Basic algorithmic tools for graphs and other discrete structures, MPRI 2017-2018

Michel Habib
habib@irif.fr
http://www.irif.fr/~habib

27 septembre 2017

Schedule

Graph representations

Partition refinement

Putting the things together : LBFS$^+$

- 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.

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)

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 ?

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)

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.

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.

Reduced matrices

A line (resp. row) is flipped if it has more than $\lceil \frac{n}{2} \rceil$ ones.
If we apply till the end such a rule we reach a matrix which has less than $\lceil \frac{n}{2} \rceil$ ones per line (resp. row).
The process necessarily terminates since at each step the total number of ones strictly decreases.

The previous algorithm is clearly in $O(n^2)$.
But experimentation show linearity in practice.
Question : could this computation being done by a linear number of flips in average ?

Hardware Applications

There exists array memories for which it is better to have less than $\lceil \frac{n}{2} \rceil$ ones per line (resp. row).

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

Sorting the adjacency lists

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

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

Sorting an adjacency list

Suppose that a graph G is given by its adjacency lists A and let σ be some total ordering of its vertices. How can we sort the adjacency with respect to σ (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 list.
- 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.

Algorithm 1: LBFS

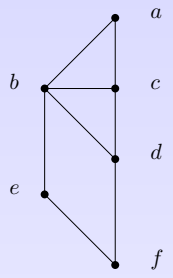
Input: undirected graph $G = (V(G), E(G))$

Output: an ordering σ of the vertices of G

foreach (*vertex* $u \in V(G)$) **do**
    $label(u) \leftarrow \epsilon$; %{where ϵ denotes the empty word }% ;
end
for ($i = 1$ **to** $|V(G)|$) **do**
    pick any unnumbered vertex u with lexicographically largest label ($\star$);
    $\sigma(i) \leftarrow u$; % {assign LBFS number i to vertex u}%;
    foreach (*unnumbered vertex* $v \in N(u)$) **do**
        append$(n - i)$ to label(v);
    end
end

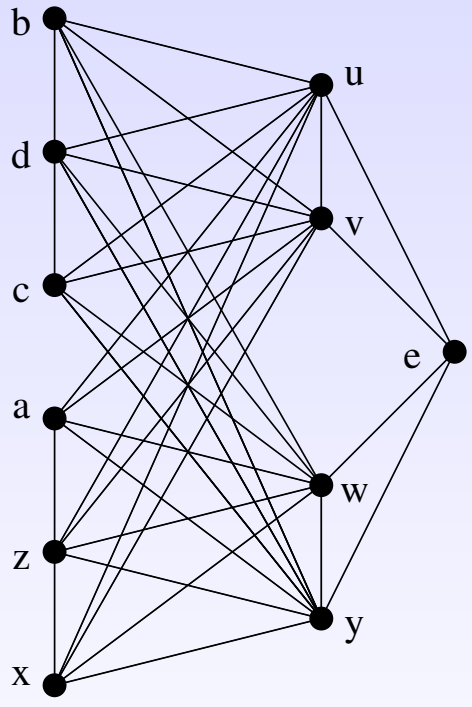
The LBFS algorithm assigns label to vertices. The labels are words over the alphabet $\{0, \ldots, n-1\}$. By convention ϵ denotes the empty word. The operation append$(n - i)$ puts the letter $n - i$ at the end of the word.

