent* to path
7. Name *parent walk*
8. Iterate over steps 3-7 until `parent = u`

# Depth-First Search

## Reconstructing a Path from $u$ to $v$

```
def construct_path(u, v, discovered):
  path = []
  if v in discovered:
    path.append(v)
    walk = v
    while walk is not u:
      e = discovered[walk]
      parent = e.opposite(walk)
      path.append(parent)
      walk = parent
    path.reverse()
  return path
```

# Depth-First Search

## Testing for Connectivity

- DFS can be used to determine whether a graph is connected
- For an undirected graph, after a DFS we must verify that `len(discovered) == n`
- If the graph is connected, then all vertices will have been discovered
- If not, there must be at least one vertex $v$ that is not reachable from $v$

- For directed graphs, DFS allows to test whether it is **strongly connected**, that is whether for every pair of vertices $u$ and $v$, both $u$ reaches $v$ and $v$ reaches $u$
- In principle the test should be accomplished for every vertex, and therefore the computational cost would be $O(n(m+n))$
- Fortunately, it suffices to perform only two DFS, the second with the orientation of all edges reversed

# Depth-First Search

## Computing all Connected Components

When a graph is not connected, it is often useful to identify all the **connected components** of an undirected graph, or the **strongly connected components** of a directed graph

```
def DFS_complete(g):
  forest = {}
  for u in g.vertices():
    if u not in forest:
      forest[u] = None
      DFS(g, u, forest)
  return forest
```

# Depth-First Search

## Computing all Connected Components

- Although the `DFS_complete` function makes multiple calls to `DFS`, the total time spent by a call is $O(n + m)$
- Indeed, a single call to `DFS` starting at vertex $s$ runs in time $O(n_s + m_s)$, where $n_s$ is the number of vertices reachable from $s$, and $m_s$ is the number of edges incident to those vertices
- Each call to `DFS` explores a different component, the sum of $n_s + m_s$ is $n + m$
- Note that $O(n + m)$ applies to directed graphs as well, even though the sets of reachable vertices are not necessarily disjoint

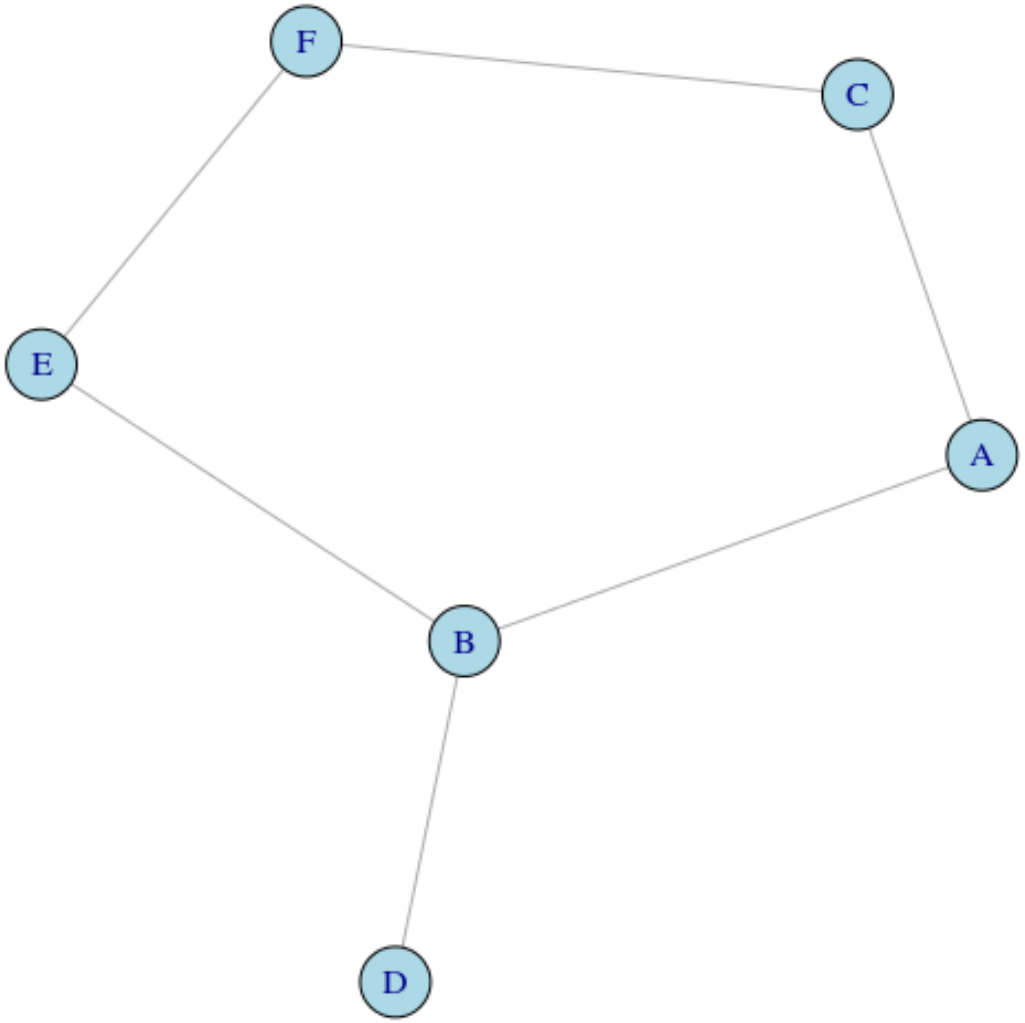
# Depth-First Search

## Detecting Cycles

- For both directed and undirected graphs, a cycle exists if and only if a **back edge** exists relative to the DFS traversal of that graph
- If a back edge exists, a cycle exists by taking the back edge from the descendant to its ancestor and then following the tree edges back to the descendant
- Conversely, if a cycle exists in the graph, there must be a back edge relative to a DFS
- In an undirected graph, all edges are either tree or back edges, so finding a cycle is relatively easy
- In a directed graph, when exploration leads to a previously visited vertex, it is necessary to recognize whether that vertex is an ancestor of the current vertex

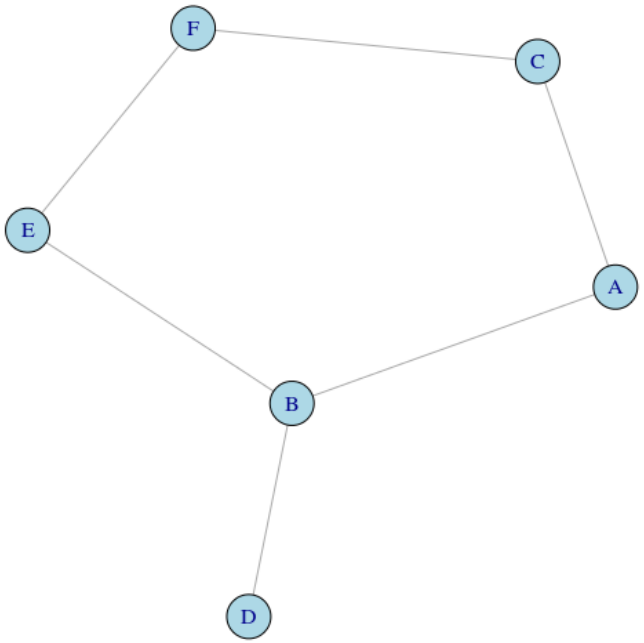
# Depth-First Search

## Implementations



# Depth-First Search

## Implementations




```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

# Depth-First Search

## Implementations

