Advanced Graph Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

About myself...

- Associate Professor (*Conferentiar*) at the Univ. of Bucharest, Faculty of Mathematics and Computer Science.

- Research Scientist at the National Institute of Research and Development in Informatics (ICI).

- PhD in Nice-Sophia Antipolis, France.

- Various visits/internships in the US, Chile, Germany,...

Research in **Graph Algorithms** *and Data Structures*

Come to see me if...

- **You want to learn more about the class! =)**

Some useful links:

  - https://algo.cs.uni-tuebingen.de/wg2022/
  - https://jgaa.info/

- **You need help for your Bachelor/Master thesis**

  - Plenty of my problems need good students to find good solutions, and/or implement some proposed solutions.

- **You want to discuss about internship/PhD/job opportunities...**

About the class

- Graphs are far-reaching generalizations of well studied Data Structures (Lists, Search Trees, Heaps, Automata, etc.)

- Complements your other algorithmic classes with more advanced **decomposition** tools and **structural/metric** insights for solving graph problems.

- A good prerequisite for some Master classes (*e.g.*, *Complex Networks*, *Advanced Programming Techniques*, . . .)

- Modern presentation of the field of Algorithmic Graph Theory.

Practical Information

- All materials (slides + documents) to be put on Moodle.

- My website – with contact information

  https://sites.google.com/view/guillaume-ducoffes-homepage/home

  Additional documents can be put on it – upon request

- **Attendance is NOT mandatory**. But...
  - Bonus for attendance + participation during seminars (max. 1p)

Your **grades**

- Written test ("*Proba scrisa*"): 60 %
  - Either pseudo code or C++/Python-like code
  - A significant part of the grade devoted to algorithm analysis (justify correctness, complexity, etc.)
- Practical exam ("*Colocviu*"): 40 %
  - In C++ or Python.
  - Two sets of problems to be solved.

Final grade: $\min\{10, .6 * \text{Written Test} + .4 * \text{Practical Exam} + \text{Bonus}\}$

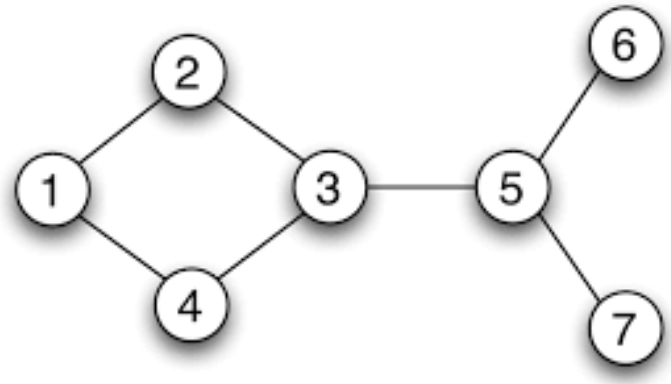
There shall be NO "punct din oficiu".

Graphs

- **(Undirected) Graph**: $G = (V, E)$

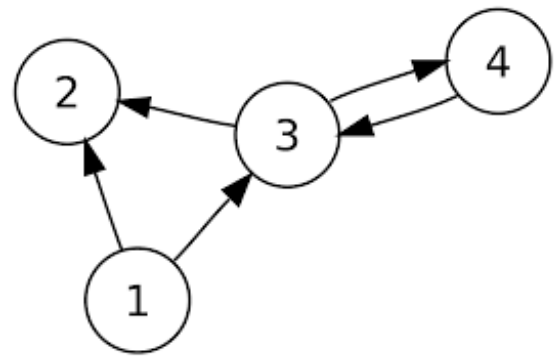
V = vertices = networks units

E = edges



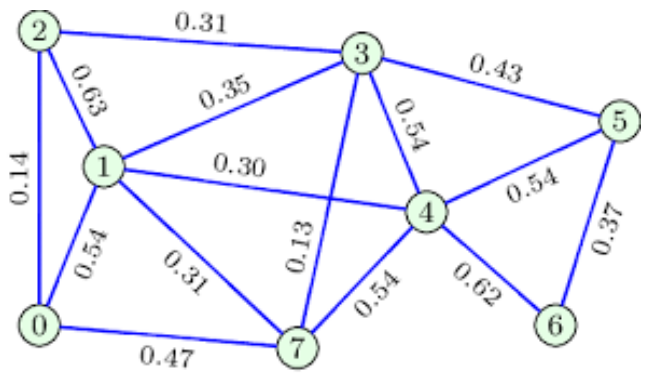
- **Directed Graph**: $\overrightarrow{G} = (V, A)$

A = arcs



- **Weighted Graph**: $G = (V, E, w)$

$w : E \to \mathbb{R}$ is a weight function (length, cost)



There may be loops, multiple edges (= Multigraphs), vertex-weights, etc.

Terminology

• If $e = uv$ is an edge, then vertices u and v are **adjacent**, and they are **incident** to edge e.

