On some recursive algorithms on graphs

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

Clermont-Ferrand, 26 janvier 2024

Plan

Standard linear recurrence equations

Applications to graphs

The *mlogn* cases

Decomposing a graph via a laminar trees

Application to modular decomposition

- $T(n) = a + T(n-1)$ donne $T(n) \in O(n)$.
  e.g. transfer a pending vertex in a tree or move forward one cell in an array

- $T(n) = a + T(n-1)$ donne $T(n) \in O(n)$.
  e.g. transfer a pending vertex in a tree or move forward one cell in an array
- It generalizes for any fixed k
  $T(n) = T(n-k) + a \cdot k$.

- $T(n) \leq a + 2T(\frac{n-1}{2})$ also gives $T(n) \in O(n)$.
  e.g. an algo that cuts a tree in 2 and removes a vertex.

- $T(n) \leq a + 2T(\frac{n-1}{2})$ also gives $T(n) \in O(n)$.
  e.g. an algo that cuts a tree in 2 and removes a vertex.
- $T(n) \leq an + T(\frac{b}{c} \cdot n)$
  if $b < c$ then $T(n) \in \Theta(n)$
  (My favourite algorithm of this kind is for the computation of the median of an array.)
  Otherwise $b > c$, $T(n) \in \Theta(n^{log_c(b)})$

Calculation of $K^{th}(TAB[1,n],i)$

Linear algorithm proposed by Blum, Floyd, Pratt, Rivest and Tarjan in 1972.
In fact, we're going to solve a more general problem, that of calculating the i^{th} element of an array of integers $K^{th}(TAB[1,n],i)$ which returns the element of TAB with the i^{th} value. To obtain the median, simply calculate $K^{th}(TAB[1,n],\lceil \frac{n}{2} \rceil)$

Calculation of $K^{th}(TAB[1, n], i)$

1. Divide TAB into $\lceil \frac{n}{5} \rceil$ packets of 5 integers (except possibly the last one, which contains the remainder of division by 5 of n).

1. As in Quicksort, taking x as the pivot. Then we know α and the sizes of sets A and B

Calculation of $K^{th}(TAB[1, n], i)$

1. Divide TAB into $\lceil \frac{n}{5} \rceil$ packets of 5 integers (except possibly the last one, which contains the remainder of division by 5 of n).
2. Calculate the median of each packet. Put them in a table *Rate* having $n' = \lceil \frac{n}{5} \rceil$ integers.

---

1. As in Quicksort, taking x as the pivot. Then we know α and the sizes of sets A and B

Calculation of $K^{th}(TAB[1,n],i)$

1. Divide TAB into $\lceil \frac{n}{5} \rceil$ packets of 5 integers (except possibly the last one, which contains the remainder of division by 5 of n).
2. Calculate the median of each packet. Put them in a table *Rate* having $n' = \lceil \frac{n}{5} \rceil$ integers.
3. $x \leftarrow K^{th}(Rate[1,n'], \lceil \frac{n'}{2} \rceil)$ */Computation of the median of medians */

1. As in Quicksort, taking x as the pivot. Then we know α and the sizes of sets A and B

Calculation of $K^{th}(TAB[1, n], i)$

1. Divide TAB into $\lceil \frac{n}{5} \rceil$ packets of 5 integers (except possibly the last one, which contains the remainder of division by 5 of n).
2. Calculate the median of each packet. Put them in a table *Rate* having $n' = \lceil \frac{n}{5} \rceil$ integers.
3. $x \leftarrow K^{th}(Rate[1, n'], \lceil \frac{n'}{2} \rceil)$ */Computation of the median of medians */
4. Partition TAB into $A[1, \alpha] < x \leq B[\alpha + 1, n]$ [1].

1. As in Quicksort, taking x as the pivot. Then we know α and the sizes of sets A and B

Calculation of $K^{th}(TAB[1, n], i)$

1. Divide TAB into $\lceil \frac{n}{5} \rceil$ packets of 5 integers (except possibly the last one, which contains the remainder of division by 5 of n).
2. Calculate the median of each packet. Put them in a table *Rate* having $n' = \lceil \frac{n}{5} \rceil$ integers.
3. $x \leftarrow K^{th}(Rate[1, n'], \lceil \frac{n'}{2} \rceil)$ */Computation of the median of medians */
4. Partition TAB into $A[1, \alpha] < x \leq B[\alpha + 1, n]$ [1].
5. If $i \leq \alpha$ then $K^{th}(A[1, \alpha], i]$
   Otherwise $K^{th}(B[\alpha + 1, n], i - \alpha]$

---

1. As in Quicksort, taking x as the pivot. Then we know α and the sizes of sets A and B

Example

Une itération des deux premières étapes de l'algorithme sur {0,1,2,3,...99}

	12	15	11	2	9	5	0	7	3	21	44	40	1	18	20	32	19	35	37	39
	13	16	14	8	10	26	6	33	4	27	49	46	52	25	51	34	43	56	72	79
Médianes	17	23	24	28	29	30	31	36	42	47	50	55	58	60	63	65	66	67	81	83
	22	45	38	53	61	41	62	82	54	48	59	57	71	78	64	80	70	76	85	87
	96	95	94	86	89	69	68	97	73	92	74	88	99	84	75	90	77	93	98	91

En rouge, la médiane des médianes.

