Graphs: Basic Concepts

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais

1st Semestre 2020/2021

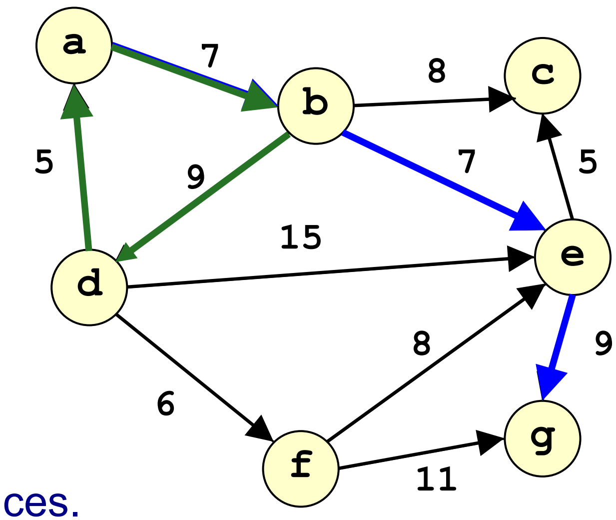
Graphs

- Graphs are a very common data structure that is useful to model a number of “network” applications, where a number of “agents” have direct connections between (some of) them.
- They range from networks of physical services (telecommunications, roads, water distribution) to more virtual services (e.g. social networks) or even to more abstract models (neighbouring countries, teams playing in several competitions, …).
- Formally, a **graph** is defined as a pair **<V,E>** where
  - **V** is a set of **vertices** (or **nodes**)
  - **E** is a set of **edges** (or **arcs**), each connecting two of the vertices
- Two characteristics of the edges, weights and direction, might be considered, leading to different types of graphs:
  - **Weighted Graphs** – Each edge has a weight, usually a positive number
  - **Directed Graphs –** Each edge has a direction, connecting one vertice to another, but not the other way round

Graphs

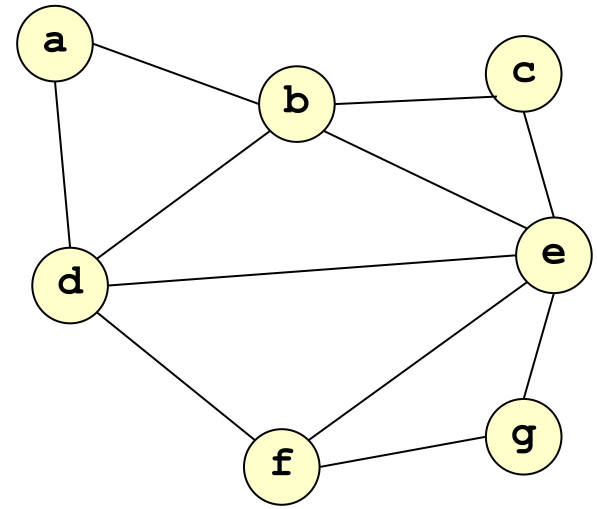
Example:

- An unweighted, undirected graph
- A weighted, undirected graph
- A weighted, directed graph



- A **path** is a sequence of connected vertices.
  - **Example**: Path: a → b → e → g
  - **Note**: A path is directional, even if the underlying graph is not.
- A **cycle** is a path starting and ending in the same vertex.
  - **Example:** Cycle: a → b → d → a

Graphs



- Two nodes are **adjacent (or neighbours)** if there is an edge between them.
  - Example: **adjacent(e,f)** but not **adjacent(a,g)**
- The **degree** of a vertex is the number of its adjacent vertices
  - Example: **degree(e) = 5**, **degree(b) = 4**
- A **graph ordering** is the assignment of a total order to the nodes of the graph, (i.e. the assignment of values **1..n** to the **n** nodes of a graph)
  - Example: **O = a < b < c < d < e < f < g**
- The **width of a node given a graph ordering**, is the number of adjacent nodes lower in the ordering.
  - Example: **width(e,O) = 3** , i.e. nodes **b,c,d** are lower in **O**
- The **width of a graph given a graph ordering**, is the maximum width of its nodes given that ordering.
  - Example: **width(G,O)** = 3 , since **e** is the node with highest width in **O**
- The **width of a** graph is the minimum width of the graph over all its orderings.

