Universidade Nova de Lisboa
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT
Faculdade de Ciências e Tecnologia

Graphs: Basic Concepts

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais
1st Semestre 2018/2019

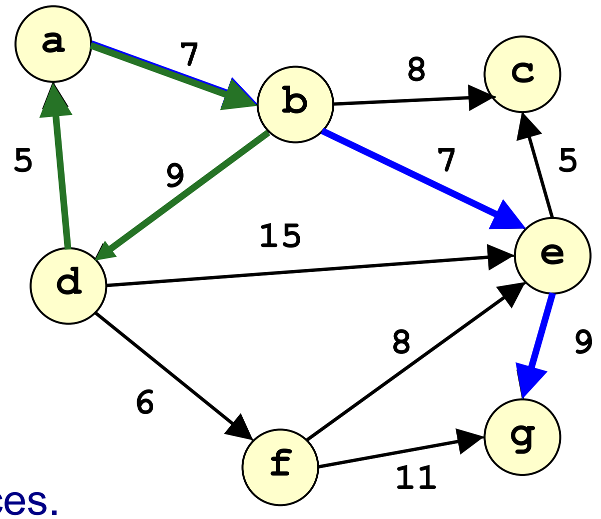
Graphs

- 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

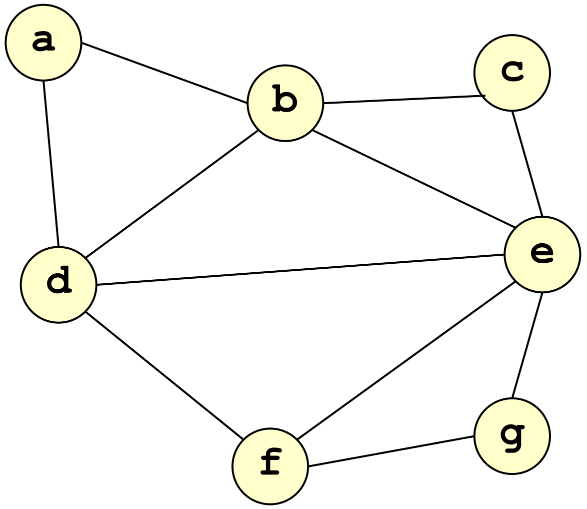
Example:

- 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?

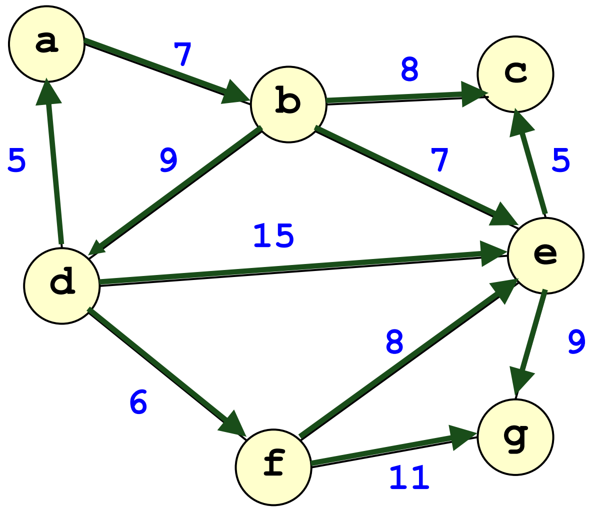
Properties of Graphs

- 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	0	0	0	0	0
B	0	0	8	9	7	0	0
C	0	0	0	0	0	0	0
D	5	0	0	0	15	6	0
E	0	0	5	0	0	0	9
F	0	0	0	0	8	0	11
G	0	0	0	0	0	0	0



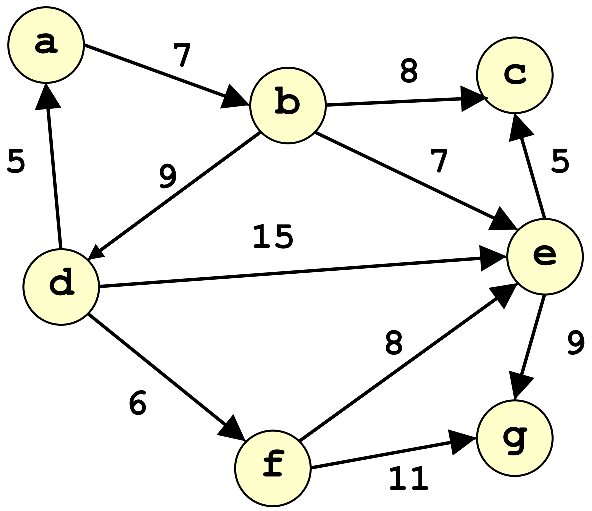
Properties of Graphs

- 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	0	0	0	0	0
B	0	0	8	9	7	0	0
C	0	0	0	0	0	0	0
D	5	0	0	0	15	6	0
E	0	0	5	0	0	0	9
F	0	0	0	0	8	0	11
G	0	0	0	0	0	0	0

