A simple linear-time modular decomposition algorithm for graphs, using order extension⋆

Michel Habib, Fabien de Montgolfier, and Christophe Paul

CNRS - LIRMM, 161 rue Ada, 34392 Montpellier Cedex 5, France,
{habib,montgolfier,paul}@lirmm.fr

Abstract. The first polynomial time algorithm ($\mathcal{O}(n^4)$) for modular decomposition appeared in 1972 [8] and since then there have been incremental improvements, eventually resulting in linear time algorithms [22, 7, 23, 9]. Although an optimal time complexity these algorithms are quite complicated and difficult to implement. In this paper we present an easily implementable linear time algorithm for modular decomposition. This algorithm use the notion of factorizing permutation and a new data-structure, the Ordered Chain Partitions.

1 Introduction

The notion of *module* naturally arises from different combinatorial structures [26] and appears under the names of *autonomous sets, homogeneous sets, intervals, partitive sets, clans...* Modular decomposition is often the first algorithmic step for many graph problems including recognition, decision and optimization problems. Indeed, it plays an important role in various graphs recognition algorithms (eg. cographs [5], interval graphs [25], permutation graphs [29] and other classes of perfect graphs [12, 2]), and in the transitive orientation problem (see [11, 23]). Interested reader should refer to [26] for a survey on modular decomposition.

For a few years, linear-time algorithms have been known to exist ([22, 7, 23, 9]) but remain still rather complicated. Therefore in the late 90's, a series of authors attempts to design practical modular decomposition algorithms, even quasi-linear. In [24], an $O(n + m \log n)$ algorithm was proposed while [16, 9] got an $O(n+m.\alpha(n, m))$ complexity bound (where $\alpha(n, m)$ is the reverse Ackermann function). Such phenomena in the algorithmic progresses for a given problem is quite common. For example the first linear time algorithm for the interval graph recognition problem appears in 1976 [1]. This algorithm uses sophisticated data-structures, namely the PQ-trees. Since then, successive simplifications has been proposed [20, 6, 14]. One can also refer to planarity. The first linear time planarity testing algorithms that appear in the early 70's [19, 1] are rather complicated. Simpler algorithms have been later designed. Designing optimal but simple algorithms is a great algorithmic challenge. It was still an open problem to design a very simple linear time modular decomposition algorithm. We propose one (depicted in Figure 7) in this paper.

⋆ For a full version of this extended abstract, see [15]

Any graph $G = (V, E)$ considered here will be simple and undirected, with $n = |V|$ vertices and $m = |E|$ edges. The complement of a graph G is denoted by $\overline{G}$. If X is a subset of vertices, then $G[X]$ is the subgraph of G induced by X. Let x be an arbitrary vertex, then $N(x)$ and $\overline{N}(x)$ stand respectively for the neighborhood of x and its non-neighborhood. A vertex x *separates* (or distinguishes) two vertices u and v iff $(x, u) \in E$ and $(x, v) \notin E$. A *module* M of a graph G is a set of vertices that is not separated by any vertex:

$$\forall x \notin M,\ \forall y, z \in M,\ (x, y) \in E \iff (x, z) \in E.$$

The modules of a graph are a potentially exponential-sized family. However, the sub-family of *strong* modules, the modules that overlap[1] no other module, has size $O(n)$. The inclusion order of this family defines the *modular tree decomposition*, which is enough to store the module family of a graph [26]. The root of this tree is the trivial module V and its n leaves are the trivial modules $\{x\}, x \in V$.

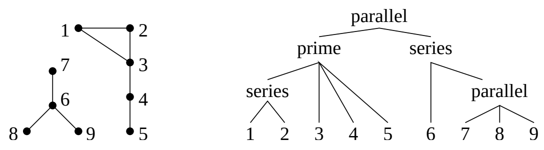


Fig. 1. A graph and its modular tree decomposition. The set $\{1, 2\}$ is a strong module. The module $\{7, 8\}$ is weak: it is overlapped by the module $\{8, 9\}$. The permutation $\sigma = (1, 2, 3, 4, 5, 6, 7, 8, 9)$ is a modular factorizing permutation.

It is well-known that any graph G with at least three vertices either is not connected (G is obtained from a *parallel composition* of its connected components); either its complement $\overline{G}$ is not connected (G is obtained from a *series composition* of the connect components of $\overline{G}$); or G and $\overline{G}$ are both connected. In the last case, the maximal (with respect to inclusion) modules define a partition of the vertex-set and are said to be a *prime* composition. It follows that the modular decomposition tree can be recursively built by top-down approach: at each step, the algorithm recurses on graphs induced by the maximal strong modules. Such a technique gives an $O(n^4)$ algorithm in [8], the first polynomial-time algorithm of a list that counts dozens of them[2].

