Techniques de base, MPRI 2016-2017

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

5 octobre 2016

Schedule

Implementations Issues

Partition refinement

Algorithm 1: LBFS

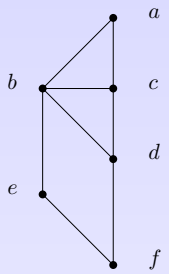
Input: undirected graph $G = (V(G), E(G))$

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

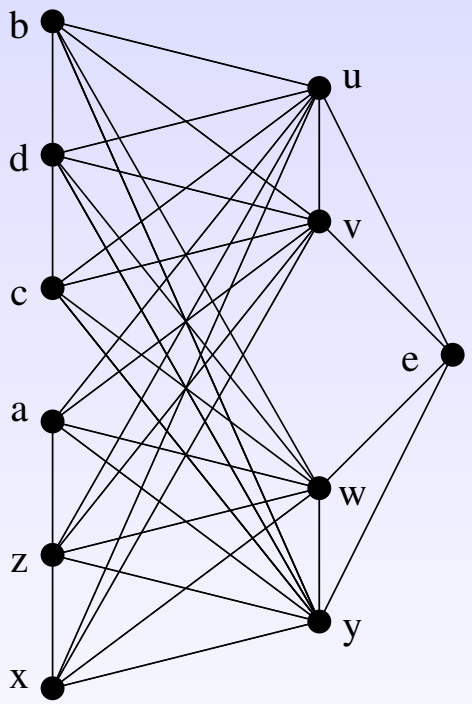
The LBFS algorithm assigns label to vertices. The labels are words over the alphabet $\{0, \ldots, n-1\}$. By convention ϵ denotes the empty word. The operation append($n - i$) puts the letter $n - i$ at the end of the word.

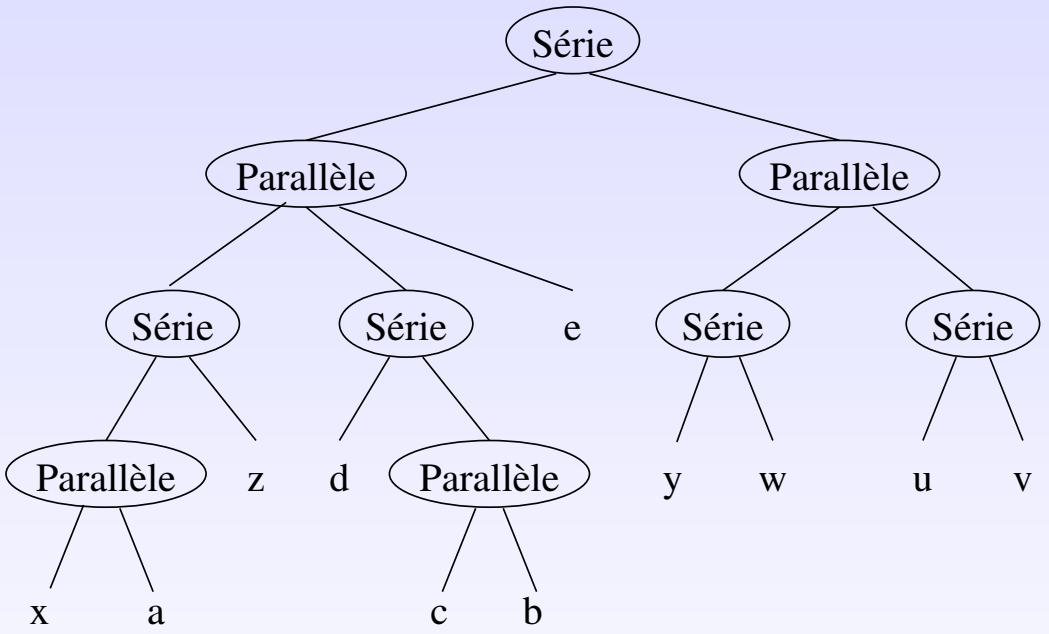
LBFS



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	{ }	

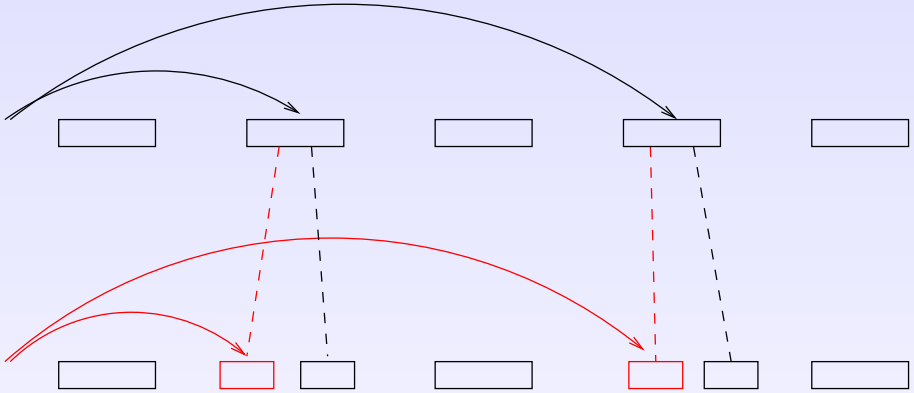
TABLE: Step by step LexBFS of G. The resulting ordering is σ : **x y w z u v a d c b e**