Properties of Graphs

- In general, given a graph, there are several problems that may be considered to compute some properties of the graphs, such as:
  - **Connectedness:** Is there a path **connecting** any two vertices of a graph?
  - What is the **shortest path** (number of edges, sum of the edges weights) between any two vertices?
  - What is the **width** of a graph?
  - Are there **cycles** in the graph, or is it a **tree** (i.e. with a unique path between two vertices, or equivalently the graph has width 1)?
  - What is the **shortest spanning tree**?
  - Are there **Hamiltonian** cycles in the graph (including all vertices only once – except the initial/final vertex). Which one(s) is the **shortest**?
  - Are there **cliques** in the graph - subset of the graph where any two nodes are adjacent). Which one(s) is **maximal** (have more nodes).
  - Is it possible to **colour** a graph with a set of colours, such that two adjacent vertices have different colours? What is the **minimum** cardinality of such set?

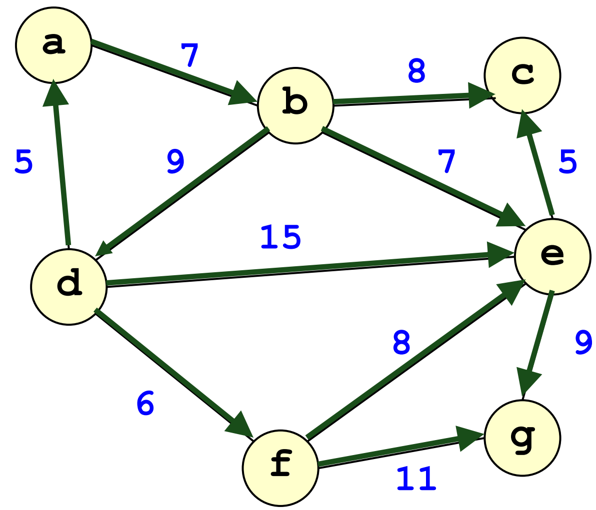
Properties of Graphs

- The problems above, and many others, are typically posed in many applications, and so a number of algorithms have been studied to solve them.
- But before studying some of these algorithms, it is important to adopt a **representation** (or **encoding**) for the implementation of a graph.
- Here we will present the two most common encodings:
  - **Adjacency matrix.**
  - **Adjacency lists.**
- The **adjacency matrix** is possibly the most intuitive way of implementing a graph. Given a **graph** with **n** vertices and some graph ordering, the adjacency matrix is a **square n × n Boolean matrix G**, whose elements $\mathbf{G_{i,j}}$ contain information about the edges between nodes **i** and **j**.
  - In an unweighted graph, the elements are Booleans
  - In a weighted graph, the elements are the weights
  - In a undirected graph the matrix is symmetric, otherwise it is usually asymmetric.

Graphs

Example:

	a	b	c	d	e	f	g
a	0	7	-1	-1	-1	-1	-1
b	-1	0	8	9	7	-1	-1
c	-1	-1	0	-1	-1	-1	-1
d	5	-1	-1	0	15	6	-1
e	0	-1	5	-1	0	-1	9
f	-1	-1	-1	-1	8	0	11
g	-1	-1	-1	-1	9	-1	0



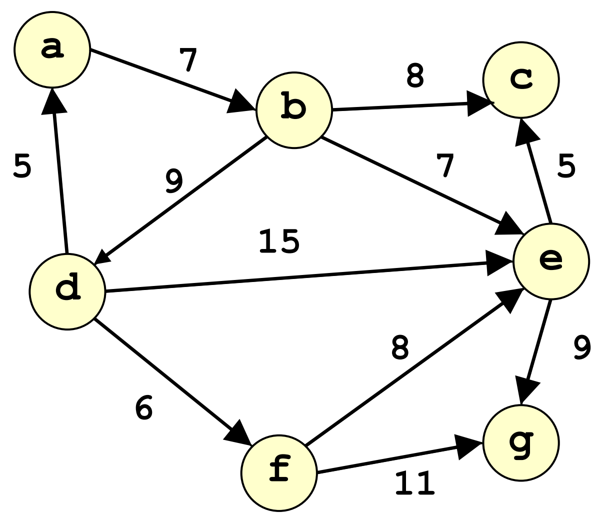
Properties of Graphs

- The adjacency matrix is a very inefficient representation of **sparse** graphs, i.e. where only a "few" of the potential arcs are presented. In this case, of the n^2 elements of the matrix only a (small) fraction of them are non-zero.