Complexity Analysis

1. $O(n)$

Complexity Analysis

1. $O(n)$
2. In $O(1)$ for each package, so in total in $O(n)$.

Complexity Analysis

1. $O(n)$
2. In $O(1)$ for each package, so in total in $O(n)$.
3. $T(\lceil \frac{n}{5} \rceil)$

Complexity Analysis

1. $O(n)$
2. In $O(1)$ for each package, so in total in $O(n)$.
3. $T(\lceil \frac{n}{5} \rceil)$
4. $\leq T(\frac{7}{10}n)$.
   The median of the medians admits $1/2$ of the medians above it and each of them has at least 2 elements above it.
   So the median of the medians admits at least $1/2 \frac{3}{5} n = \frac{3}{10} n$ elements greater than or equal to it, and and therefore at most $\frac{7}{10} n$ elements smaller than it.
   Symmetrically, at least $\frac{3}{10} n$ elements are smaller than or equal to the median of the medians, and it has at most $\frac{7}{10} n$ elements which are superior to it. above it.
   So $|A[1, \alpha]| < \frac{7}{10} n$ and $|B[\alpha + 1, n]| < \frac{7}{10} n$.
   In all cases, the recursive call array will be smaller than $\frac{7}{10} n$.

- Therefore the inequations :

$$\begin{cases} T(1) = 1 \\ n \geq 2, T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(\frac{7}{10}n) + an \approx T(\frac{9}{10}n) + an \end{cases}$$

- Therefore the inequations :

$$\begin{cases} T(1) = 1 \\ n \geq 2, T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(\frac{7}{10}n) + an \approx T(\frac{9}{10}n) + an \end{cases}$$

- Which gives $T(n) \in O(n)$ and the previous algorithm is very efficient in practice.

- Therefore the inequations :

$$\begin{cases} T(1) = 1 \\ n \geq 2, T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(\frac{7}{10}n) + an \approx T(\frac{9}{10}n) + an \end{cases}$$

- Which gives $T(n) \in O(n)$ and the previous algorithm is very efficient in practice.
- **Remark :** Using packets of size 3 does not work and packets of size 7 or more are less efficient (since $\frac{6}{7} < \frac{9}{10}$).

General formula :

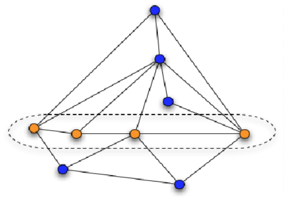
$$\begin{cases} T(1) = a \\ n \geq 2,\ T(n) = \sum_{i=1}^{i=k} a_i T(\frac{n}{b_i}) + an \end{cases}$$

If $\sum_{i=1}^{i=k} \frac{a_i}{b_i} < 1$ then $T(n) \in O(n)$.

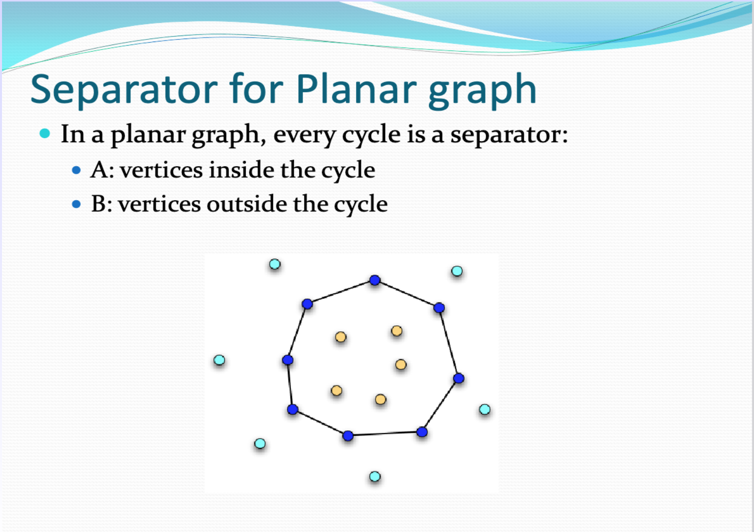
Planar

Separator

- The vertices of G are partitioned into three sets: A,B,C, such that no edge joins a vertex in A with a vertex in B, then C is a separator.
  - Useful for "divide-and-conquer" method.
  - usually requires C is small and A,B are at most αn (α is a constant less than 1)



Planar


Separator for Planar graph
In a planar graph, every cycle is a separator:
A: vertices inside the cycle
B: vertices outside the cycle

Planar separator theorem

Tarjan Lipton's planar separator Theorem

For a planar it is possible to find a simple cycle separator $C \in O(\sqrt{n})$, such that the inside and the outside of the cycle each have at most $|A|, |B| \geq \frac{2n}{3}$ vertices.

This separator cycle can be computed with a Breadth First Search (BFS) in linear time. Since planar graphs are hereditary one can easily derive recursive algorithms.
Furthermore all problems that can be solved using such theorem with the following inequality :

$$\begin{cases} T(1) = a \\ T(n) \leq T(\frac{2n}{3}) + an \end{cases}$$

$T(n) \in O(n)$

Can also be used for a divide an conquer approach
Example to compute a shortest cycle in $O(n^{3/2} logn)$

Many generalizations . . .