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) ;

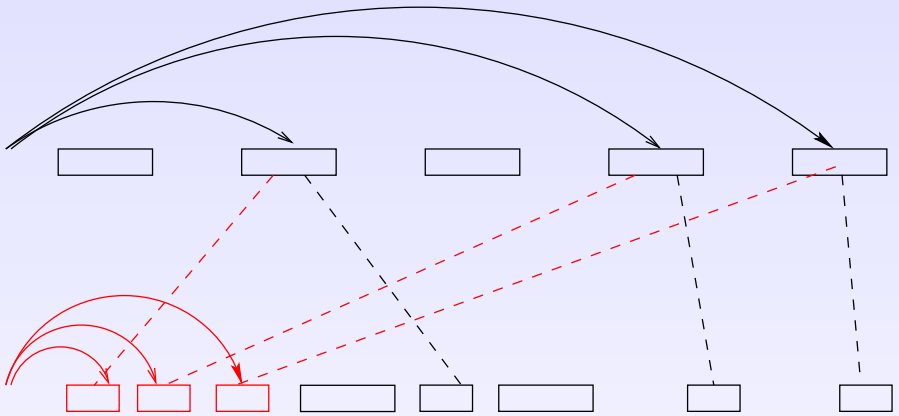
end LexBFS

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.

LDFS : harder to realize in linear time



We need to move to the front of the current ordered partition all the $P_i \cap N(x)$, keeping their respective ordering (as can be seen in Figure 2).
Let us call this operation **move to front**. By the way no linear time computation is known to handle this move to front operation, the best in $O(mloglogn)$ (using Van der Boas trees from J. Spinrad)

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

$$x_1 \nearrow\swarrow \rightleftarrows \overset{C1}{x_3} \rightleftarrows \nwarrow\searrow x_5 \rightleftarrows x_2 \nearrow\swarrow \rightleftarrows \overset{C2}{\nwarrow\searrow x_7} \rightleftarrows x_4 \nearrow\swarrow \rightleftarrows \overset{C3}{\nwarrow\searrow x_6}$$

First Step

Processing x_3

$$x_3 \nearrow\swarrow \rightleftarrows \overset{1,C1}{x_1} \rightleftarrows \nwarrow\searrow x_5 \rightleftarrows x_2 \nearrow\swarrow \rightleftarrows \overset{C2}{\nwarrow\searrow} x_7 \rightleftarrows x_4 \nearrow\swarrow \rightleftarrows \overset{C3}{\nwarrow\searrow} x_6$$

Processing x_4

$$x_3 \nearrow\swarrow \rightleftarrows \overset{1,\ C1}{x_1} \rightleftarrows \nwarrow\searrow x_5 \rightleftarrows x_2 \nearrow\swarrow \rightleftarrows \overset{C2}{\nwarrow\searrow} x_7 \rightleftarrows x_4 \nearrow\swarrow \rightleftarrows \overset{1,\ C3}{\nwarrow\searrow} x_6$$

x_5

$$x_5 \nearrow\swarrow \rightleftarrows \overset{2,\ C1}{x_3} \rightleftarrows \nwarrow\searrow x_1 \rightleftarrows x_2 \nearrow\swarrow \rightleftarrows \overset{C2}{\nwarrow\searrow} x_7 \rightleftarrows x_4 \nearrow\swarrow \rightleftarrows \overset{1,\ C3}{\nwarrow\searrow} x_6$$

Second step

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

Result

$$\begin{array}{ccccccccccccccccccccc}
 & & & \text{C'1} & & & & & \text{C''1} & & & & & & \text{C2} & & & & & \text{C'3} & & & & & \text{C''3} & \\
x_5 & \nearrow\swarrow & \rightleftarrows & & \nwarrow\searrow & x_3 & \rightleftarrows^{\uparrow} & x_1 & \downarrow & \rightleftarrows & x_2 & \nearrow\swarrow & \rightleftarrows & & \nwarrow\searrow & x_7 & \rightleftarrows^{\uparrow} & x_4 & \downarrow & \rightleftarrows^{\uparrow} & x_6 & \downarrow
\end{array}$$

$Refine(P, S) = \{C'1, C''1, C2, C'3, C''3\}$
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**
    $Q \leftarrow Refine(Q, N(x))$
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.

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.

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

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 $O(1)$.
3. Implement a backtrack

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.