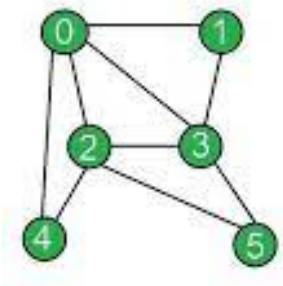
• The **degree** $d(v)$ of a vertex v equals its number of incident edges (loops are counted twice!).

• The (open) **neighbourhood** of a vertex v is the set $N(v)$ of all its adjacent vertices. The **closed neighbourhood** is defined as $N[v] = N(v) \cup \{v\}$.

• The neighbourhood of a set M equals $N(M) = \bigcup\{N(v) \setminus M \mid v \in M\}$. The closed neighbourhood is defined as $N[M] = N(M) \cup \{M\}$.

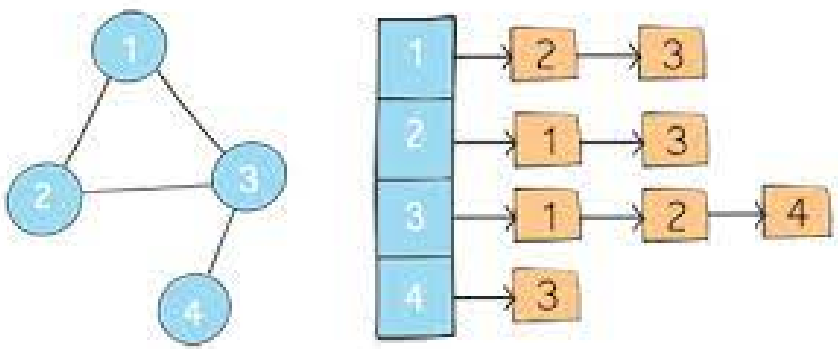
Graph representations

- **Adjacency matrix**



	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

- **Adjacency list**



undirected graph and its adjacency list

Comparison

Throughout ALL this semester:

- **n = number of vertices;**
- **and m = number of edges.**

- Adjacency matrix requires $\mathcal{O}(n^2)$ space
- Adjacency list requires $\mathcal{O}(n+m)$ space, which is optimal.

Remark 1: $m \leq \binom{n}{2} = \mathcal{O}(n^2)$, therefore adjacency list is *never worse* than adjacency matrix.

Remark 2: if there is no isolated vertex, $m \geq n/2$, and therefore we can simplify $\mathcal{O}(n+m) = \mathcal{O}(m)$.

Basic operations on graphs

- Enumerate all edges
- Enumerate all neighbours of a vertex
- Output the degree of a vertex
- Decide whether two vertices are adjacent
- Addition/Removal of edges/vertices.
- **Contraction** of an edge $e = uv$: replace u, v by some new vertex x whose neighbourhood equals $N(u) \cup N(v) \setminus \{u, v\}$.

With Adjacency matrix

- Enumerate all edges: scan the whole matrix in $\mathcal{O}(n^2)$ time.

With Adjacency matrix

- Enumerate all edges: scan the whole matrix in $\mathcal{O}(n^2)$ time.

- Enumerate all neighbours of a vertex: scan the corresponding line/column of the matrix in $\mathcal{O}(n)$ time.

With Adjacency matrix

- Enumerate all edges: scan the whole matrix in $\mathcal{O}(n^2)$ time.
- Enumerate all neighbours of a vertex: scan the corresponding line/column of the matrix in $\mathcal{O}(n)$ time.
- Output the degree of a vertex: scan the corresponding line/column of the matrix in $\mathcal{O}(n)$ time.

  Better: $\mathcal{O}(1)$ time if we store the degree in a separate variable.

With Adjacency matrix

- Enumerate all edges: scan the whole matrix in $\mathcal{O}(n^2)$ time.
- Enumerate all neighbours of a vertex: scan the corresponding line/column of the matrix in $\mathcal{O}(n)$ time.
- Output the degree of a vertex: scan the corresponding line/column of the matrix in $\mathcal{O}(n)$ time.

  Better: $\mathcal{O}(1)$ time if we store the degree in a separate variable.
- Decide whether two vertices are adjacent: Look at the corresponding cell in the matrix in $\mathcal{O}(1)$ time.

With Adjacency matrix

- Enumerate all edges: scan the whole matrix in $\mathcal{O}(n^2)$ time.
- Enumerate all neighbours of a vertex: scan the corresponding line/column of the matrix in $\mathcal{O}(n)$ time.
- Output the degree of a vertex: scan the corresponding line/column of the matrix in $\mathcal{O}(n)$ time.

  Better: $\mathcal{O}(1)$ time if we store the degree in a separate variable.
- Decide whether two vertices are adjacent: Look at the corresponding cell in the matrix in $\mathcal{O}(1)$ time.
- Addition/Removal of edges: Modify the two corresponding cells in $\mathcal{O}(1)$ time.

With Adjacency matrix (cont'd)

- Addition/Removal of a vertex: Reallocate the actualized matrix in $\mathcal{O}(n^2)$ time.

  Better: use doubling arrays, which support addition/removal of elements at front/back of the vector in amortized $\mathcal{O}(1)$ time.