```
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
      vertex = stack.pop()
      if vertex not in visited:
        visited.add(vertex)
        stack.extend(graph[vertex] - visited)
    return visited
```

# Depth-First Search

## Implementations

```
Iteration 0
Found node A connected to {'C', 'B'}
Already visited nodes {'A'}
Added nodes {'C', 'B'}
Stack contains ['C', 'B']

Iteration 1
Found node B connected to {'A', 'E', 'D'}
Already visited nodes {'A', 'B'}
Added nodes {'E', 'D'}
Stack contains ['C', 'E', 'D']

Iteration 2
Found node D connected to {'B'}
Already visited nodes {'A', 'B', 'D'}
Added nodes set()
Stack contains ['C', 'E']

Iteration 3
Found node E connected to {'B', 'F'}
Already visited nodes {'A', 'B', 'E', 'D'}
Added nodes {'F'}
Stack contains ['C', 'F']
```

```
Iteration 4
Found node F connected to {'C', 'E'}
Already visited nodes {'A', 'B', 'E', 'D', 'F'}
Added nodes {'C'}
Stack contains ['C', 'C']

Iteration 5
Found node C connected to {'A', 'F'}
Already visited nodes {'A', 'C', 'B', 'E', 'D', 'F'}
Added nodes set()
Stack contains ['C']

Iteration 6
Found a vertex already visited: C
```

# Depth-First Search

## Implementation: path

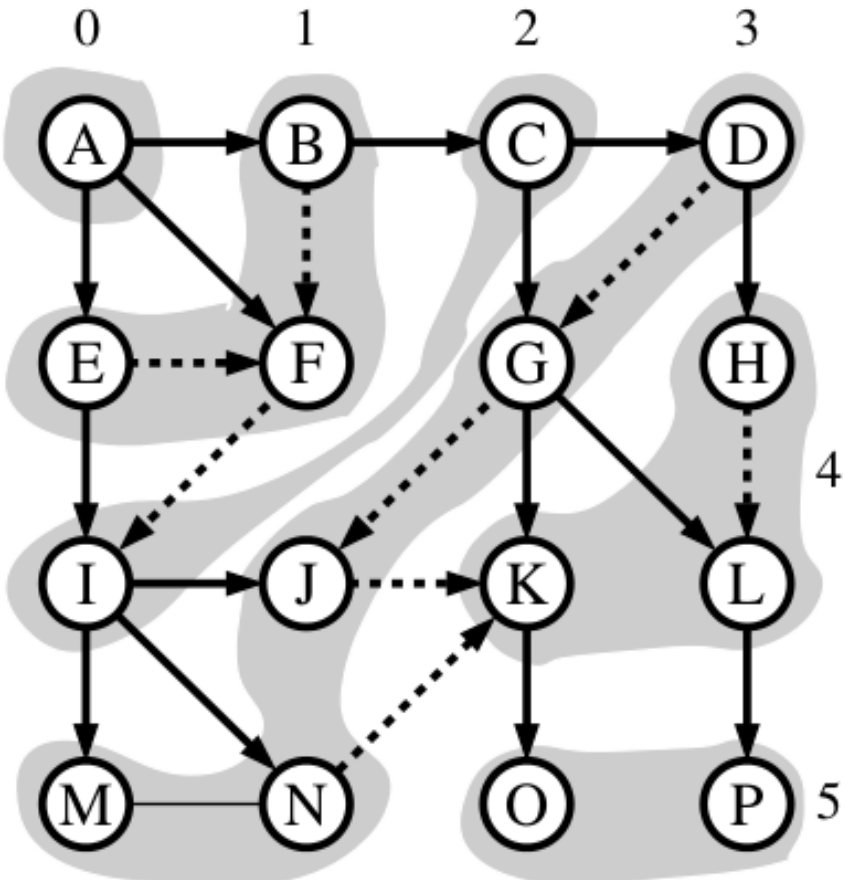
```
def dfs_paths(graph, start, goal):
  stack = [(start, [start])]
  while stack:
    (vertex, path) = stack.pop()
      for next in graph[vertex] - set(path):
        if next == goal:
          yield path + [next]
        else:
          stack.append((next, path + [next]))
```

# Depth-First Search

## Implementation: path