Theorem Folklore : path separator theorem

For every connected graph G on n vertices there exists a path P which partitions the vertices into L, P, R s.t.
$|L \cup P|, |P \cup R| \geq \frac{n}{3}$,
no edge between L and R

This domain is still very active

Recent improvement for planar graphs with small δ-hyperbolicity (2023).

A separator theorem for strongly connected digraphs (Bessy, Thomassé, Viennot STACS 2024) coming soon here.

And extensions to bounded treewidth graph . . .

What I like with these theorems is that they have proofs based on graph searches.
BFS for the planar separator theorem
DFS for the folklore undirected one
special DFS for strong digraph one

Between linear and quadratic

Except for sparse graphs such as planar graphs, for which
$m \in O(n)$,
we must evaluate the complexity in terms of the size of the graph, namely $n + m$.
For some problems, such as diameter computation, it is important to obtain algorithms non quadratic in m.

Without any separator theorem

Sparse expander graphs do not have any cycle separator theorem.
Sometimes we can only decompose a graph into k components $G_1, \ldots, G_k$ with no bounds on their sizes.
Applying it recursively it yields some inequality like :
$T(n+m) = a(n+m) + \Sigma T(n_i + m_i)$
If the decomposition and the merging operations can be done in linear time.
But this recursive equation does not provide linearity, could be quadratic.

Degrees parts

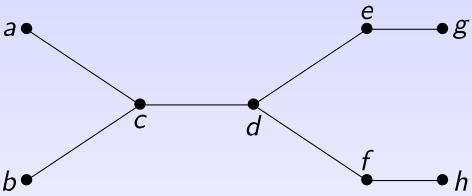
Classification of the vertices in parts having the same degree.
A variation of the folklore algorithm for twins.

Degrees parts

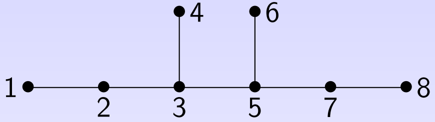
Classification of the vertices in parts having the same degree.
A variation of the folklore algorithm for twins.

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.



$$P(T) = \{\{c, d\}_3, \{e, f\}_2, \{a, b, g, h\}_1\}$$
$$P_{final}(T) = \{\{d\}_{3(3,2,2)}, \{c\}_{3(3,1,1)}, \{e, f\}_2, \{a, b\}_{1(3)}, \{g, h\}_{1(2)}\}$$



$$P(T') = \{\{3,5\}_3, \{2,7\}_2, \{1,4,6,8\}_1\}$$
$$P_{final}(T') = \{\{3,5\}_3, \{2,7\}_2, \{4,6\}_{1(3)}\{1,8\}_{1(2)}\}$$

- $P(T) = \{\{c, d\}_3, \{e, f\}_2, \{a, b, g, h\}_1\}$
  $P(T') = \{\{3, 5\}_3, \{2, 7\}_2, \{1, 4, 6, 8\}_1\}$
  These two degree partitions are isomorphic but T and T' are not isomorphic.

- $P(T) = \{\{c, d\}_3, \{e, f\}_2, \{a, b, g, h\}_1\}$
  $P(T') = \{\{3, 5\}_3, \{2, 7\}_2, \{1, 4, 6, 8\}_1\}$
  These two degree partitions are isomorphic but T and T' are not isomorphic.
- $P_{final}(T) =$
  $\{\{d\}_{3(3,2,2)}, \{c\}_{3(3,1,1)}, \{e, f\}_2, \{a, b\}_{1(3)}, \{g, h\}_{1(2)}\}$
  $P_{final}(T') = \{\{3, 5\}_3, \{2, 7\}_2, \{4, 6\}_{1(3)}\{1, 8\}_{1(2)}\}$
  But their two generalized degree partitions are not isomorphic.

Proposition

If G, G' are isomorphic graphs then also their generalized degree partitions are isomorphic.
Moreover : two trees T and T' are isomorphic iff their generalized degree partitions are isomorphic

Proposition

If G, G' are isomorphic graphs then also their generalized degree partitions are isomorphic.
Moreover : two trees T and T' are isomorphic iff their generalized degree partitions are isomorphic

Proof

Clearly if two graphs are isomorphic their generalized degree partitions must be isomorphic.
Let us consider the converse in the case of trees.
By induction on $|T| = |T'|$ and deleting one leaf in each tree denoted by x, x' (these leaves must belong to isomorphic parts of the generalized degree partitions).
The generalized degree partitions of $T \setminus x$ and $T' \setminus x'$ are still isomorphic but parts could be different due to some merging of parts from T and T'.

How to compute this generalized degree partition for a given graph G ?

- The first degree partition can be computed in $O(|V(G)| + |E(G)|)$

How to compute this generalized degree partition for a given graph G ?

- The first degree partition can be computed in $O(|V(G)| + |E(G)|)$
- Generalized degree partition can be computed using some partition refinement techniques.
  In $O(n + mlogn)$ using Hopcroft's rule : ignoring the largest new cell, after splitting a cell which ensures that an edge is at most consider *logn* times.
  A variant rule : Avoid the biggest part provides the same complexity.

