# Graph representations, L3 Cachan 2015–2016

Michel Habib
habib@liafa.univ-Paris-Diderot.fr
http://www.liafa.univ-Paris-Diderot.fr/~habib

Novembre 2015

# Notations

Here we deal with finite loopless and simple undirected graphs.
For such a graph $G$
we denote by $V(G)$ the set of its vertices
and by $E(G)$ the edge set
By convention $|V(G)| = n$ and $|E(G)| = m$

# Bound on the number of edges

## Triangle free graphs (Turan's theorem)

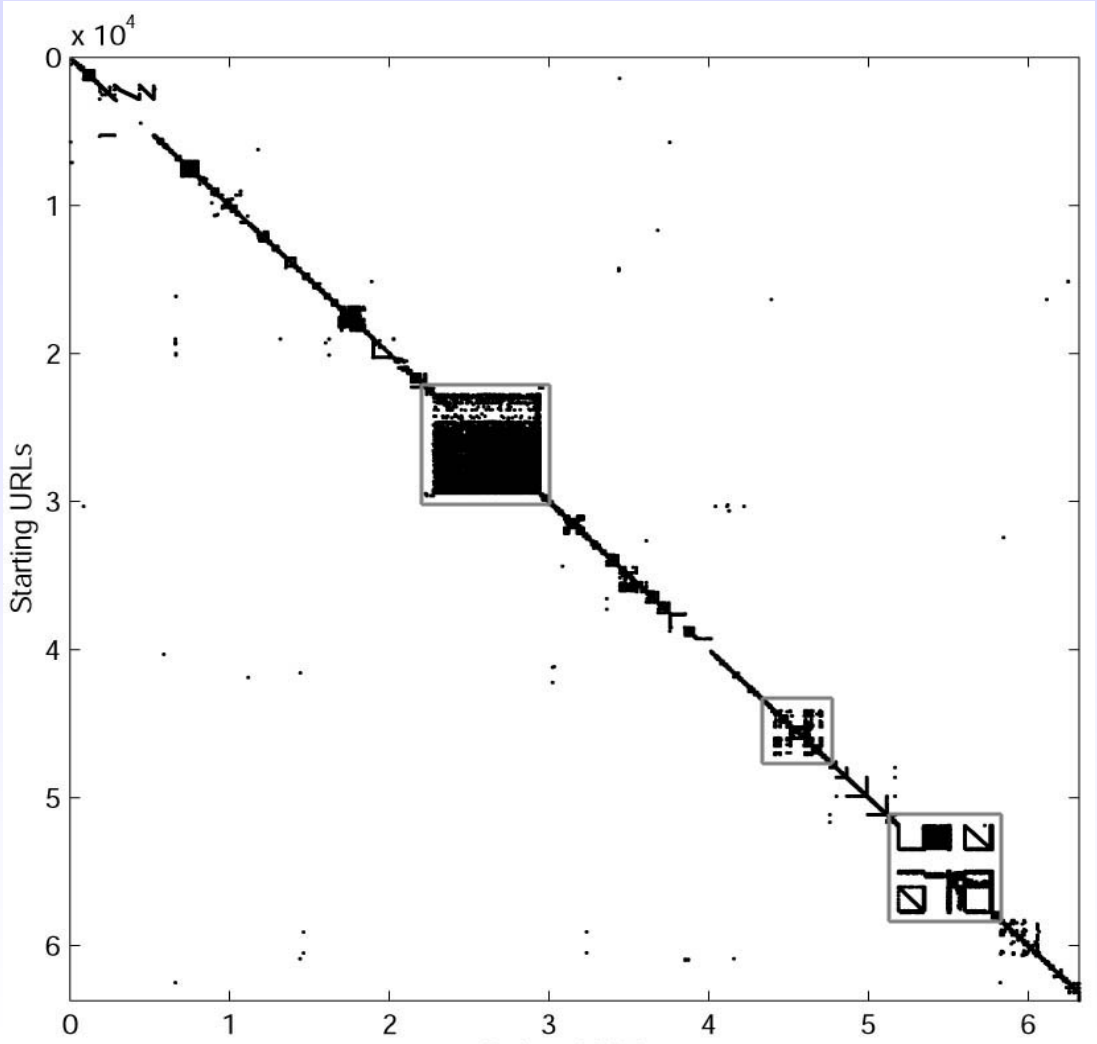
Show that if $G$ has no triangle then :
$|E(G)| \leq \frac{|V(G)|^2}{4}$