```
giacomo@giacomo-N76VB2:~/Dropbox/Teach/AACM/Lectures-20152016/examples$ python dfs_path
Stack contains:  [('C', ['A', 'C'])]
['A', 'C']
Stack contains:  [('C', ['A', 'C']), ('B', ['A', 'B'])]
['A', 'B']
Stack contains:  [('C', ['A', 'C']), ('E', ['A', 'B', 'E'])]
['A', 'B', 'E']
Stack contains:  [('C', ['A', 'C']), ('E', ['A', 'B', 'E']), ('D', ['A', 'B', 'D'])]
['A', 'B', 'D']
['A', 'B', 'E', 'F']
['A', 'C', 'F']
```

# Breadth-First Search



# Breadth-First Search

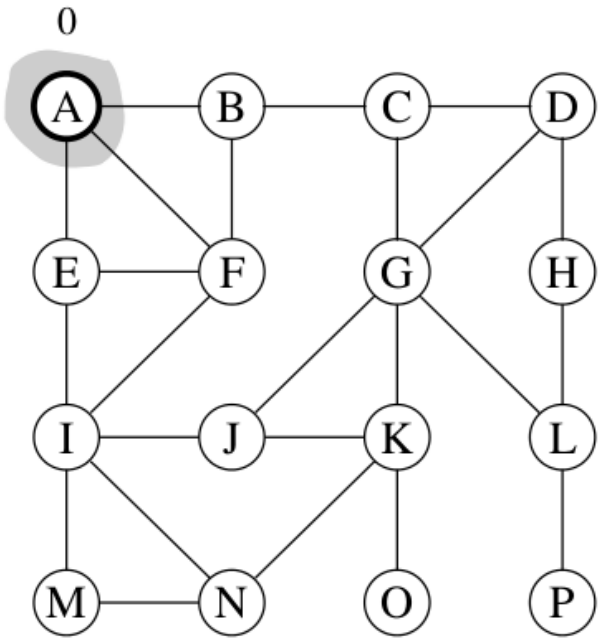
- DFS defines a traversal that could be physically traced by a single person exploring the graph
- **Breadth-First Search** (BFS) can be described as sending many explorers who collectively explore the graph (nothing to do with parallel implementations)
- A BFS proceeds in rounds and divides the vertices in levels
- In the first round, are marked as "visited" all vertices adjacent to the start vertex $s$ (these vertices are one step away from the beginning and are placed into level 1)
- In the second round, explorations concerns vertices two steps (edges) away from the starting vertex. The new vertices are adjacent to level 1 vertices and are therefore placed in level 2
- The process continues in similar fashion terminating when no new vertices are found in a level

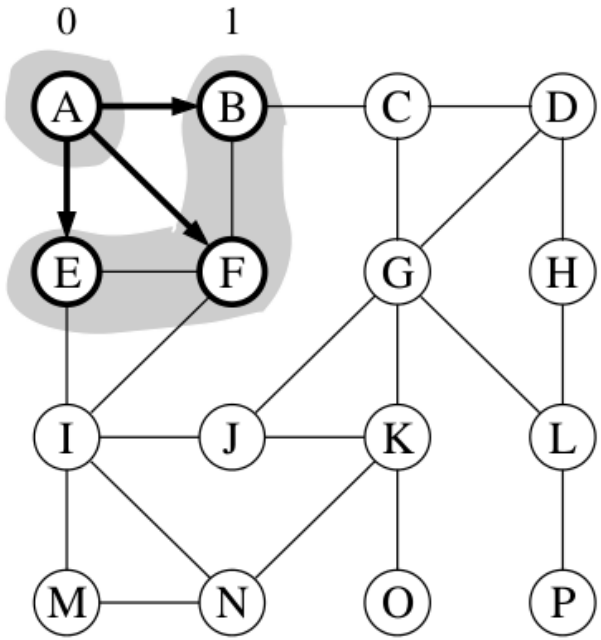
# Breadth-First Search

## Implementation

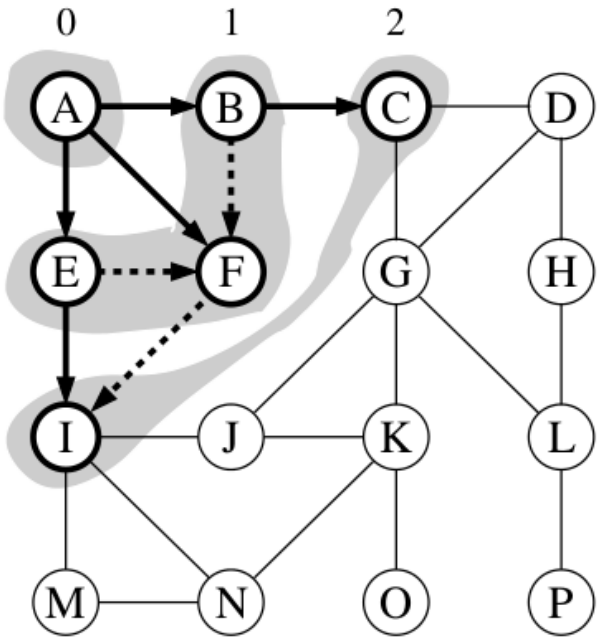
```
def BFS(g, s, discovered):
""" g          : graph
    s          : starting node
    discovered: dictionary
"""
  level = [s]
  while len(level) > 0:
    next_level = []
    for u in level:
      for e in g.incident_edges(u):
        v = e.opposite(u)
        if v not in discovered:
          discovered[v] = e
          next_level.append[v]
    level = next_level
```

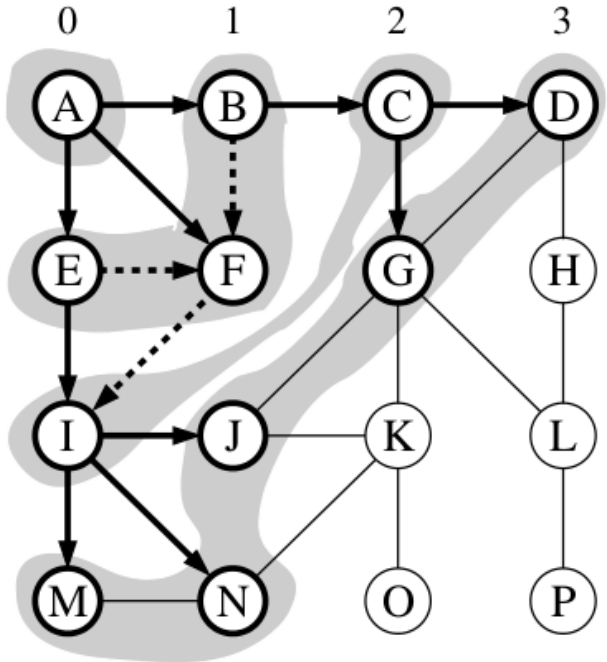
# Breadth-First Search



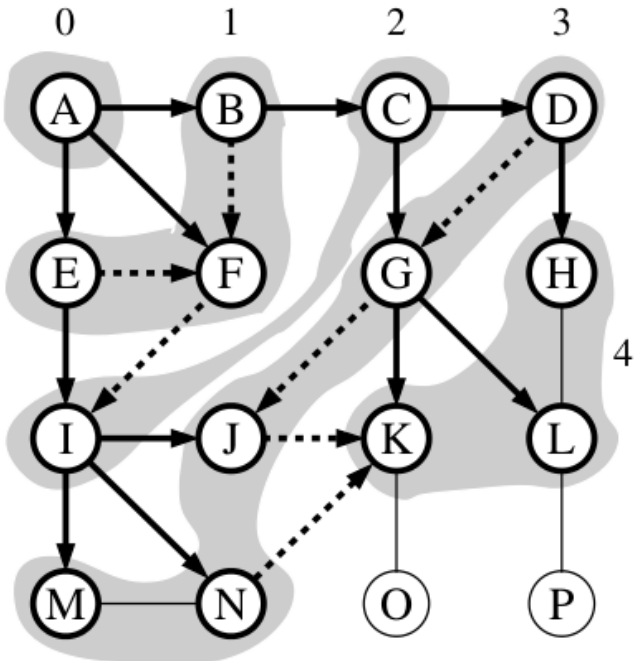



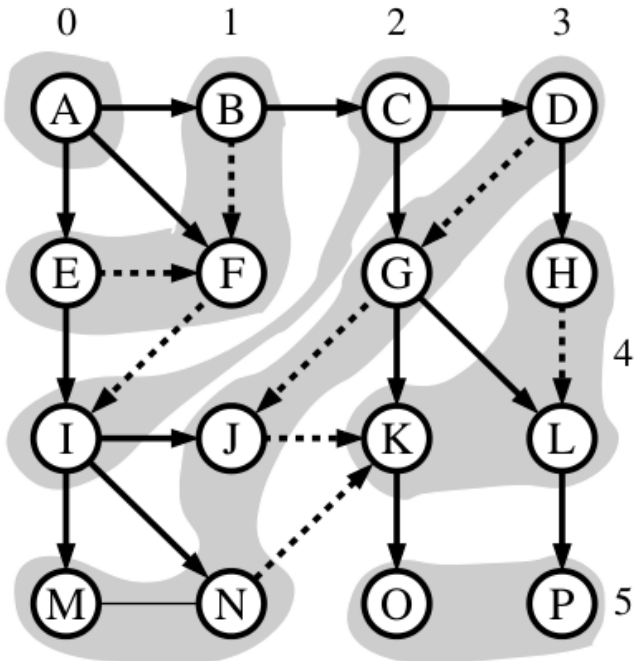
# Breadth-First Search



# Breadth-First Search



| **Type of edge** | **Connects ...** |
|---|---|
| back edge | a vertex to one of its ancestors |
| forward edge | a vertex to one if descendants |
| cross edge | a vertex to another vertex (neither ancestor nor descendant) |

For BFS:

- on undirected graphs, all nontree edges are cross edges
- on directed graphs, all nontree edges are wither back edges or cross edges

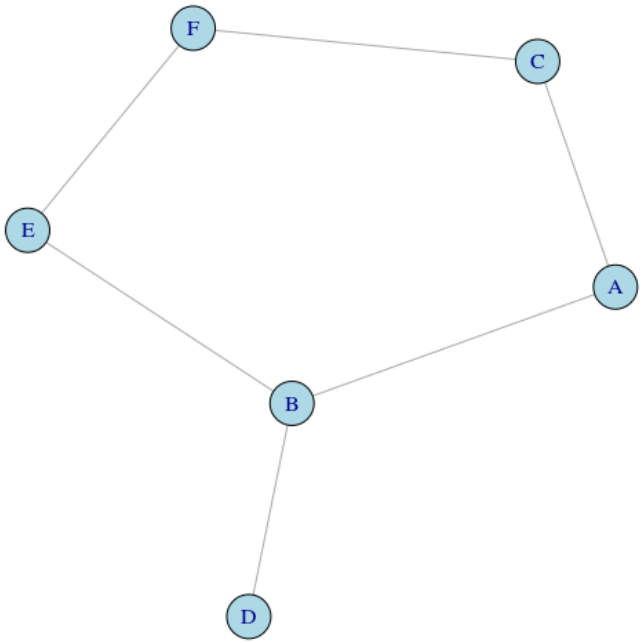
**Proposition**

*Let $G$ be an undirected or directed graph on which a BFS traversal starting at vertex $s$ has been performed. Then*

- *The traversal visits all vertices of $G$ that are reachable from $s$*
- *For each vertex $v$ at level $i$, the path of the BFS tree $T$ between $s$ and $v$ has $i$ edges, and any other path of $G$ from $s$ to $v$ has at least $i$ edges*
- *If $(u, v)$ is an edge that is not in the BFS tree, then the level number of $v$ can be at most 1 greater than the level number of $u$*

# Breadth-First Search

## Another Implementation