This Generalized degree partition is the first step of every "good" isomorphism algorithm.
Such as Brendand McKay and Adolfo Piperno in Nauty or Traces `https://pallini.di.uniroma1.it/`, name it as the coarsest equitable partition.
Idem the Babai's subexponential algorithm in $O(2^{\sqrt{nlogn}})$ starts by computing such a partition.
In mathematics it is also known as an optimal fibration
see `http://Vigna.di.unimi.it/fibrations/` a well done web page by S. Vigna.

- It is also the core of the :
  Weisfeiler-Leman graph isomorphism test (1968).
  Very used in graph mining.
  A nice survey `https://arxiv.org/pdf/2201.07083.pdf`

- It is also the core of the :
  Weisfeiler-Leman graph isomorphism test (1968).
  Very used in graph mining.
  A nice survey `https://arxiv.org/pdf/2201.07083.pdf`
- A one hour lecture by Martin Grohe on the connexions with graph neural networks and descriptive logics.

- It is also the core of the :
  Weisfeiler-Leman graph isomorphism test (1968).
  Very used in graph mining.
  A nice survey `https://arxiv.org/pdf/2201.07083.pdf`
- A one hour lecture by Martin Grohe on the connexions with graph neural networks and descriptive logics.
- For many classes of graphs the Weisfeiler-Leman graph isomorphism test works, such as graphs with bounded rankwidth or twinwidth . . .

Bad news for graph isomorphism

- If G is regular then $P(G) = \{V(G)\} = P_{final}(G)$ and there exists non isomorphic regular graphs. As for example G_1 is the disjoint union of 2 triangles and G_2 is a cycle of length 6. Both are degree 2 regular and non isomorphic.

Bad news for graph isomorphism

- If G is regular then $P(G) = \{V(G)\} = P_{final}(G)$ and there exists non isomorphic regular graphs. As for example G_1 is the disjoint union of 2 triangles and G_2 is a cycle of length 6. Both are degree 2 regular and non isomorphic.
- There exists a lower bound to compute the generalized degree partition.

Berkholz, Bonsma and Grohe theorem (2016)

They called it color refinement, and they computed a canonical color refinement.
They proved $\Omega((n+m)logn)$ lower bound roughly for algorithms based on partition refinement.

Theorem

For every integer $k \geq 2$, there exist a graph with $n \in O(2^k k)$ vertices and $m \in O(2^k k^2)$ edges on which any partition refinement algorithm to compute a canonical color refinement requires $\Omega((n+m)logn)$.

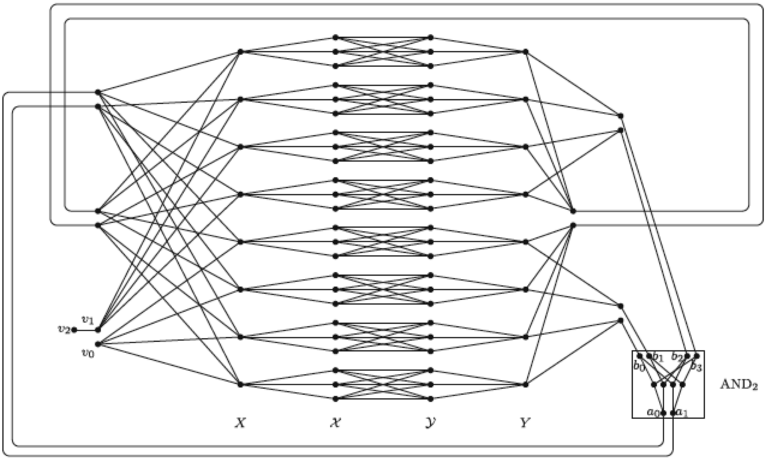
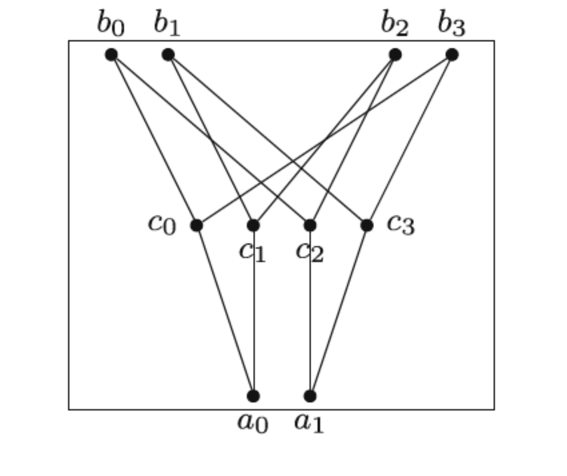


Fig. 1 The graph G_3


b_0 b_1 b_2 b_3
c_0 c_1 c_2 c_3
a_0 a_1

Some open questions

It seems that the proof can be applied to bisimilarity in transition systems.

But it does not work for the minimization of a deterministic automaton.

Can this be applied to the computation of doubly lexicographic ordering of a square *nxn* positive matrix for which the best algorithm is in $O(n + mlogn)$ where m is the non-zero entries of the matrix ?