- To avoid this waste of space, one may adopt an **adjacency lists**, i.e. a set of lists each representing, for each node, the information about its neighbours (taking into account the directedness).

	a	b	c	d	e	f	g
a	0	7	-1	-1	-1	-1	-1
b	-1	0	8	9	7	-1	-1
c	-1	-1	0	-1	-1	-1	-1
d	5	-1	-1	0	15	6	-1
e	0	-1	5	-1	0	-1	9
f	-1	-1	-1	-1	8	0	11
g	-1	-1	-1	-1	9	-1	0

a	b:7		
b	c:8	d:9	e:7
c			
d	a:5	e:15	f:6
e	e:5	g:9	
f	e:8	g:11	
g			



- The space required is thus **O(|E|)** which is much less than **O(|V²|)** for sparse graphs.

Types of Algorithms

- As we will see, some of these problems require algorithms whose asymptotical complexity is **polynomial** on **n**, the **input size** of the problem. Assuming that reads from and writes to memory are *basic operations,* **polynomial algorithms** require **O(n^k)** *basic operations,* where **k** is an integer, typically small.
- Problems that can be solved by polynomial algorithms are said to be in class **P**.
- Other algorithms have exponential complexity, i.e. require **O(k^n)** *basic operations.* Problems that can only be solved by these are said to be in class **NP**.
- Take a computer where each elementary operation takes 1 nsec. The following table shows the “practical” consequences of the problem being in **P** or in **NP**. Here the size **n** is the size of an input vector or matrix, or the size **|V|** or **|E|** of a graph.

  n^1: **Search** in a vector; **n^2**: **Sorting** (naïf) a vector; **n^3**: Matrix **multiplication**

n	10	20	30	40	50	60	70
n^1	10 nsec	20 nsec	30 nsec	40 nsec	50 nsec	60 nsec	70 nsec
n^2	100 nsec	400 nsec	900 nsec	1.6 μsec	2.5 μsec	3.6 μsec	4.9 μsec
n^3	1 μsec	8 μsec	27 μsec	64 μsec	125 μsec	216 μsec	343 μsec
2^n	1 μsec	1 msec	1 sec	18 min	13 days	37 years	37 K years

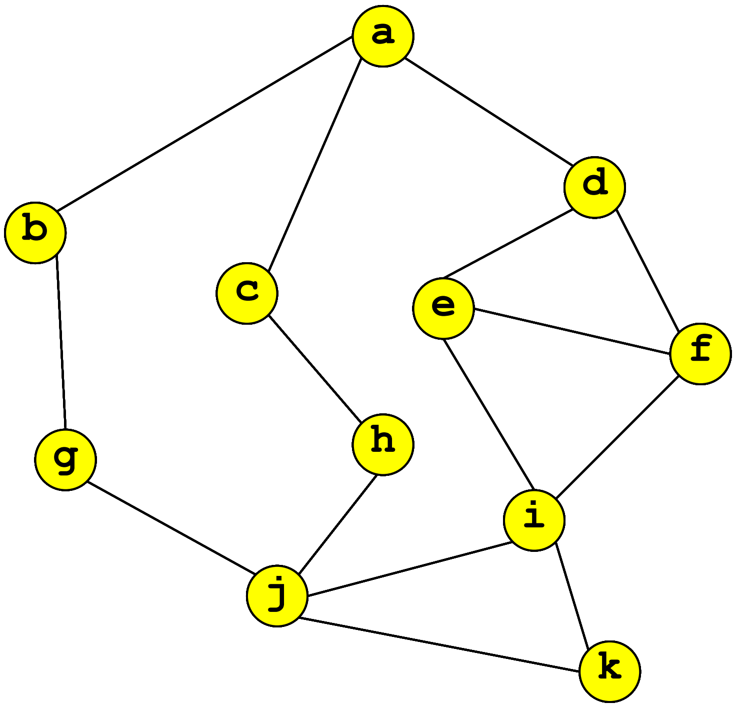
Connectedness of Graphs

Problem (Connectedness): Check whether a graph G is connected.

- The definition of connectedness of a graph depends on its type:
  - An undirected graph is **connected** if there is a path between any two nodes of the graph.
  - A directed graph is **strongly connected** is there is a path between any two nodes of the graph, respecting the direction of the its arcs.
  - A directed graph is **weakly connected** is there is a path between any two nodes of the corresponding undirected graph.
- Here we will study the case for the undirected graphs, which is easier to decide, since paths (being reflexive, symmetric and transitive) create classes of equivalence.
- We will thus present an algorithm to check the connectedness of undirected graphs, by checking whether all its nodes are in the same equivalence class.