```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

# Breadth-First Search

## Another Implementation

```
def bfs(graph, start):
  visited, queue = set(), [start]
  while queue:
    vertex = queue.pop(0)
    if vertex not in visited:
      visited.add(vertex)
      queue.extend(graph[vertex] - visited)
  return visited
```

# Breadth-First Search

## Another Implementation

```
Iteration 0
Found node A connected to {'C', 'B'}
Already visited nodes {'A'}
Added nodes {'C', 'B'}
Stack contains ['C', 'B']

Iteration 1
Found node C connected to {'A', 'F'}
Already visited nodes {'A', 'C'}
Added nodes {'F'}
Stack contains ['B', 'F']

Iteration 2
Found node B connected to {'A', 'E', 'D'}
Already visited nodes {'A', 'C', 'B'}
Added nodes {'E', 'D'}
Stack contains ['F', 'E', 'D']

Iteration 3
Found node F connected to {'C', 'E'}
Already visited nodes {'A', 'C', 'B', 'F'}
Added nodes {'E'}
Stack contains ['E', 'D', 'E']

Iteration 4
Found node E connected to {'B', 'F'}
Already visited nodes {'A', 'C', 'B', 'E', 'F'}
Added nodes set()
Stack contains ['D', 'E']

Iteration 5
Found node D connected to {'B'}
Already visited nodes {'A', 'C', 'B', 'E', 'D', 'F'}
Added nodes set()
Stack contains ['E']

Iteration 6
Found a vertex already visited: E
```

# Transitive Closure

## The Mathematical Definition

- The **transitive closure** of a binary relation $R$ on a set $X$ is the transitive relation $R^\dagger$ on set $X$ such that $R^\dagger$ contains $R$ and $R^\dagger$ is minimal
- If the relation itself is **transitive**, then the transitive closure is that same binary relation
- Otherwise, the transitive closure is a different relation

If $X$ is a set of airports and $xRy$ means "there is a direct flight from airport $x$ to airport $y$", then the transitive closure of $R$ on $X$ is the relation $R^\dagger$: "it is possible to fly from $x$ to $y$ in one or more flights."

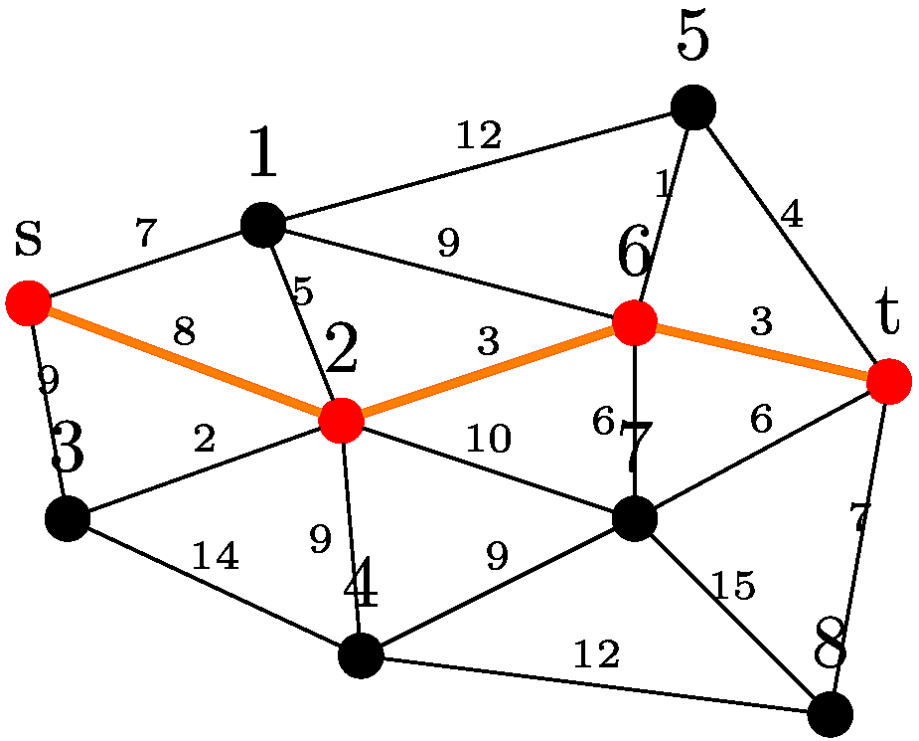
# Transitive Closure

- Graph traversals can be used to answer basic questions of reachability in a directed graph
- In particular, DFS or BFS can be used to know whether there is a path from vertex $u$ to vertex $v$
- If the graph is represented with an adjacency list, the question of reachability can be computed in $O(n + m)$ time
- In certain applications, reachability may be computed more efficiently, for example precomputing a more convenient representation of a graph
- One possible solution is represented by the **transitive closure** of a directed graph $\overrightarrow{G}$