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

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

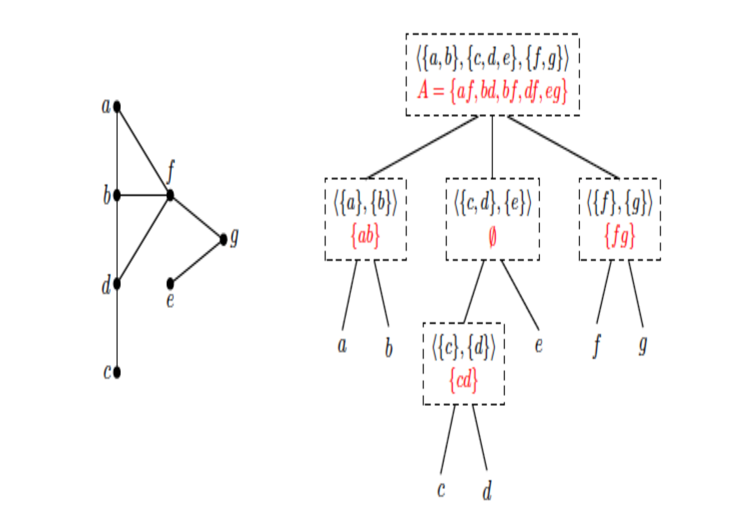
	C3	C1	C4	C5	C2
L4	0	0	0	1	0
L1	1	1	0	1	0
L5	1	0	0	0	1
L2	0	0	1	1	1
L3	0	1	1	1	1

- Such an ordering always exists

- 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.

- 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.

As defined in Schrivjer 2003 a *laminar* family $\mathcal{F}$ on a set V satisfies :
$\mathcal{F} \subseteq 2^V$ and for all $A, B \in \mathcal{F}$, either $A \subseteq B$ or $B \subseteq A$ or $A \cap B = \emptyset$.
Laminar families can be represented using a forest of rooted trees. A rooted tree $T_r = (T, r)$ is a pair composed of a tree T and a distinguished vertex r, call the root. A *leaf* of a rooted tree is a degree-one node. So the root may be a leaf. A node that is not a leaf is an *internal* node. **Moreover, we assume that every non-leaf node of a rooted tree has at least two children.**


a
b
f
g
d
e
c
⟨{a,b},{c,d,e},{f,g}⟩
A = {af, bd, bf, df, eg}
⟨{a},{b}⟩
{ab}
⟨{c,d},{e}⟩
∅
⟨{f},{g}⟩
{fg}
a
b
⟨{c},{d}⟩
{cd}
e
f
g
c
d

Procedure Compute(G,T)

Input: T a laminar-tree on V and its ordering τ_T and a preprocessed graph $G = (V, E)$

begin
  if $|V| \leq 2$ **then**
    Compute := A trivial value;
  end
  Let $u_1, \ldots u_k$ be the children of the root of T in increasing τ_T ordering;
  for $1 \leq i \leq k$ **do** $G_i := G(\mathcal{L}_{u_i}, E_i)$ and $T_i := T_{u_i}$;
  $Compute := Merge(Compute(G_1, T_1) \ldots, Compute(G_k, T_k))$
end

Theorem

The procedure Compute can be implemented in linear time, if the 2 following conditions (i) and (ii) are satisfied.
(i) The merge of 2 non connected subgraphs can be done in $O(1)$.
(ii) The merge of every connected subgraph partition P can be done in $O(|E_P|)$.

Proof

Using (i) we then notice that in the whole procedure at most $|V|$ merges of non connected subgraphs can be done in at most $O(|T|) = O(|V|)$ steps, since by definition of laminar-trees every node in T has at least 2 children.
Now we can consider the merging of connected subgraph partition and using (ii) we have the following recursive inequalities :
$T(n+m) \leq \Sigma_{1 \leq i \leq k} T(n_i + m_i) + a \cdot k + b \cdot |E_P|$, where
$n = \Sigma_{1 \leq i \leq k} n_i$ and $m = \Sigma_{1 \leq i \leq k} m_i + |E_P|$.
Since $k \leq |E_P| + 1$, then an easy induction gives
$T(n+m) \leq c \cdot (n+m)$ for every $c \leq 2 \cdot max\{a, b\}$.
And the procedure is therefore linear.

Modules

Modules

For a graph $G = (V, E)$, a **module** is a subet of vertices $A \subseteq V$ such that
$\forall x, y \in A,\ N(x) - A = N(y) - A$
The problem with this definition : must we check all subsets A ?

Modules

Modules

For a graph $G = (V, E)$, a **module** is a subet of vertices $A \subseteq V$ such that
$\forall x, y \in A,\ N(x) - A = N(y) - A$
The problem with this definition : must we check all subsets A ?

Trivial Modules

$\emptyset$, $\{x\}$ and V are modules.

Modules

Modules

For a graph $G = (V, E)$, a **module** is a subet of vertices $A \subseteq V$ such that
$\forall x, y \in A,\ N(x) - A = N(y) - A$
The problem with this definition : must we check all subsets A ?

Trivial Modules

$\emptyset$, $\{x\}$ and V are modules.

Prime Graphs

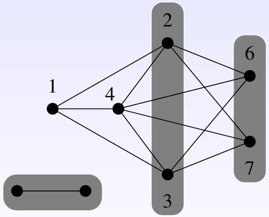
A graph is **prime** if it admits only trivial modules.