A	B:7		
B	C:8	D:9	E:7
C			
D	A:5	E:15	F: 6
E	C: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**

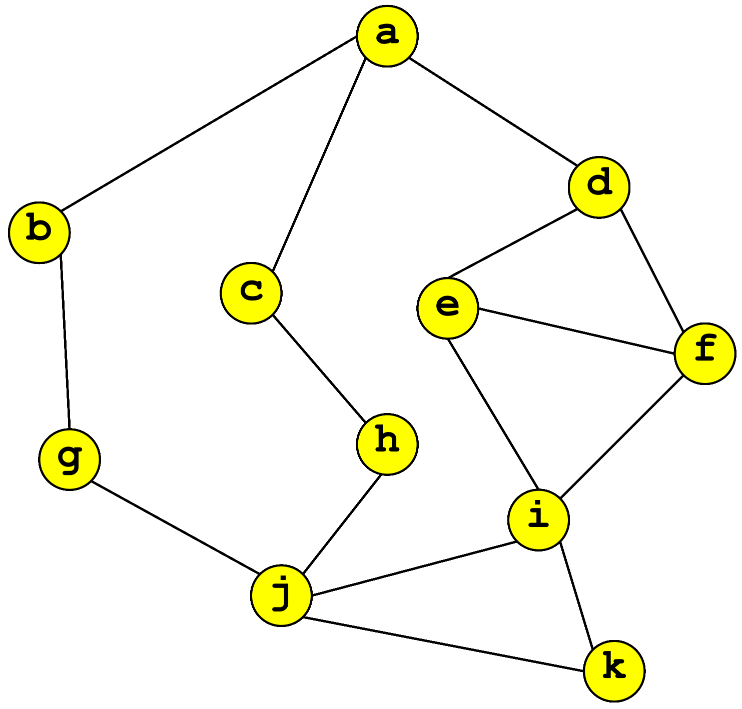
n	**10**	**20**	**30**	**40**	**50**	**60**	**70**
n^1	10 nsec	20 nsec	30 nsec	40 nsec	50 nsec	60 nsec	70 nsec
n^2	100 nsec	400 nsec	900 nsec	1.6 μsec	2.5 μsec	3.6 μsec	4.9 μsec
n^3	1 μsec	8 μsec	27 μsec	64 μsec	125 μsec	216 μsec	343 μsec
2^n	1 μsec	1 msec	1 sec	18 min	13 days	37 years	37 K years

Connectedness of Graphs

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

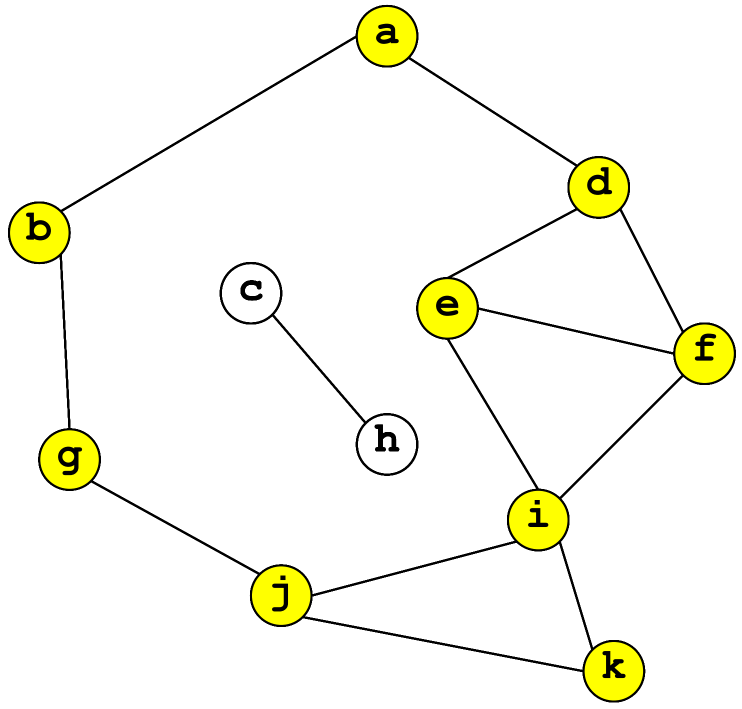
- The definition of connectedness of a graph depends on its type:
    - An undirected graph is **connected** if there is a path between any two nodes of the graph.
    - A directed graph is **strongly connected** is there is a path between any two nodes of the graph, respecting the direction of the its 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 n the same equivalence class.