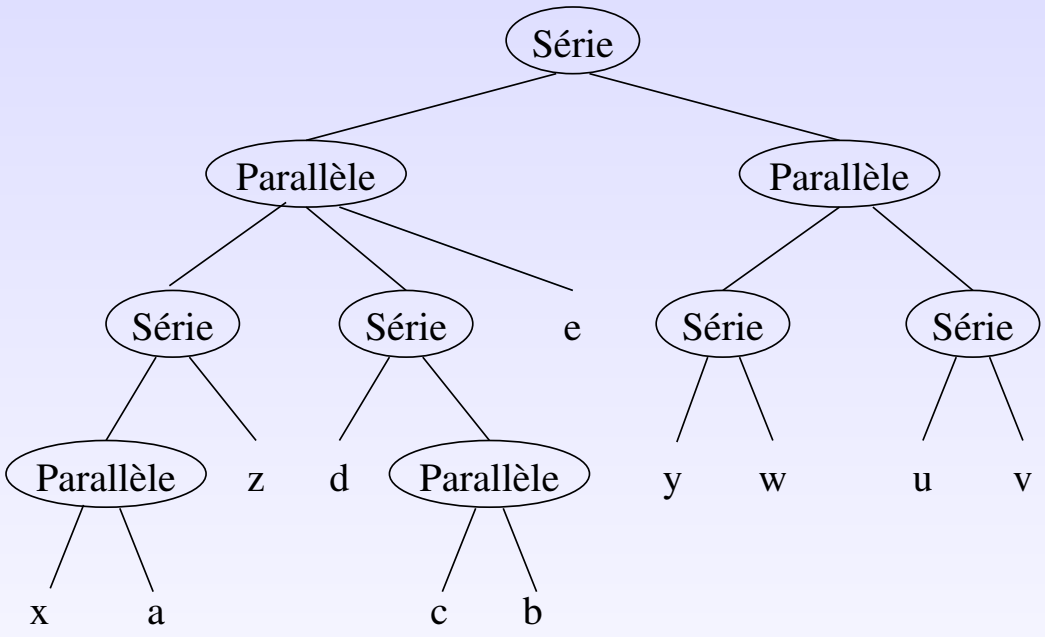
LBFS



vertex	i=1	i=2	i=3	i=4	i=5	i=6
d	ϵ					
c	ϵ	5				
b	ϵ	5	54			
f	ϵ	5	5	5		
a	ϵ	ϵ	4	43	43	
e	ϵ	ϵ	ϵ	3	32	32

Another example of LBFS





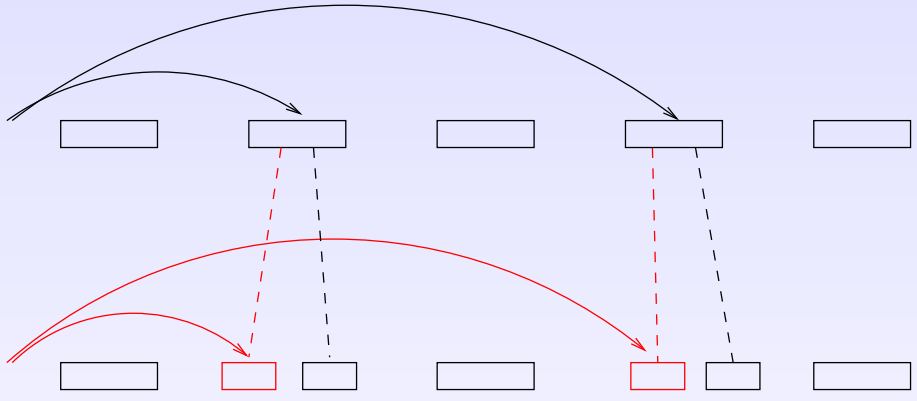
$\sigma(v)$	v	$N'(v)$	Partitions
			x d y u e v w c a z b
1	x	{y u v w z}	y u v w z | d e c a b
2	y	{d e w c a b z}	w z | u v | d e c a b
3	w	{d e c a z b}	z | u v | d e c a b
4	z	{a u v}	u v | a | d e c b
5	u	{d e v c a b}	v | a | d e c b
6	v	{d e c a b}	a | d e c b
7	a	{ }	d e c b
8	d	{b c}	c b | e
9	c	{b}	b | e
10	b	{ }	e
11	e	{ }	

TABLE: Step by step LexBFS of G. The resulting ordering is σ : **x y w z u v a d c b e**