Examples

Characterization of Modules

A subset of vertices M of a graph $G = (V, E)$ is a **module** iff

$$\forall x \in V \setminus M, \text{ either } M \subseteq N(x) \text{ or } M \cap N(x) = \emptyset$$

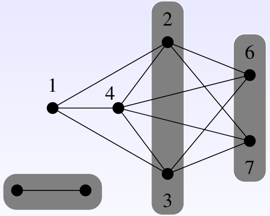


Examples of modules

Examples

Characterization of Modules

A subset of vertices M of a graph $G = (V, E)$ is a **module** iff
$$\forall x \in V \setminus M,\ \text{either } M \subseteq N(x) \text{ or } M \cap N(x) = \emptyset$$



Examples of modules

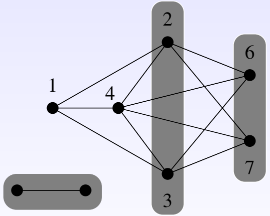
- connected components of G

Examples

Characterization of Modules

A subset of vertices M of a graph $G = (V, E)$ is a **module** iff

$$\forall x \in V \setminus M,\ \text{either } M \subseteq N(x) \text{ or } M \cap N(x) = \emptyset$$



Examples of modules

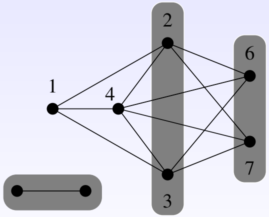
- connected components of G
- connected components of $\overline{G}$

Examples

Characterization of Modules

A subset of vertices M of a graph $G = (V, E)$ is a **module** iff

$$\forall x \in V \setminus M, \text{ either } M \subseteq N(x) \text{ or } M \cap N(x) = \emptyset$$



Examples of modules

- connected components of G
- connected components of $\overline{G}$
- any vertex subset of the complete graph (or the stable)

- Modules can be also defined for directed graphs but also for many discrete structures such as hypergraphs, matroids, boolean functions, submodular functions, automaton, ...

- Modules can be also defined for directed graphs but also for many discrete structures such as hypergraphs, matroids, boolean functions, submodular functions, automaton, . . .
- But also an operation on graphs : Modular composition
  a graph grammar with a simple rule : replace a vertex by a graph

- Modules can be also defined for directed graphs but also for many discrete structures such as hypergraphs, matroids, boolean functions, submodular functions, automaton, ...
- But also an operation on graphs : Modular composition
  a graph grammar with a simple rule : replace a vertex by a graph
- Very natural notion, (re)discovered under many names in various combinatorial structures
  such as : clan, homogeneous set, ...

- Modules can be also defined for directed graphs but also for many discrete structures such as hypergraphs, matroids, boolean functions, submodular functions, automaton, ...
- But also an operation on graphs : Modular composition
  a graph grammar with a simple rule : replace a vertex by a graph
- Very natural notion, (re)discovered under many names in various combinatorial structures
  such as : clan, homogeneous set, ...
- An important tool in graph theory, there exists a modular width (which is just the maximal size of a prime node in the modular decomposition tree).

Modular decomposition tree

Strong modules

A **strong module** is a module that does not strictly overlap any other module.

Modular decomposition tree

Strong modules

A **strong module** is a module that does not strictly overlap any other module.

Tree

A recursive a this theorem yields a tree T in which :

Modular decomposition tree

Strong modules

A **strong module** is a module that does not strictly overlap any other module.

Tree

A recursive a this theorem yields a tree T in which :

- The root corresponds to V

Modular decomposition tree

Strong modules

A **strong module** is a module that does not strictly overlap any other module.

Tree

A recursive a this theorem yields a tree T in which :

- The root corresponds to V
- Leaves are associated to vertices

Modular decomposition tree

Strong modules

A **strong module** is a module that does not strictly overlap any other module.

Tree

A recursive a this theorem yields a tree T in which :

- The root corresponds to V
- Leaves are associated to vertices
- Each node corresponds to a strong module

Modular decomposition tree

Strong modules

A **strong module** is a module that does not strictly overlap any other module.

Tree

A recursive a this theorem yields a tree T in which :

- The root corresponds to V
- Leaves are associated to vertices
- Each node corresponds to a strong module

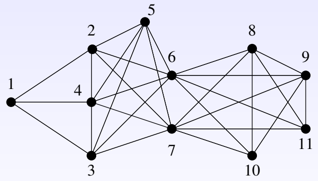
There are 3 types of nodes :

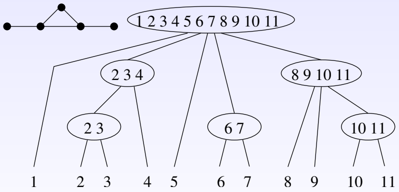
Parallel, Series and Prime

Laminar-tree

The modular decomposition tree of a graph G is particular Laminar-tree on $V(G)$

The set of strong modules is nested into an inclusion tree (called the **modular decomposition tree** $MD(G)$ of G).





- A preprocessing step : a graph search in fact a LexBFS, that produces an ordering τ of the vertices. Sort the adjacency lists with τ.

- A preprocessing step : a graph search in fact a LexBFS, that produces an ordering τ of the vertices. Sort the adjacency lists with τ.
- Partition the graph G into $G_1, \ldots G_k$ and apply the algorithm recursively on the $G_i's$.