Properties of Graphs

Is the graph connected?

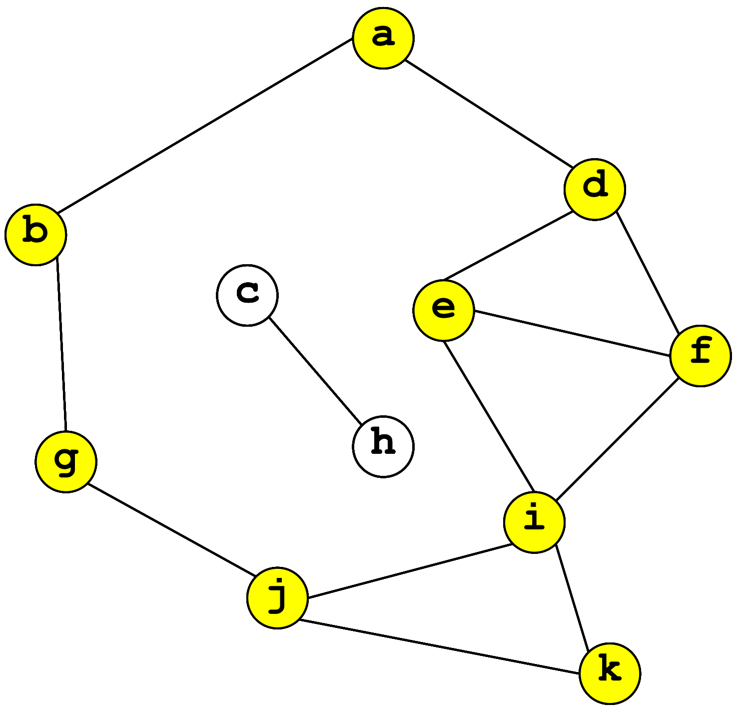



```
Fr = [a];               Out = [b,c,d,e,f,g,h,i,j,k]
- NewFr = [b,c,d]       NewOut = [e,f,g,h,i,j,k]

Fr = [b,c,d];           Out = [e,f,g,h,i,j,k]
- NewFr = [g,h,e,f]     NewOut = [i,j,k]

Fr = [g,h,e,f]          Out = [i,j,k]
- NewFr = [i,j]         NewOut = [k]

Fr = [i,j]              Out = [k]
- NewFr = [k]; NewOut = []

Fr = [k]; Out = []
- NewFr = []; NewOut = []

Fr = []; Out = []
```

The Out list is empty. The graph is connected!

Properties of Graphs

Is the graph connected?




```
Fr = [a]; Out = [b,c,d,e,f,g,h,i,j,k]
– NewFr = [b,d]; NewOut = [c,e,f,g,h,i,j,k]

Fr = [b,d]; Out = [c,e,f,g,h,i,j,k]
  – NewFr = [g,e,f]; NewOut = [c,h,i,j,k]

Fr = [g,e,f]; Out = [c,h,i,j,k]
  – NewFr = [i,j]; NewOut = [c,h,k]

Fr = [i,j]; Out = [c,h,k]
  – NewFr = [k]; NewOut = [c,h]

Fr = [k]; Out = [c,h]
  – NewFr = []; NewOut = [c,h]

Fr = []; Out = [c,h]
```

The Out list is NOT empty. The graph is NOT connected!

Properties of Graphs

- The algorithm presented can be implemented as the following function (where all sets are implemented as lists. In fact set **In** is not needed and is not considered).
- List **Fr** is initialised with one arbitrary node (here we chose node 0)
- List **Out** is initialised with the other nodes.
- The iterations proceed while the frontier (list **Fr**) is not empty.
- In every iteration, both lists **Fr** and **Out** are updated.
- After the last iteration, the connectedness is equated to having the **Out** list empty (both the **connected** Boolean and the remaining **Out** list are returned).

```
def connected(G):
    """This function returns a Boolean that is True if Graph G
    is connected and False otherwise"""
    Fr = [0]
    Out = [i for i in range(1,len(G))]
    while len(Fr) > 0:
        # move nodes from Out to Fr if connected
        ...
    return (len(Out) == 0, Out)
```