Input : *A graph $G = (V, E)$ and an initial ordering τ of the vertices.*
Output : *An ordering σ of the vertices of G.*
1. $L \leftarrow (V)$; %{L, the list of parts, is initialized to the trivial partition (V)}
2. $i \leftarrow 1$; %{The counter for assigning vertex positions}
3. **while** $\exists P_i \neq \emptyset$ in $L = (P_1, \ldots, P_k)$ **do** %{Exists a non-empty part}
4. Let P_l be the leftmost non-empty part ;
5. Remove the first vertex x (smallest wrt. τ) from P_l; *the pivot* **(*)**
6. $\sigma(x) \leftarrow i$;
7. $i \leftarrow i + 1$;
8. **for** each partition $P_j, j \geq l$ **do**
9. Let $P' = \{v | v \in N(x) \cap P_j\}$; *P' are the vertices of P_j adjacent to x*
10. **if** P' is non-empty and $P' \neq P_j$ **then**
11. Remove P' from P_j ;
12. Insert P' to the left of P_j in L ; %{P' is a new part}
13. **end for**
14. **end while**
15. **return** (σ) ;

end LexBFS

LBFS



From a partition refinement perspective, this operation is quite easy since parts can be divided but there are not shuffled, the global ordering of the parts is maintained.
So for $LBFS$ (resp. $\overline{LBFS}$) the refining rules are simply :
For every part P_i insert $P_i \cap N(x)$ just before (resp. after) $P_i \setminus N(x)$. So the two splitted parts remain contiguous.

Definition

Partition Refinement

If $Q = \{C_1, \ldots C_k\}$ is a partition over a ground set X, for every $S \subseteq X$ we define from Q and S a new partition :
$Refine(Q, S) = \{C_1 \cap S, C_1 - S, \ldots, C_k \cap S, C_k - S\}$ [a]

a. empty sets are removed

More formally

$Refine(Q, S) = Q \wedge \{S, X\text{-}S\}$ in the partition lattice on X.

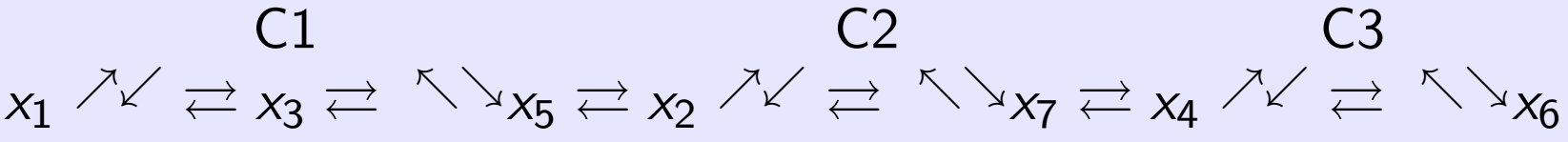
The method

The partition P is made up with a list of classes.
For each element x of the pivot set S, find the unique part C it belongs to.
Then move (or mark) x in C and tag C
At last, separate all marked parts C into $C \cap S$ and C-S.

Implementation

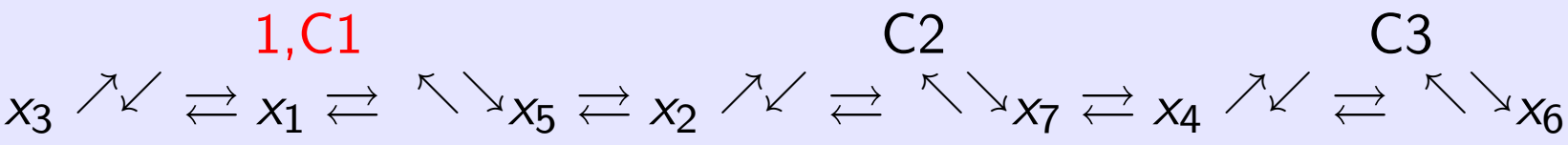
$V = \{x_1, \ldots, x_7\}$
$P = \{C1, C2, C3\}$ and $S = \{x_3, x_4, x_5\}$