The idea of *modular factorizing permutation* has been introduced in [3]: a permutation σ of the vertices of the graph such that, for each strong module, the reverse image $\sigma^1(M)$ is an interval of $\mathbb{N}$. It is clear that a DFS on the modular tree decomposition orders the leaves as a modular factorizing permutation. Conversely, [4] proposed a simple and linear-time algorithm that, given a graph and one of its factorizing permutations, computes the modular decomposition

[1] A overlaps B if $A \cap B \neq \emptyset$, $A \setminus B \neq \emptyset$ and $B \setminus A \neq \emptyset$

[2] See [27] or `http://www.lirmm.fr/~montgolfier/HistDM.html`

tree. This algorithm first computes a bracketing of the factorizing permutation, where each couple of parentheses encloses the separators of two consecutive vertices. This bracketing defines a tree, and with a few node operations, one can produce the modular decomposition tree (in other words, a factorizing permutation can be seen as the compression of the modular tree decomposition into a linear structure). It follows that the modular decomposition problem reduces to the computation of a modular factorizing permutation. In some cases such a permutation is given for free. In the case of chordal graphs, any Cardinality Lexicographic BFS yields a modular factorizing permutation [20]. [10, 13] used a similar notion to decompose an inheritance graph into blocks or modules. Recently, it has been shown for some intersection graphs families (namely interval graphs and permutation graphs), whose intersection model requires $O(n)$ space, that a factorizing permutation can be easily retrieved from the model yielding an $O(n)$ algorithm that computes the modular tree decomposition (see [27] or [21] for similar results).

We propose here the first linear-time algorithm that computes a modular factorizing permutation without computing the underlying decomposition tree. The present algorithm, combined with the one of [4], is therefore a simple linear-time modular decomposition algorithm, in two steps (first the modular factorizing permutation, then the modular decomposition tree). Using new data-structure, the Ordered Chain Partitions, we reduce the complexity to linear time. Avoiding in a first step the computation of the decomposition tree provides a real simplification of the modular decomposition algorithm. Indeed, as in [16], easy vertex partitioning rules are used.

2 Module-factorizing orders

Let $G = (V, E)$ be a graph and let $\mathcal{O}$ be a partial order on V. For two comparable elements x and y where $x \prec_{\mathcal{O}} y$ we state x *precedes* y and y *follows* x. Two subsets A and B *cross* if $\exists a, a' \in A$ and $\exists b, b' \in B$ such that $a \prec_{\mathcal{O}} b$ and $a' \succ_{\mathcal{O}} b'$. A linear extension of a poset is a completion of the poset into a total order.

Definition 1. *A partial order $\mathcal{O}$ is a* Module-Factorizing Partial Order *(MFPO) of $V(G)$ if two non-intersecting strong modules of G do not cross.*

The *modular factorizing permutations* (hereafter factorizing permutation for short) are exactly the module-factorizing total orders.

Proposition 1. *A partial order $\mathcal{O}$ is an MFPO if and only if it can be completed into a factorizing permutation.*

Our algorithm starts with the *trivial partial order* (with no comparison between any pair of vertices), which is an MFPO. Then the order is extended with new comparisons between vertices. When the order is total, the process stops. Proving that the extensions preserve its module-factorizing property shows the correctness of the algorithm, namely the final order is a factorizing permutation.

3 Towards a linear time algorithm

In [16], an $O(n + m \log n)$ algorithm, based on partition refinement techniques [28], was proposed to compute a modular factorizing permutation. This algorithm uses a restricted class of MFPO: the *ordered partitions* [28]. They are easy to handle, using a simple implementation, where most operations can be performed in $O(1)$. This section describes the main techniques of [16], also used by our algorithm.

Definition 2. *An* ordered partition *is a collection* $\{\mathcal{P}_1, \dots \mathcal{P}_k\}$ *of pairwise disjoint* parts*, with* $V = \mathcal{P}_1 \uplus \dots \uplus \mathcal{P}_k$*, and an order* $\mathcal{O}$ *such that for all* $x \in \mathcal{P}_i$ *and* $y \in \mathcal{P}_j$*,* $x \prec_{\mathcal{O}} y$ *iff* $i < j$*.*

Algorithm of [16] starts with the trivial partition (a single part equal to the vertex set) and iteratively extends (or *refines*) it until every part is a singleton. A *center* vertex $c \in V$ is distinguished and two refining rules, preserving the MPFO property, are used. These rules are defined by Lemma 1.