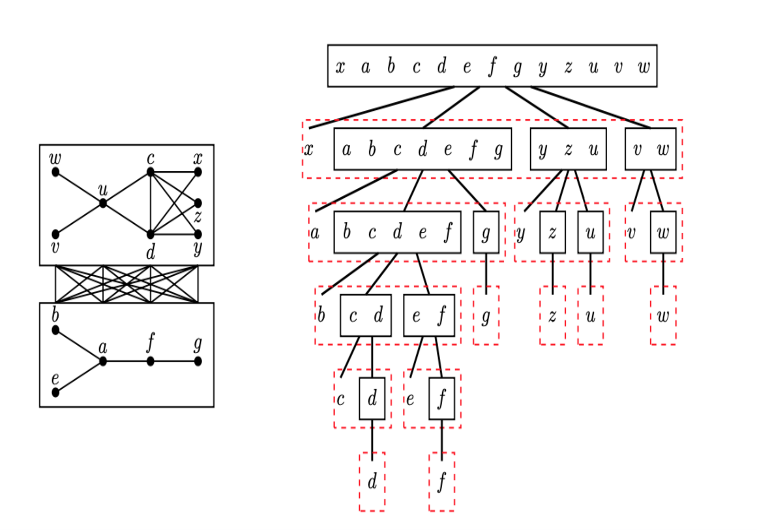
LexBFS

Algorithm 1: Lexicographic Breadth First Search (LexBFS) [?]

Input: A graph $G = (V, E)$.
Output: A LexBFS sequence $\vec{\sigma}$ on the vertices of V.

begin
  every vertex x is assigned the empty label $\ell(x) \leftarrow \langle \varepsilon \rangle$;
  let $\vec{\sigma} \leftarrow \langle \varepsilon \rangle$ be the empty sequence, $U \leftarrow V$ and $i \leftarrow n$;
  while $U \neq \emptyset$ **do**
    let $x \in U$ be such that $\ell(x)$ is lexicographically largest among all labels of vertices of U;
    $S \leftarrow S \setminus \{x\}$;
    for *every vertex* $y \in U \cap N(x)$ **do** $\ell(y) \leftarrow \ell(y) \cdot \langle i \rangle$;
    $\vec{\sigma} \leftarrow \vec{\sigma} \cdot \langle x \rangle$ and $i \leftarrow i - 1$;
  end
end
return $\vec{\sigma}$;

LexBFS laminar-tree



Although LexBFS is a BFS, il explores and builds its laminar-tree in a DFS way ! ! !

Algo for modular decomposition

Procedure MD(G,T)

Input: T a Lexicographic laminar-tree on V and its ordering τ_T and a preprocessed graph $G = (V, E)$

begin
    if $|V| \leq 2$ **then**
        MD(G,T) := a simple tree;
    end
    Let $u_1, \dots u_k$ be the children of the root of T in increasing τ_T ordering;
    for $1 \leq i \leq k$ **do** $G_i := G(\mathcal{L}_{u_i}, E_i)$ and $T_i := T_{u_i}$;
    $MD(G,T) := Merge(MD(G_1, T_1) \dots, MD(G_k, T_k))$
end

A recursive step

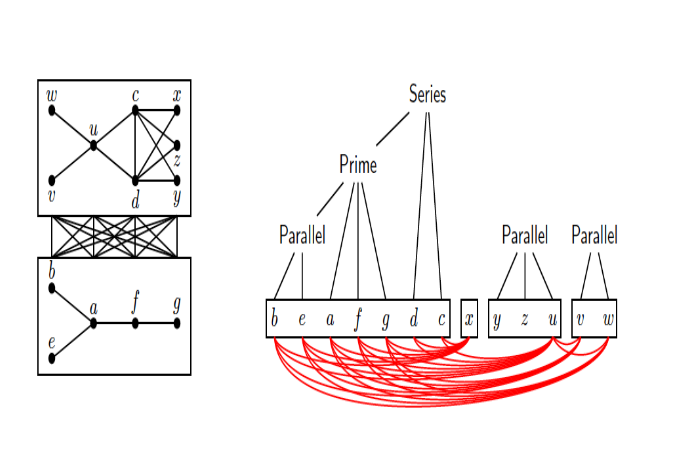


FIGURE: The idea is to merge the local MD trees in $O(|rededges|)$

- Preprocessing via a unique LexBFS.
  The G_i are the "slices" of the LexBFS. More precisely
  If the LexBFS starts at a vertex $x \in G$, $S_1 = N_G(x)$, S_i is the LexBFS tie-break set when the last vertex of S_{i-1} has been visited by LexBFS.
  (The definition also applies recursively on the $S_i's$).

- Preprocessing via a unique LexBFS.
  The G_i are the "slices" of the LexBFS. More precisely
  If the LexBFS starts at a vertex $x \in G$, $S_1 = N_G(x)$, S_i is the LexBFS tie-break set when the last vertex of S_{i-1} has been visited by LexBFS.
  (The definition also applies recursively on the $S_i's$).
- It is well-known that a LexBFS on G generates legitimate LexBFS's on the $G(S_i)'s$, and the slices are consecutive within the visiting ordering τ of the LexBFS. Furthermore vertices in some S_i have the same neighbourhood to the left.

- By definition all vertices in a slice S_i are connected the same way to the left.