Data Structure

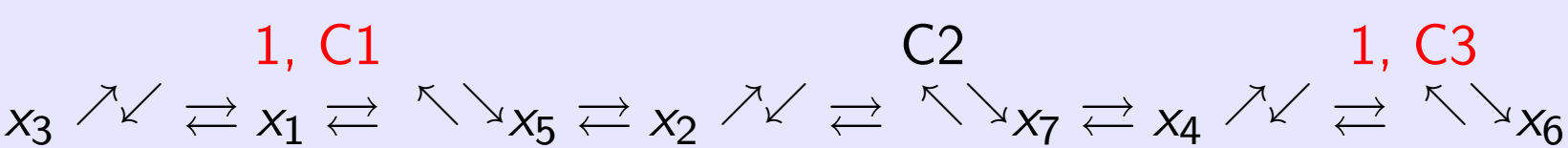


First Step

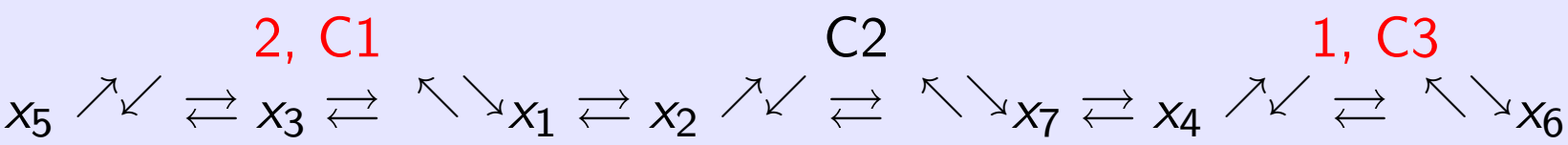
Processing x_3



Processing x_4



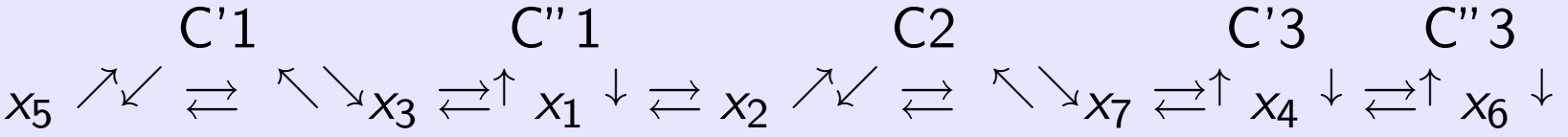
x_5



Second step

Maintain a list of the $Ci's$ that intersect S. List bounded by $|S|$.

Result



$Refine(P, S) = \{C'1, C''1, C2, C'3, C''3\}$
computed in $O(|S|)$

To implement this data structure we need at least :

- A doubled linked list L for the partition itself
- For each element of V, we need to maintain a reference to its position in the list
- For every element in L we need to maintain a reference to its part.
- Every part has to maintain a reference to its first element.

Of course it could be implemented using arrays instead of linked data structures. Furthermore this could be much more efficient in many programming languages (for example those in which list are badly implemented).
This technique is not only efficient theoretically (with respect to complexity measures) but also for practical purpose, since this technique can be implemented with a small overhead.

Exercises

1. Propose an alternative implementation using arrays.
2. Propose an implementation in an array stable (i.e. compatible with an initial ordering) and within the same complexity.

Refining a partition

Definition

Let $S \subseteq V$, and $P = \{X_1, \ldots, X_n\}$ be a partition of V.
$Q = \mathit{Refine}(S, P) = \{X_1 \cap S, X_1 - S, \ldots, X_n \cap S, X_n - S\}$
S is called a pivot.
NB Some sets can be empty and then ignored.

- $\mathit{Refine}(S, P) \leq P$
- $\mathit{Refine}(S, P) = P$ iff S is an union of parts of P

Duality

$\mathit{Refine}(S, P) = \mathit{Refine}(\overline{S}, P)$

Classes of twin vertices

Definition

x and y are called false twins, (resp. true twins) if $N(x) = N(y)$ (resp. $N(x) \cup \{x\} = N(y) \cup \{y\}$))

Exercise of the first lecture

Propose a good algorithm to compute these classes

Algorithm Folklore

Data: $G = (V, E)$ a graph with n vertices and m edges

Result: The classes of false twin vertices

$Q \leftarrow \{V\}$

for *Every* $x \in V$ **do**

    $Q \leftarrow \mathit{Refine}(Q, N(x))$

end

Proof

At the end, parts of Q have no splitter outside and therefore are modules.
Furthermore they have no splitter inside the part.
They are made up with false twins (non connected).

Complexity

$\Sigma_{x \in V} |N(x)| \in O(n + m)$

Other applications

Detection of multi-occurency in a list of subsets

Just construct the incidence bipartite elements–subsets and compute the twins.