## Planar graphs

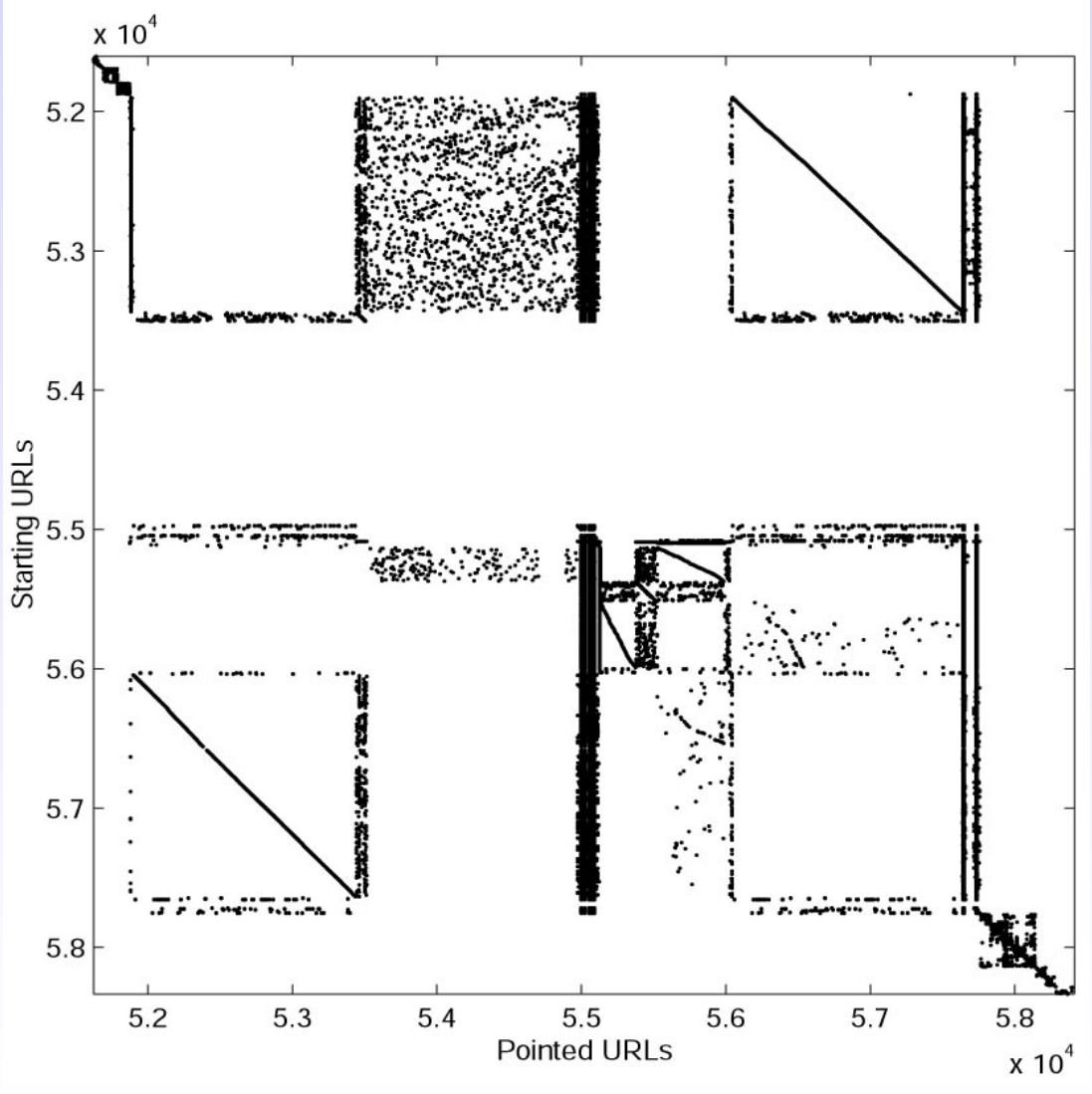
Show that if $G$ is a simple planar graph (i.e. without loop and parallel edge) then :
$|E(G)| \leq 3|V(G)| - 6$

Sparse graphs satisfy : $|E(G)| \in O(|V(G)|)$
Planar graphs are sparse, but also many graphs coming from applications are sparse.
For example the WEB graph is sparse.
For sparse graphs one has to use an adjacency lists representation.

# Matrice ordonnée par l'ordre alphabétique des noms des URL



# Un zoom autour de la diagonale



- Implicit hypothesis : the memory words have $k$ bits with $k > \lceil log(|V(G)|) \rceil$
- To be sure, consider the bit encoding level

- Adjacency lists
  $O(|V(G)| + |E(G)|)$ memory words
  Adjacency test : xy is an arc in $O(|N(x)|)$
- Basic one :
  The number of vertices and a list of edges. This often the format in which you cand find graphs in graph databases.
- Customized representations, a pointer for each arc . . .
- All these representations are linearly equivalent.