Properties of Graphs



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

Properties of Graphs



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

Properties of Graphs

- The informal algorithm presented can be implemented as the following function:

```
function b = connected(G);
   In = []; Fr = [1]; Out = 2:size(G,1);
   while length(Fr) > 0
      % move all the nodes from Out to
      %  - NewFr, if they have a neighbour in In;
      %  - NewOut, otherwise
      ...
      In = [In,Fr]; Fr = NewFr; Out = NewOut;
   end
   b = (length(Out) == 0);
end
```

- Set **In** is initialised to empty, a node is chosen arbitrarily to initialise set **Fr** (here we chose node 1), and the others to initialise **Out**. All sets are represented as vectors.
- The iterations proceed while the frontier (set **Fr**) is not empty.
- In every iteration, the new frontier (**NewFr**) and the remaining nodes (**NewOut**) are computed. The previous frontier is added to set **In**, and the frontier **Fr** is updated.
- After the last iteration, the connectedness is equated to all nodes being in set **In**.

Properties of Graphs

- The core of the algorithm is the construction of sets **NewFr** and **NewOut** in every iteration (again implemented as vectors).

```
% move all the nodes from Out to NewFr or NewOut
NewFr = []; NewOut = [];
for j = 1:length(Out)
    i = 1; inserted = false;
    while i <= length(Fr) && !inserted
        if G(Fr(i),Out(j))
            NewFr = [Out(j), NewFr]; inserted = true;
        end
        i = i + 1;
    end
    if !inserted NewOut = [Out(j),NewOut]; end
end
```

- Both sets **NewFr** and **NewOut** start being empty
- Then every node in **Out** is checked for a neighbour in the frontier (set **Fr**).
- If there is one such node, and if not done so before, the node is inserted in set **NewFr**, and the node is marked as **inserted**.
- If no neighbour is found , the out node is added to set **NewOut**.

Properties of Graphs

- The complete function is shown bellow (comments removed):

```
function b = connected(G);
   In = []; Fr = [1]; Out = 2:size(G,1);
   while length(Fr) > 0
      NewFr = []; NewOut = [];
      for j = 1:length(Out)
         i = 1; inserted = false;
         while i <= length(Fr) && !inserted
            if G(Fr(i),Out(j))
               NewFr = [Out(j), NewFr]; inserted = true;
            end
            i = i + 1;
         end
         if !inserted NewOut = [Out(j),NewOut]; end
      end
      In = [In,Fr]; Fr = NewFr; Out = NewOut;
   end
   b = (length(Out) == 0);
end
```

Properties of Graphs

- An discussed, an important issue in algorithm design is to study their complexity, which can be considered for the best, worse or average case.

- In this case the time complexity is similar for all cases (if the graph is connected). In fact the algorithm performs several iterations that compare nodes in the frontier with outside nodes

```
   while length(Fr) > 0
      ...
      for j = 1:length(Out)
         ...
         while i <= 1:length(Fr)
            if G(Fr(i),Out(j))
               ...
```

- Since a node can only be in the frontier (**Fr** set) during one iteration (in the next one it is moved to the **In** set), the pairs **< i, j >** are never repeated, nor reversed: (if in some iteration node **i** is in **Fr** and j is in **Out**, then it will never be the case that in subsequent iterations **j** is in **Fr** and **i** in **Out**).

- Hence, being **n** the number of vertices in the graph (**n = |V|**) the number of comparisons is at most **n*(n-1)/2**.

Properties of Graphs

- In every iteration "simple" operations are performed, namely
  - adding a value to a vector, as in
    - **`NewFr = [j,NewFr]`** and **`NewOut = [j,NewOut]`**
  - appending two vectors, as in
    - **`In = [In,Fr]`**
- If suitably implemented, these operations can be performed with a small number of elementary operations (reads from and writes into memory).
- Hence, the asymptotical complexity of this algorithm is **quadratic** on the number of nodes of the graph (i.e. the algorithm is in **P**)
  - **$O(|V|^2)$**
- Notice that for graphs represented by adjacency lists, only pairs in the lists would be checked (instead of **`G(i,j)`** checks), and the complexity would be **O(E),** which is much better in the case of sparse graphs.