Recognition of a laminar family

Laminar Family

A family $\mathcal{F}$ of subsets of a ground set X is laminar if :
$\forall F', F" \in \mathcal{F}$, either $F', F"$ are disjoint or included.
Such a family is ordered by inclusion with a forest structure.

Computing the tree structure

Sort the elements of $\mathcal{F}$ by decreasing size.
Compute using partition refinement the sets contained in F_0 ...
whole complexity in $O(\Sigma|F|_{F\in\mathcal{F}})$.

Degrees parts

Classification of the vertices in parts having the same degree.
A variation of the folklore algorithm for twins.

Generalized degree partition

Classification of the vertices in parts having the same degree with respects to the other parts. To compute this partition we can use a variation of the partition refinement.
DegreeRefine(P, S) :
computes the partition of S in parts having same degree with P
The computation of this partition is the first step of the main isomorphism algorithms.

Tree isomorphism using Partition refinement

Compute the generalized degree partitions of the two graphs G and H

Folklore Property

iF G and H are isomorphic then their partitions are identical.

Particular case of trees

For trees the converse is also true.

Graph search

Most of the classical graph searches can be implemented using partition refinement and sometimes this gives a good way to obtain an optimal implementation.

Another exercice

Data: A family $\mathcal{F}$ of subets of V

Result: Compute the overlap components of $\mathcal{F}$

Partition refinement a kind of technique dual to Union-Find.
Complementary uses :

- x et y belong to the same part $\rightarrow$ Union-Find
- x et y do not belong to the same part$\rightarrow$ Partition refinement.

Generic Refinement Algorithm

Input : P a partition and $\mathcal{S}$ a set of pivots
While $\mathcal{S} \neq \emptyset$
Choose $S \in \mathcal{S}$
$P = \mathit{Refine}(S, P)$
add all new generated parts to $\mathcal{S}$

Hopcroft's rule

In many applications when a part C is cut into 2 parts : C', C'' : it is enough to consider as a pivot in the following only C' or C''

Hopcroft's rule

Choose the smallest half

This assures an $O(nlogn)$ algorithm.

proof

The number of time an element can be used in a pivot set, is bounded by $logn$.

Variant

Avoid the biggest one

This also assures an $O(logn)$ factor in the complexity of the algorithm

Historical Notes

This technique is very powerfull not only for graph algorithms.
First used by Corneil for Isomorphism Algorithms 1970
Hopcroft Automata minimisation 1971
Cardon and Crochemore string sorting 1981
J. Spinrad Graph Partitioning (generic tool vertex spliting) 1986
Paigue, Tarjan 1987 (generic tool presented on three problems)
...

Some applications

- Quicksort : Hoare, 1962
- Minimal deterministic automaton Hopcroft $O(nlogn)$ 1971.
- Relational coarset partition Paige, Tarjan $O(nlogn)$ 1987
- Coarsest functional partition Paigue, Tarjan $O(nlogn)$ 1987 improved to $O(n)$ by Paigue, Tarjan, Bonic 1985 and Chrochemore 1982.
- String sorting $O(nlogn)$
- Doubly Lexicographic ordering Paige and Tarjan 1987 $O(LlogL)$, using a 2-dimensional refinement technique. where $L = \#$ones in the matrix using a 2-dimensional refinement technique.

Research Problems

- Find a linear-time algorithm which computes a doubly lexicographic ordering of a 0-1 matrix, i.e. an ordering of the columns and the lines for which lines and columns appear to be lexicographically ordered.
- Or show that such a linear algorithm does not exist.

Example of a doubly lexicographic ordering

	C1	C2	C3	C4	C5
L1	1	0	1	0	1
L2	0	1	0	1	1
L3	1	1	0	1	1
L4	0	0	0	0	1
L5	0	1	1	0	0

	C3	C1	C4	C5	C2
L4	0	0	0	1	0
L1	1	1	0	1	0
L5	1	0	0	0	1
L2	0	0	1	1	1
L3	0	1	1	1	1

- Such an ordering always exists
- Best algorithm to compute one :
  Paige and Tarjan [1987] proposed an $O(LlogL)$ where $L = n + m + e$ for a matrix with n lines, m columns and e non zero values, using partition refinement.
- For undirected graphs, such an ordering of the symmetric incidence matrix, yields an ordering of the vertices which has nice properties.