# Transitive Closure

- The **transitive closure** of a directed graph $\overrightarrow{G}$ is a directed graph $\overrightarrow{G^*}$ such that the vertices of $\overrightarrow{G^*}$ are the same as the vertices of $\overrightarrow{G}$, and $\overrightarrow{G^*}$ has an edge $(u, v)$ whenever $\overrightarrow{G}$ has a directed path from $u$ to $v$
- If the graph is represented as an adjacency list, its transitive closure can be computed in $On(n + m)$ time by making use of $n$ graph traversals, one from each starting vertex

# Shortest-path



# The Floyd-Warshall Algorithm

- The **Floyd-Warshall** (FW) algorithm is a solution to the problem of all-pairs shortest-paths on a directed graph $G = (V, E)$
- It runs in $O(V^3)$ time
- The FW algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $p = (v_1, v_2, \ldots, v_l)$ is any vertex of $p$ other than $v_1$ or $v_l$, that is any vertex in $\{v_2, v_3, \ldots, v_{l-1}\}$

# The Floyd-Warshall Algorithm

- Suppose that the vertices of $G$ are $V = \{1, 2, \ldots, n\}$
- Let us consider a subset $\{1, 2, \ldots, k\}$ of vertices for some $k$
- Let $p$ be a minimum-weight path from among them
- The FW algorithm exploits a relationship between path $p$ and shortest paths from $i$ to $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$
- The relationship depends on whether or not $k$ is an intermediate vertex of path $p$

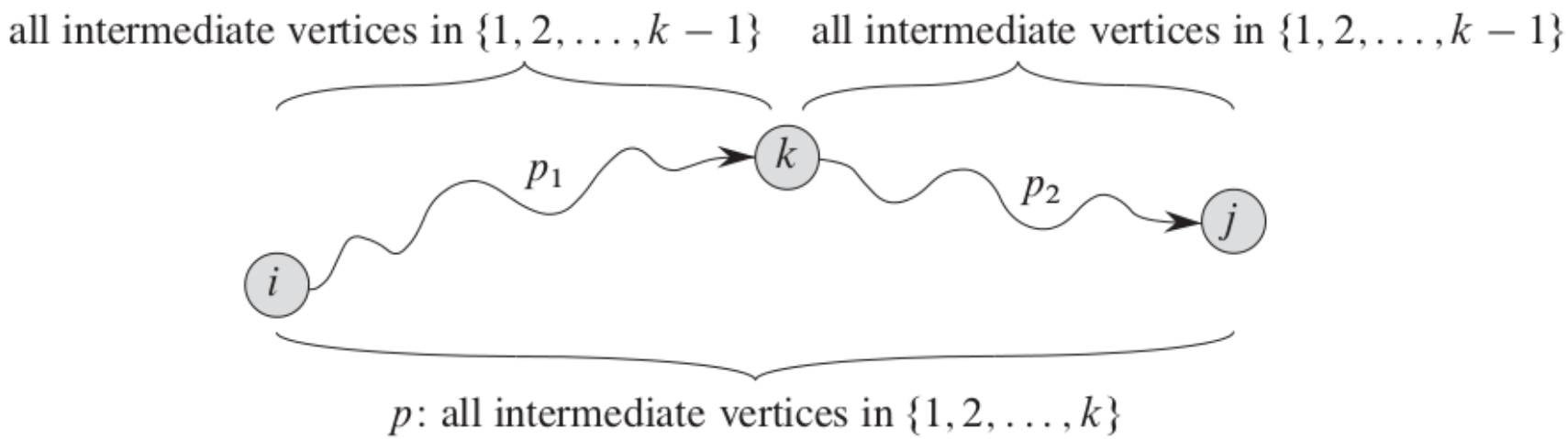
# The Floyd-Warshall Algorithm

- If $k$ is not an intermediate vertex of path $p$, then all intermediate vertices of path $p$ are in the set $\{1, 2, \ldots, k-1\}$
- Thus, a shortest path from vertex $i$ to vertex $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from $i$ to $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k\}$

# The Floyd-Warshall Algorithm

- If $k$ is an intermediate vertex of path $p$, then the path $p$ can be decomposed into two paths $p_1$ (from $i$ to $k$) and $p_2$ (from $k$ to $j$)
- $p_1$ is a shortest path from $i$ to $k$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$
- Similarly, $p_2$ is a shortest path from $k$ to $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$