  $\rightarrow$ Addition of a new vertex requires inserting a new bottom line: in amortized $\mathcal{O}(n)$ time, and a new cell at the end of each previous line: in amortized $n \times \mathcal{O}(1) = \mathcal{O}(n)$ time.

  $\rightarrow$ For vertex removal: swap the corresponding line and column with the *last* line and column of the matrix. It can be done in $\mathcal{O}(n)$ time. Then, delete the last line and column in amortized $\mathcal{O}(n)$ time.
  Requires reference updates for the vertices.

With Adjacency matrix (cont'd)

- Addition/Removal of a vertex: Reallocate the actualized matrix in $\mathcal{O}(n^2)$ time.

  Better: use doubling arrays, which support addition/removal of elements at front/back of the vector in amortized $\mathcal{O}(1)$ time.

  $\rightarrow$ Addition of a new vertex requires inserting a new bottom line: in amortized $\mathcal{O}(n)$ time, and a new cell at the end of each previous line: in amortized $n \times \mathcal{O}(1) = \mathcal{O}(n)$ time.

  $\rightarrow$ For vertex removal: swap the corresponding line and column with the *last* line and column of the matrix. It can be done in $\mathcal{O}(n)$ time. Then, delete the last line and column in amortized $\mathcal{O}(n)$ time.
  Requires reference updates for the vertices.

- **Contraction** of an edge $e = uv$: Reduces to two vertex removals, one vertex addition, and up to $\mathcal{O}(n)$ edge additions.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

- Enumerate all neighbours of a vertex v: scan the corresponding list in $\mathcal{O}(d(v))$ time.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

- Enumerate all neighbours of a vertex v: scan the corresponding list in $\mathcal{O}(d(v))$ time.

- Output the degree of a vertex: we output the list size in $\mathcal{O}(1)$ time.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

- Enumerate all neighbours of a vertex v: scan the corresponding list in $\mathcal{O}(d(v))$ time.

- Output the degree of a vertex: we output the list size in $\mathcal{O}(1)$ time.

- Addition of a new vertex: insertion of a new empty list at the end in amortized $\mathcal{O}(1)$ time.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

- Enumerate all neighbours of a vertex v: scan the corresponding list in $\mathcal{O}(d(v))$ time.

- Output the degree of a vertex: we output the list size in $\mathcal{O}(1)$ time.

- Addition of a new vertex: insertion of a new empty list at the end in amortized $\mathcal{O}(1)$ time.

- Removal of an isolated vertex v: Switch v with the last vertex, then remove the last vertex, in amortized $\mathcal{O}(1)$ time.

Adjacency Testing

For static graphs (no vertex/edge modifications allowed), we may replace adjacency list by adjacency vectors.

If adjacency vectors are **sorted**, then we can test whether u, v are adjacency in $\mathcal{O}(\log \min\{d(u), d(v)\})$ time, using *binary search*.

Adjacency Testing

For static graphs (no vertex/edge modifications allowed), we may replace adjacency list by adjacency vectors.

If adjacency vectors are **sorted**, then we can test whether u, v are adjacency in $\mathcal{O}(\log \min\{d(u), d(v)\})$ time, using *binary search*.

Sorting all adjacency lists can be done naively in $\mathcal{O}(n + m \log n)$ time.

If vertices are labeled $0, 1, \ldots, n-1$, then we can sort adjacency lists in $\mathcal{O}(n + m)$ time and space:

- Create a new graph G' with no edges.
- For $i = 0 \ldots n-1$, traverse the adjacency list of vertex i in the original graph G. For every adjacent vertex j, add i to j's adjacency list in the copy G'.

Reminder: Hash Table

Store a collection of pairs `(key,value)`.

- Three operations:
  - **int** `lookup(e);` Returns the value associated to some key `e` (if it is present in the table)
  - **void** `insert(e, v);` Adds a new pair `(e,v)` (if the key `e` is not already present in the table)
  - **void** `delete(e);` Deletes a pair `(key,value)` – given its key `e`.

Each operation runs in expected $\mathcal{O}(1)$ time.

Applications to Adjacency Testing

We store every edge in a Hash table. To every edge uv, we associate pointers its positions in the respective adjacency lists of u and v.

- Adjacency testing: Lookup in the table in expected $\mathcal{O}(1)$ time.
- Addition of an edge uv: Insertion of u (v, resp.) at the bottom of the adjacency list of v (u, resp.). Insertion in expected $\mathcal{O}(1)$ time in the Hash table, along with pointers to the bottom positions in the respective adjacency lists of u and v.
- Removal of an edge uv: Lookup in the Hash-table in order to find the positions of u, v in the respective adjacency lists of v, u. Then, removal in both adjacency lists and in the table.
- Removal of a vertex: removal of every incident edge + removal of an isolated vertex.

Tentative schedule for this semester

- Graph searches.
- Cographs.
- Modular decomposition.
- Partitive families and their applications to Graph decomposition.
- Chordal graphs.
- Clique-separator decomposition.
- Tree decompositions.
- Parameterized complexity (mostly, Treewidth).
- Courcelle's theorem and relatives.
- Perfect graphs and Linear Programming.
- Planar graphs.
- Dynamic graph algorithms.

Questions