Many other applications on graphs

Partition refinement has many applications in graph algorithm design, mainly for undirected graphs. Kind of generic tool to obtain efficient algorithms easy to understand.
Vertex splitting, (also called vertex partitioning) when the neighborhood $N(x)$ is used as a pivot set. Provides a linear algorithm if the neighbourhood of every vertex is used a constant number of times.

- Interval graph recognition $O(n+m)$ using partition refinement on maximal cliques, 1-consecutiveness property $O(n+m)$, Habib, McConnell, Paul and Viennot 2000.
- Modular decomposition,
- Cograph recognition $O(n+m)$, Habib, Paul 2000.
- Transitive orientation

Duality for graphs applications

When the ground set is the set of all vertices and the pivots sets used in the algorithm are only neigbourhood, then the algorithm applies on G and $\overline{G}$ within the same complexity.

Consequence

The recognition of complement on chordal graphs can be done in linear time.

Proof

Just apply a LexBFS on G with the rule : S-P_i left-to $S \cap P_i$.
It will provides a LexBFS on $\overline{G}$.

In some application the order between parts matters and we play with ordered partitions.
Variations :

1. Parts are equipped with a counter representing its size.
2. Predicate "left-to" between parts in an ordered partition in $O(1)$.

Exercise :

Implement the classical graph search : BFS and DFS using partition refinement.

Research aspects

1. Find an efficient way to implement a backtrack operation (Kind of UnRefine)
2. Generalize the applications of partition refinement to directed graphs


J.E. Hopcroft, *A nlogn algorithm for minimizing states in a finite automaton*, Theory of Machine and Computations, (1971) 189-196.


A. Cardon and M. Crochemore, *Partitioning a Graph in* $O(|A|log|V|)$, Theor. Comput. Sci., 19 (1982) 85-98.


R. Paige and R. E. Tarjan, *Three Partition Refinement Algorithms*, SIAM J. Computing 16 : 973-989, 1987.


M. Habib, R. M. McConnell, C. Paul and L. Viennot, *Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing*, Theor. Comput. Sci. 234 :59-84, 2000.

Multisweep algorithms

In this talk a graph search is identified with the visiting ordering of the vertices it produces and therefore we can compose graph searches in a natural way.
Therefore we can denote by $M(G, x_0)$ the order of the vertices obtained by applying M on G starting from the vertex x_0.

Definition of the + Rule

Let M be a graph search and σ an ordering of the vertices of G, $M^+(G, \sigma)$ be the ordering of the vertices obtained by applying M on G starting from the vertex $\sigma(n)$ (last vertex of the previous search) and tie-breaking using σ in decreasing order.

Why this Rule ?

The $+$ Rule forces to keep the ordering of the previous sweep in case of tie-break

This $+$ rule was introduced for LBFS by Ma and Simon when dealing with interval graph recognition.

Algorithm 2: LBFS$^+$

Input: undirected graph $G = (V(G), E(G))$

Output: an ordering σ of the vertices of G

foreach (*vertex* $u \in V(G)$) **do**

    $label(u) \leftarrow \epsilon$; %{where ϵ denotes the empty word }% ;

end

for ($i = 1$ **to** $|V(G)|$) **do**

    $L \leftarrow$ { the set of lexicographically largest label unnumbered vertices };

    $u \leftarrow$ max of L with respect of σ;

    $\sigma(i) \leftarrow u$; % {assign LBFS number i to vertex u}%;

    foreach (*unnumbered vertex* $v \in N(u)$) **do**

        append$(n - i)$ to label(v);

    end

end

Putting the tools together : implementation of LBFS$^+$

An implementation of LBFS$^+(G, \tau)$ which is a special LBFS preserving in case of tie-break the τ^d ordering.
To this aim we need an implementation in $O(|S|)$ of Refine stable which preserves a given initial ordering of the vertices of the elements $x_i's$ of the parts.

1. Initial partition $< V >$ as a doubly linked list L ordered with τ^d
2. Reorder the adjacency lists of G following τ
3. Apply the usual LBFS on L (with the usual rule ; move to front, during the refinement process).
4. Invariant : During the LBFS the parts are sorted in τ^d ordering.