# The Floyd-Warshall Algorithm



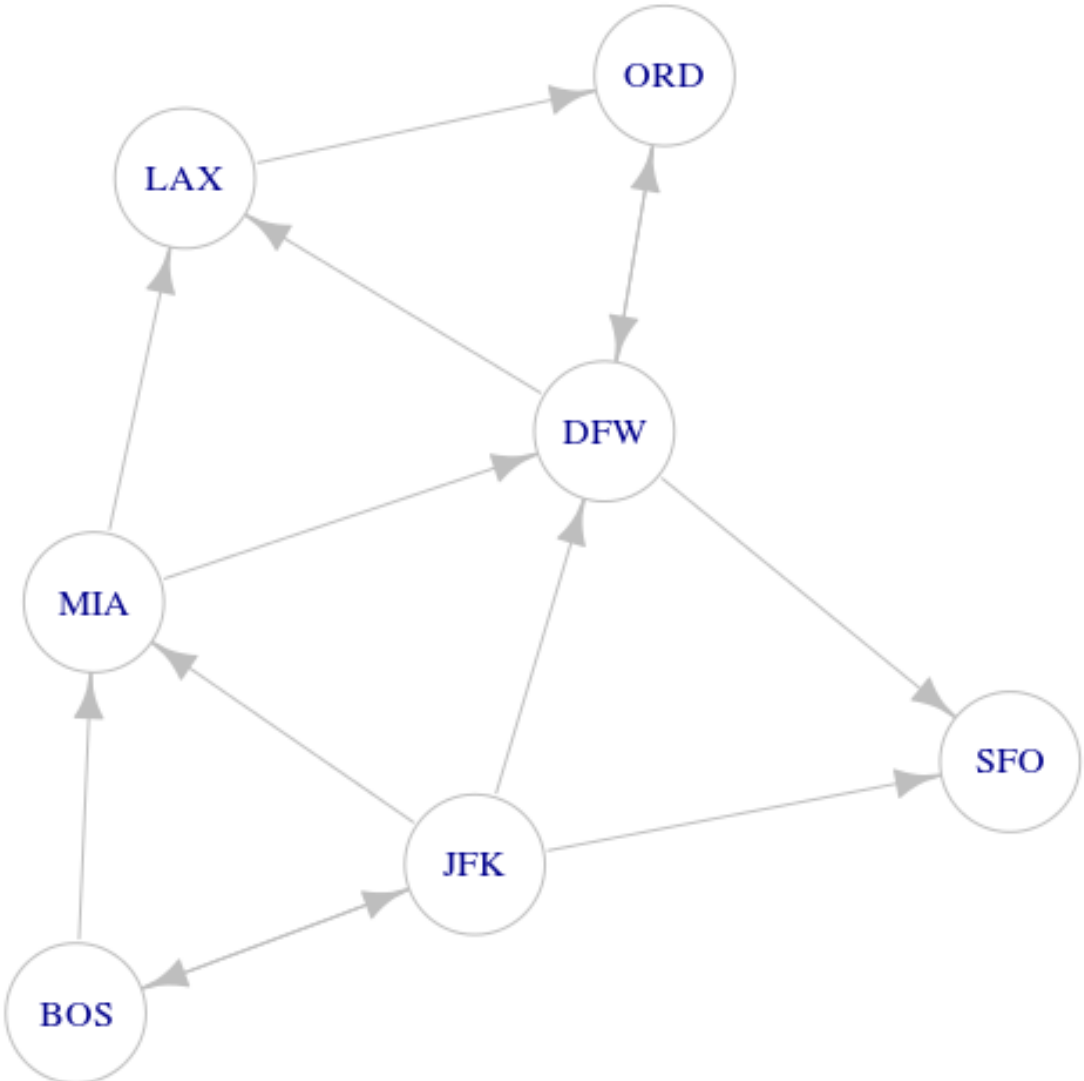
# The Floyd-Warshall Algorithm

## Implementation

```
def floyd_warshall(g):
  closure = deepcopy(g)
  verts = list(closure.vertices())
  n = len(verts)
  for k in range(n):
    for i in range(n):
      if i != k and closure.get_edge(verts[i], verts[k])
           is not None:
        for j in range(n):
          if i != j != k and closure.get_edge(verts[k],
              verts[j]) is not None:
            if closure.get_edge(verts[i], verts[j]) is
                None:
              closure.insert_edge(verts[i], verts[j])
  return closure
```