Properties of Graphs

- The core of the algorithm is the updating of lists **Fr** and **Out** in every iteration.
  - The new frontier (list **NewFr**) is initialised to empty.
  - Then every node in **Out** is checked for a neighbour in the frontier (list **Fr**).
  - If there is one such node, the node is appended to list **NewFr**.
- After all **Out** nodes have been checked,
  - All nodes in the new frontier are removed from list **Out**.
  - The list Fr is updated with the nodes of the new frontier.

```
while len(Fr) > 0:
  # move nodes from Out to Fr if connected
  NewFr = []
  for k in Out:
      if connected_to_Frontier(k, G, Fr):
          NewFr.append(k)
  for k in NewFr:
      Out.remove(k)
  Fr = NewFr
```

Properties of Graphs

- The complete function is shown bellow:

```
def connected(G):
    """This function returns a Boolean that is True if Graph G
    is connected and False otherwise"""
    Fr = [0]
    Out = [i for i in range(1,len(G))]
    while len(Fr) > 0:
        # move nodes from Out to Fr if connected
        NewFr = []
        for k in Out:
            if connected_to_Frontier(k, G, Fr):
                NewFr.append(k)
        for k in NewFr:
            Out.remove(k)
        Fr = NewFr
    return (len(Out) == 0, Out)
```

Properties of Graphs

- The function above, uses an auxiliary function, **`move_to_Frontier`**, that tests whether a node, **k** (from list **Out**) is connected to a node in the frontier (list **Fr**), according to the given graph G.
- This function can be implemented straightforwardly:

```
def connected_to_Frontier(k, G, Fr):
    """ Moves node k from the Out List to the Frontier list if
    there is a link between any node in the Frontier and k"""
    for i in Fr:
        if G[i][k] >= 0:
            return True
    return False
```

- For every node in **Fr** the connection is tested.
- If the connection exists (an arc with value ≥ 0) the function returns **True**.
- If no connection exists with any node in Fr, the function returns **False**.

Properties of Graphs

- In this case the worst-case time complexity is obtained by analysing the algorithm for a Graph with **n** nodes.

```
...
while len(Fr) > 0:
    for k in Out:
        if connected_to_Frontier(k, G, Fr):
        ...
```

- Since a node can only be in the frontier (**Fr** list) during one iteration of the **while** loop (in the next iteration it removed from the list), the **while** loop can only be executed **n** times.
- Each node k can thus be analysed at most **n** times (with **move_to_Frontier**).
- To check it against all the nodes in the frontier **Fr**, requires at most **n** comparisons.
- Hence, the number of comparisons is at most **n*n*n** and the worst case complexity of the algorithm is no more than

$O(n^3)$.

Properties of Graphs

- In fact, the number of comparisons is less than n^2, since in each iteration there is at least one less node to be considered (i.e. removed from the Fr list).

```
    while len(Fr) > 0:
        for k in Out:
            if connected_to_Frontier(k, G, Fr):
```

- Hence, assuming there is only one node in list Fr, the actual number of comparisons is
  - (n-1) in the 1st execution of the loop
  - (n-2) in the 2nd execution of the loop
  - …
  - 1 in the n-1th execution of the loop

- Hence the number of comparisons is (n-1)+(n-2)+…+ 1 i.e $(1+n-1)(n-1)/2 \approx n^2/2$

- Of course there are operations for addition and removal of elements in the lists, but these can be at most n operations in each of the while loop, hence $O(n^2)$ operations which do not change the asymptotic complexity of

$O(n^3)$.

Properties of Graphs

- Note that the complexity that we considered is a worst case complexity.
- For example if the first node used in the Fr list is not connected to any other node, than only **n-1** comparisons between the node and the other are made, the complexity becomes

O(n).

- On the other hand, if half the out nodes are moved to the frontier in each iteration, there will be log(n) iterations of the while loop, each with (n/2, n/4, …, 1 nodes in the Fr and Out lists.
- Then the number of comparisons are

$$n/2 * n/2 + n/4 * n/4 + n/8 * n/8 \dots + n/n * n/n =$$

$$= n^2 (1/ 2^2 + 1/ 2^4 + 1/ n^{2n})$$

$$= n^2 / 2^2 (1 + 1/ 2^2 + 1/ 2^n) \approx 1.5 *n^2 / 4$$

and the complexity becomes, approximately,

O(n²)