- By definition all vertices in a slice S_i are connected the same way to the left.
- But for a slice S_i to be a module of G we have to check that : $\forall y \in S_j$, with $i < j$ y is connected to all vertices in S_i.

- By definition all vertices in a slice S_i are connected the same way to the left.
- But for a slice S_i to be a module of G we have to check that : $\forall y \in S_j$, with $i < j$ y is connected to all vertices in S_i.
- This will be done using the active edges.

Sketch of the complexity

- Let us consider a step of the algorithm
  $x, S_1, \ldots S_k$

Sketch of the complexity

- Let us consider a step of the algorithm $x, S_1, \ldots S_k$
- $|S_1| = |N(x)|$ therefore there exists $O(|MD(G(S_1)|)$ active edges for S_1

Sketch of the complexity

- Let us consider a step of the algorithm
  $x, S_1, \ldots S_k$
- $|S_1| = |N(x)|$ therefore there exists $O(|MD(G(S_1)|)$ active edges for S_1
- Between S_i and S_{i+1} :
  either there is no edge and G is not connected.
  or at least one vertex in S_i is connected to all vertices in S_{i+1}.
  Therefore the active edges are in $\Omega(|MD(G(S_{i+1})|)$

Sketch of the complexity

- Let us consider a step of the algorithm
  $x, S_1, \dots S_k$
- $|S_1| = |N(x)|$ therefore there exists $O(|MD(G(S_1)|)$ active edges for S_1
- Between S_i and S_{i+1} :
  either there is no edge and G is not connected.
  or at least one vertex in S_i is connected to all vertices in S_{i+1}.
  Therefore the active edges are in $\Omega(|MD(G(S_{i+1})|)$
- So for the merging operation we can use an algorithm linear in the size of the already computed modular decomposition trees.

- This merging of the decomposition trees is the technical part of the algorithm.

- This merging of the decomposition trees is the technical part of the algorithm.
- Two steps :

- This merging of the decomposition trees is the technical part of the algorithm.
- Two steps :
  1. Compute the modules of $G(S_i)$ for $1 \leq i < k$ (since S_k is a module of G) that are not modules of G

- This merging of the decomposition trees is the technical part of the algorithm.
- Two steps :
  1. Compute the modules of $G(S_i)$ for $1 \leq i < k$ (since S_k is a module of G) that are not modules of G
  2. Insert x when gluing the trees.

- This recursive generic idea was hidden in our ICALP 2008 paper (an obscure "Complexity issues" paragraph at the end of the 12 pages paper).

- This recursive generic idea was hidden in our ICALP 2008 paper (an obscure "Complexity issues" paragraph at the end of the 12 pages paper).
- The same scenario may be possible for transitive orientation a very closely related problem. Since with we wrote two sibling algorithms for modular decomposition and transitive orientation in $O((n+m)(logn))$.

- This recursive generic idea was hidden in our ICALP 2008 paper (an obscure "Complexity issues" paragraph at the end of the 12 pages paper).
- The same scenario may be possible for transitive orientation a very closely related problem. Since with we wrote two sibling algorithms for modular decomposition and transitive orientation in $O((n+m)(logn))$.
- But perhaps it could be useful for some other problems such as split decomposition , circular arc graph recognition, or even hypergraph modular decomposition starting with doubly lexicographic ordering (instead of LexBFS).

- This recursive generic idea was hidden in our ICALP 2008 paper (an obscure "Complexity issues" paragraph at the end of the 12 pages paper).
- The same scenario may be possible for transitive orientation a very closely related problem. Since with we wrote two sibling algorithms for modular decomposition and transitive orientation in $O((n+m)(logn))$.
- But perhaps it could be useful for some other problems such as split decomposition , circular arc graph recognition, or even hypergraph modular decomposition starting with doubly lexicographic ordering (instead of LexBFS).
- For Median graphs also very related to LexBFS.

As a conclusion

Three related problems :

- From the complexity viewpoint :
  Generalized degree partition $\geq$ minimal deterministic automaton $\geq$ modular decomposition

As a conclusion

Three related problems :

- From the complexity viewpoint :
  Generalized degree partition $\geq$ minimal deterministic automaton $\geq$ modular decomposition
- It deals with the computation of 3 equivalence relations on the vertices of a graph.

As a conclusion

Three related problems :

- From the complexity viewpoint :
  Generalized degree partition $\geq$ minimal deterministic automaton $\geq$ modular decomposition
- It deals with the computation of 3 equivalence relations on the vertices of a graph.
- But their complexities do not completely behave as my intuition.

As a conclusion

Three related problems :

- From the complexity viewpoint :
  Generalized degree partition $\geq$ minimal deterministic automaton $\geq$ modular decomposition
- It deals with the computation of 3 equivalence relations on the vertices of a graph.
- But their complexities do not completely behave as my intuition.
- In fact our modular decomposition algorithm uses partition refinement but not only.

As a conclusion

Three related problems :

- From the complexity viewpoint :
  Generalized degree partition $\geq$ minimal deterministic automaton $\geq$ modular decomposition
- It deals with the computation of 3 equivalence relations on the vertices of a graph.
- But their complexities do not completely behave as my intuition.
- In fact our modular decomposition algorithm uses partition refinement but not only.
- Discrete exact algorithmic is hard.

Many thanks for listening