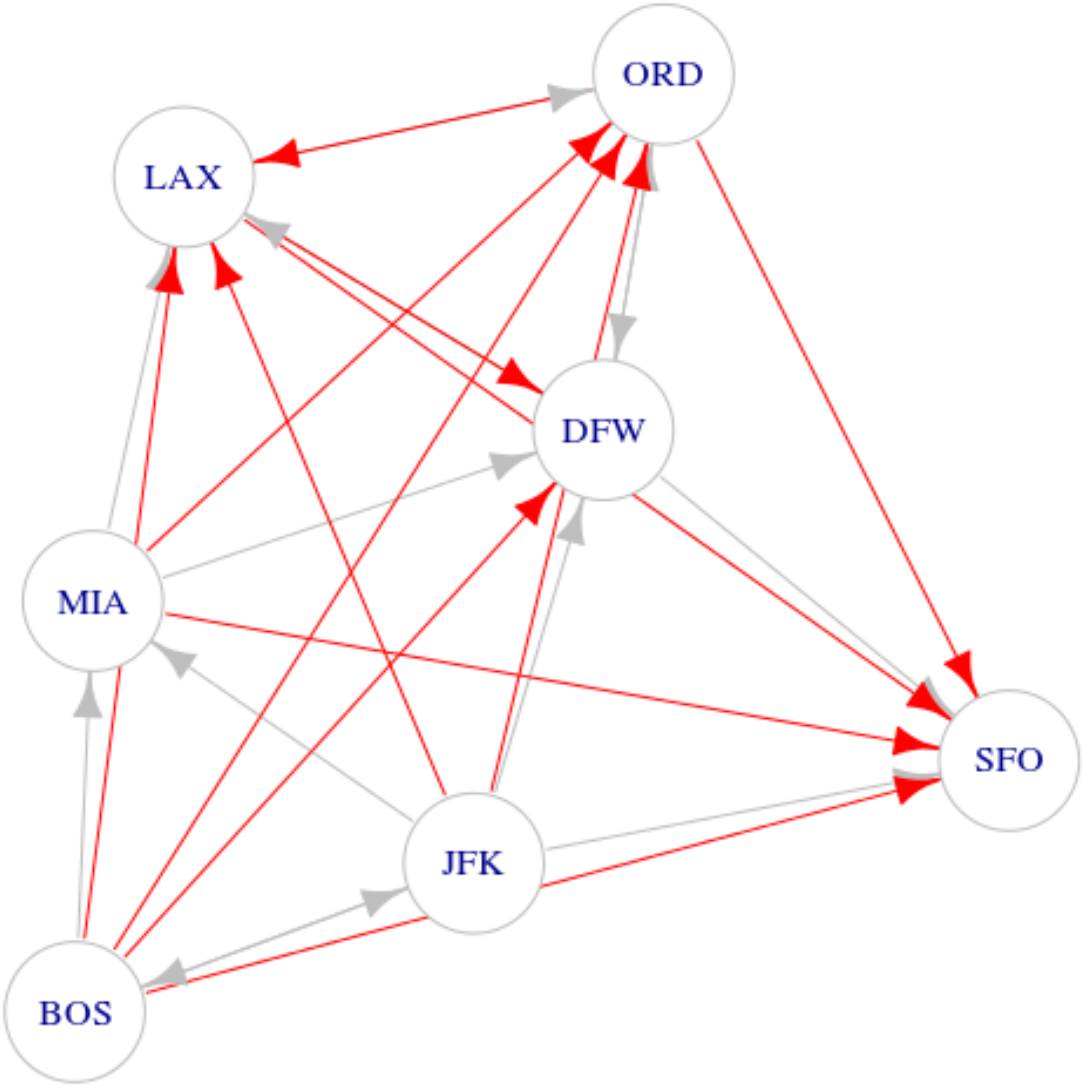
# The Floyd-Warshall Algorithm

## An Example



# The Floyd-Warshall Algorithm

## An Example

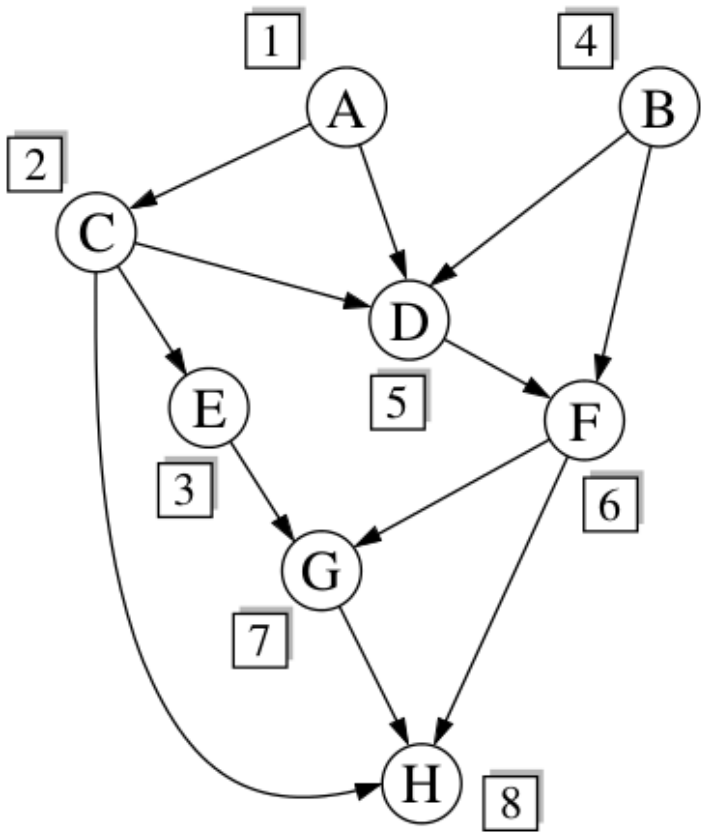


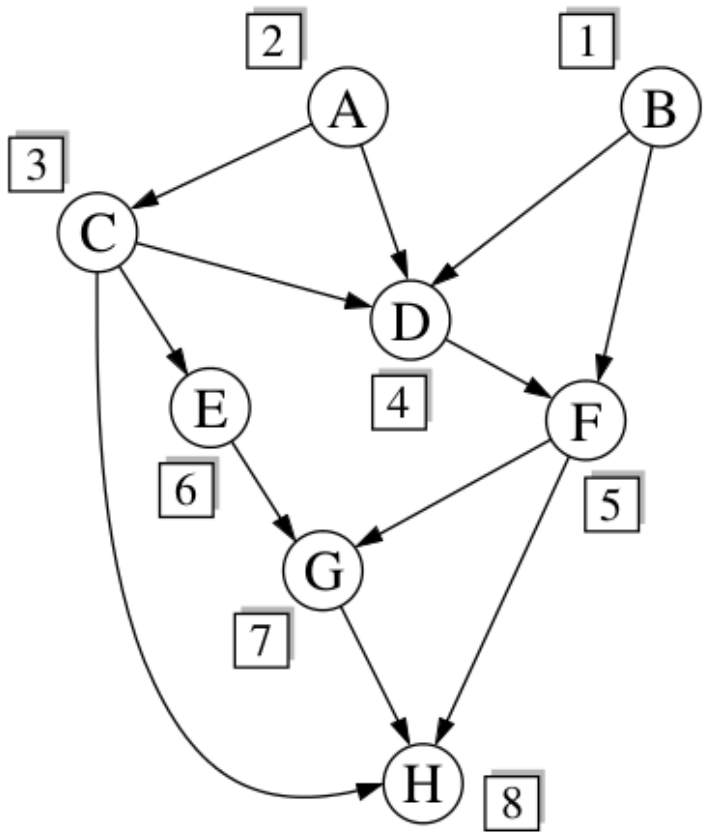
# Directed Acyclic Graph

Directed graphs without directed cycles are quite common. For example:

- Prerequisites between courses of a degree program
- Inheritance between classes of an object-oriented program
- Inheritance between directories in a file system

# Directed Acyclic Graph



# Directed Acyclic Graph

## Topological Ordering

Let $\overrightarrow{G}$ be a directed graph with $n$ vertices. A **topological ordering** of $\overrightarrow{G}$ is an ordering $v_1, \ldots, v_n$ of the vertices of $\overrightarrow{G}$ such that for every edge $(v_i, v_j)$ of $\overrightarrow{G}$, it is the case that $i < j$.

A topological ordering is an ordering such that any directed path in $\overrightarrow{G}$ traverses vertices in increasing order (a topological ordering is not necessarily unique)

**Proposition**
$\overrightarrow{G}$ has a topological ordering if and only if it is acyclic.

The last **Proposition** suggests an algorithm for computing a topological ordering of a directed graph, called **topological sorting**

The implementation uses a dictionary `incount` to map each vertex $v$ to a counter that represents the current number of incoming edges to $v$, excluding those coming from vertices that have previously been added to the topological order.

# Directed Acyclic Graph - Topological Sorting

```
def topological_sort(g):
  topo = []
  ready = []
  incount = {}
  for u in g.vertices():
    incount[u] = g.degree(u, False)
    if incount[u] == 0:
      ready.append(u)
  while len(ready) > 0:
    u = ready.pop()
    topo.append(u)
    for e in g.incident_edges(u):
      v = e.opposite(u)
      incount[v] -= 1
      if incount[v] == 0:
        ready.append(v)
  return topo
```

**Proposition**

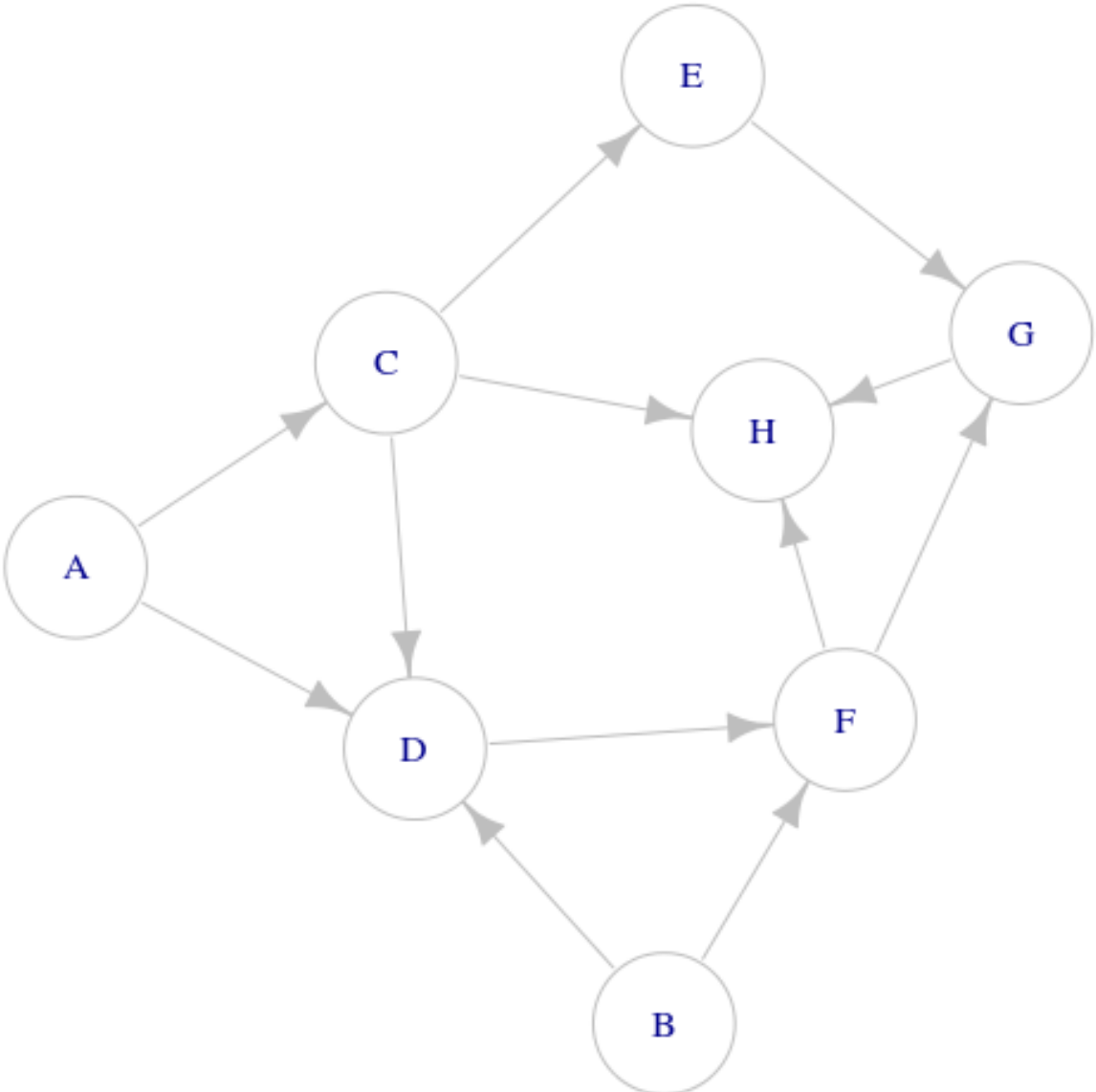
Let $\overrightarrow{G}$ be a directed graph with $n$ vertices and $m$ edges, using an adjacency list representation. The topological sorting algorithm runs in $O(n+m)$ time using $O(n)$ auxiliary space, and either computes a topological ordering of $\overrightarrow{G}$ or fails to include some vertices, which indicates that $\overrightarrow{G}$ has a directed cycle.