# Représentations analogiques

- Avec des anneaux et des ficelles
- Avec des morceaux d'ADN
- Pour les problèmes de flots qui vérifient les lois de Kirchoff, on peut modéliser le graphe par un réseau électrique et analyser comment le courant électrique y circule.
- Ou encore avec des pointeurs

## Adjacency Matrix

Adjacency Matrix
$O(|V(G)|^2)$ memory words (can be compressed)
Adjacency test : xy is an arc in $O(1)$

## Comment

This representation is not linearly equivalent to the previous one (adjacency lists)

# Exercice

Comment allier les avantages des deux représentations ?
liste d'adjacence : construction en $O(n+m)$
Matrices d'adjacence : accès à un arc en $O(1)$

Espace $O(n^2)$ mais en gardant la possibilité d'écrire des alogrithmes linéaires ?

# Quadratic space in linear time

- Select a 2-dimensional array GRAF of size $n^2$
  construct an auxillary unidimensional array of size m EDGE :
  For j=1 to m
  $xy$ being the $j^{th}$ edge of G
  $GRAF[x, y] = j$
  $EDGE[j] =$ a pointer to the memory word $GRAF[x, y]$
- The construction of the EDGE array requires $O(m)$ time
- Memory used $n^2 + m \in O(n^2)$

- $xy \in E$ iff $EDGE[GRAF[x, y]]$ contains a pointer pointing to the memory word $GRAF[x, y]$
- Therefore the query : $xy \in E$ ?
  Can be done in 2 tests $O(1)$.

For some large graphs, the Adjacency matrix, is not easy to obtain and manipulate.
But the neighbourhood of a given vertex can be obtained. (WEB Graph or graphs is Game Theory)

# Where can we find graphs ?

- Stanford Large Network Dataset Collection
- `https://snap.stanford.edu/data/`
- A universal data exchange format :
  n, m followed by an ordered list of edges

# Quicksands

- A sentence like :
  "To compute this invariant or this property of a given graph $G$ one needs to "see"( or visit) every edge at least once".
- False statement as for example the computation of twins resp. connected components on $\overline{G}$ knowing $G$.

# Exercise

Can the advantages of the 2 previous representations can be mixed in a unique new one ?
Adjacency lists : construction in $O(n+m)$
Incidence matrix : cost of the query : $xy \in E(G)$? in $O(1)$

## In other words

Using $O(n^2)$ space, but with linear time algorithms on graphs ?

# Auto-complemented representations

## Initial Matrix

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

## Tagged Matrix

| | $\overline{1}$ | 2 | $\overline{3}$ | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 1 | 0 |

- At most $2n$ tags (bits).
  $O(n + m')$ with $m' << m$.
  Dalhaus, Gustedt, McConnell 2000
- What can be computed using such representations ?

- Find a minimum sized representation
- If $G$ is undirected a minimum is unique.

# What is an elementary operation for a graph ?

- Traversing an edge or Visiting the neighbourhood ?
- It explains the very few lower bounds known for graph algorithms on a RAM Machine.
- Our graph algorithms must accept any auto-complemented representation.
  Partition Refinement

# Exercise

Suppose that a graph $G$ is given by its adjacency lists and let $\sigma$ be some total ordering of its vertices.

- How can we sort the adjacency with respect to $\sigma$ (increasing) ?
- What is the complexity of this operation ?
- Let $\sigma$ be the total ordering of the vertices with decreasing degrees, how to compute $\sigma$ ?

# Sorting an adjacency list

Suppose that a graph $G$ is given by its adjacency lists $A$ and let $\sigma$ be some total ordering of its vertices. How can we sort the adjacency with respect to $\sigma$ (increasing) ?

## One solution

Build another adjacency list structure $B$ from the old one by taking the vertices from $\sigma(n)$ down to $\sigma(1)$ in the following way :
read $A(\sigma(i))$ and add $\sigma(i)$ in front of the lists of $B$ corresponding to the neighbors of $\sigma(i)$
At the end of the process, the lists of $B$ are sorted in the right way.

# Complexity

Time : Linear time complexity (the size of the data structure $A$).
Memory : Twice the size of the adjacency lists $2|A|$

## Exercise

Adapt the solution for directed graphs

# Sorting the vertices by their degrees

1. Compute the degrees by scanning the adjacency lists in an array $D$
2. Use any per value sorting algorithm since all encoding numbers of vertices are bounded by $log_2(n)$ for a simple graph, to sort the vertices by decreasing degrees. This sorting algorithm is well known to be linear time.
3. Apply the previous algorithm to sort the adjacency lists.

# Remark : a second solution to sort the adjacency lists

- Just use any per value sorting algorithm to sort every adjacency lists.
- Can this be implemented linearly in the whole ?
- This method only uses $O(n)$ extra memory (the previous method uses $O(n+m)$).

The previous solutions need that the data structure is available at once in the memory.
This will be implicit in the remaining of the course, as well as the RAM model.