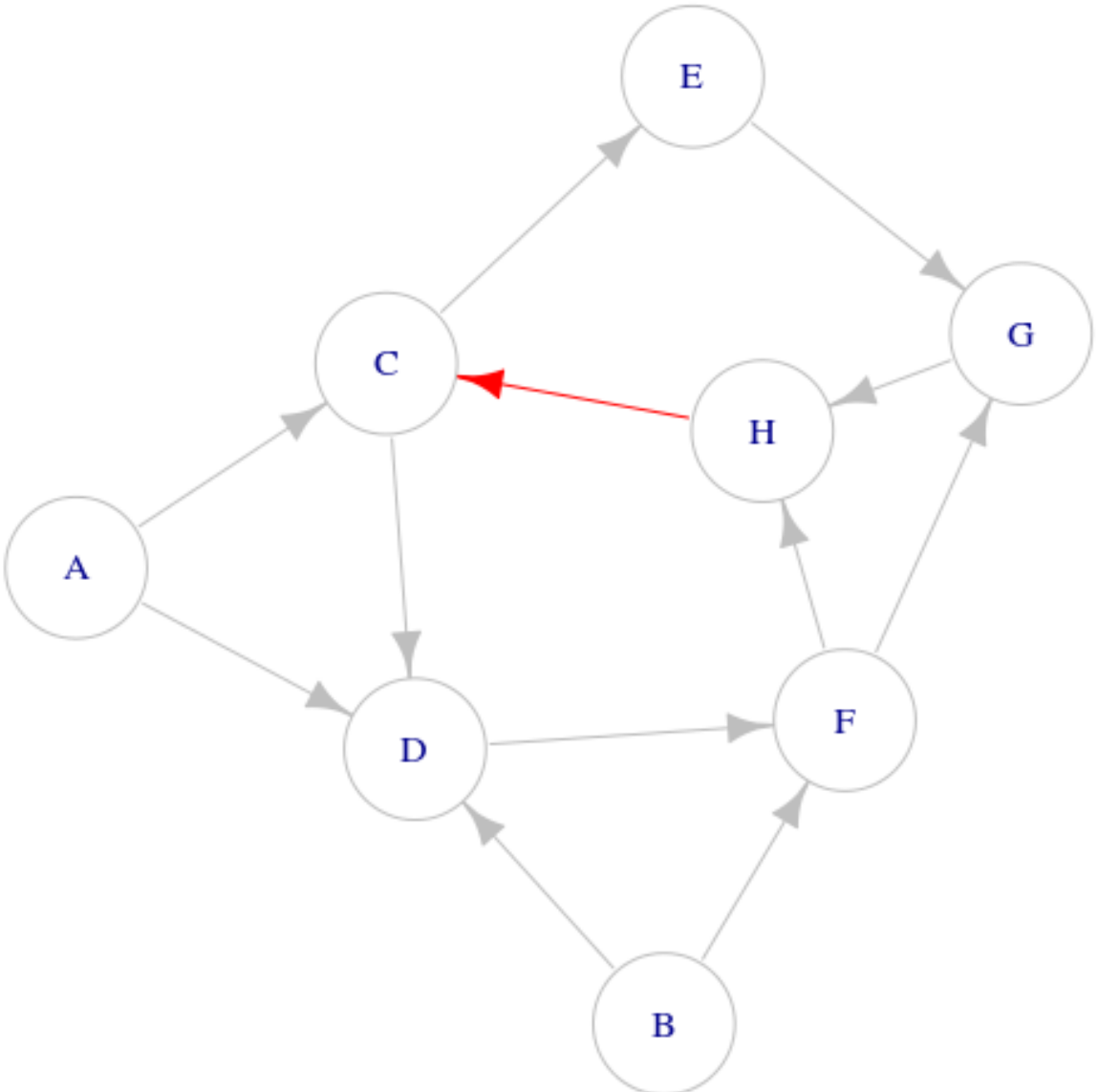
# Directed Acyclic Graph

## Topological Ordering

```
def topological_sort(g):
  topo = []
  ready = []
  incount = {}
  for u in g.vertices():
    incount[u] = g.degree
        (u, False)
    if incount[u] == 0:
      ready.append(u)
  while len(ready) > 0:
    u = ready.pop()
    topo.append(u)
    for e in g.
        incident_edges(u):
      v = e.opposite(u)
      incount[v] -= 1
      if incount[v] == 0:
        ready.append(v)
  return topo
```

- The initial recording of the $n$ in-degrees uses $O(n)$ time based on the `degree` method
- A vertex $u$ is visited by the topological sorting algorithm when $u$ is removed from the ready list
- A vertex $u$ can be visited only when `incount(u)` is zero, namely all its predecessors (vertices which outgoing edges into $u$) were previously visited
- All the outgoing edges of each visited vertex are traversed once, so its running time is proportional to the number of outgoing edges of the visited vertices


E
G
C
H
A
D
F
B


E
C
G
H
A
D
F
B