Lemma 1. *[16]*

1. Center Rule: *For any vertex* c*, the ordered partition* $\overline{N}(c) \uplus \{c\} \uplus N(c)$ *is module-factorizing.*
2. Pivot Rule: *Let* $\mathcal{O} = \mathcal{P}_1 \uplus \dots \uplus \{c\} \uplus \dots \uplus \mathcal{P}_k$ *be an ordered partition with center* c *and let* $p \in \mathcal{P}_i$ *such that* $\mathcal{P}_j$*,* $i \neq j$*, overlaps* $N(p)$*. If* $\mathcal{O}$ *is an MFPO, then the following refinements preserve the module-factorizing property:*
   (a) if $\mathcal{P}_i \prec_{\mathcal{O}} \mathcal{P}_j \prec_{\mathcal{O}} \{c\}$ *or* $\{c\} \prec_{\mathcal{O}} \mathcal{P}_j \prec_{\mathcal{O}} \mathcal{P}_i$*, then replace* $\mathcal{P}_j$ *by* $(N(p) \cap \mathcal{P}_j) \uplus (\overline{N}(p) \cap \mathcal{P}_j)$ *(in that order),*
   (b) otherwise replace $\mathcal{P}_j$ *by* $(\overline{N}(p) \cap \mathcal{P}_j) \uplus (N(p) \cap \mathcal{P}_j)$ *(in that order),*

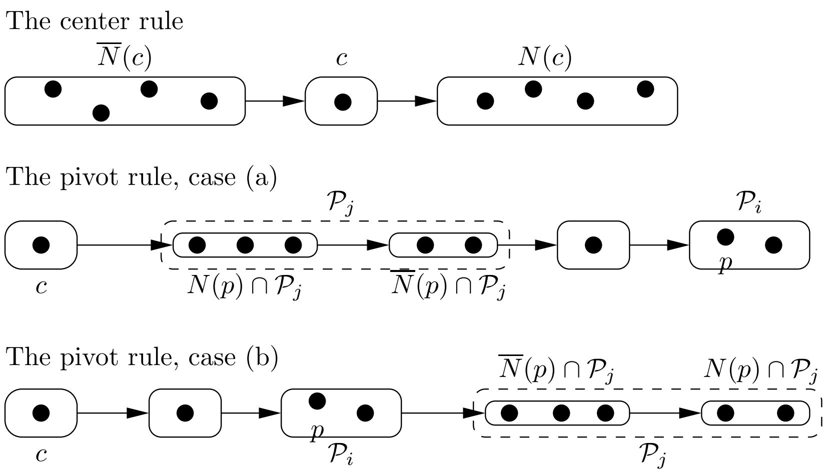


Fig. 2. The refinement rules defined in [16].

The *center rule* set a center and breaks a trivial partition to start the algorithm. Once launched, the process goes on based on the *pivot rule*, that splits each part $\mathcal{P}_j$ (saves the pivot part $\mathcal{P}_i$), according to the neighborhood of the pivot. When Algorithm of Figure 3 ends, every part is a module. To obtain a factorizing permutation, it has to be recursively relaunched on the non-singleton parts. The complexity issues depend on the choice of the part $\mathcal{P}$ (l.4 of Algorithm of Figure 3). Using *Hopcroft's rule* [18], [16] achieves an $O(m \log n)$ time-complexity.

```
Refine($G$,$\mathcal{O} = \{V\}$)
1. Pick a center $c$
2. Extend $\mathcal{O}$ using the center rule with $c$
3. While the partition can be extended further Do
4.     Select a part $\mathcal{P}$
5.     For each $p \in \mathcal{P}$ Do
6.         Extend $\mathcal{O}$ using the pivot rule with $p$
7. End of while
```

Fig. 3. Partition refinement scheme of [16].

4 Ordered chain partition and linear time algorithm

To improve the complexity down to linear-time, our algorithm uses each vertex a constant number of times as a pivot. This algorithm is depicted in Figure 7. An example of execution is presented in Appendix.

Definition 3. *An* ordered chain partition (OCP) *is a partial order such that each vertex belongs to one and only one* chain*, and one chain belongs to one and only one* part*. The vertex of the same chain are totally ordered, the chains of the same part are uncomparable, and the parts are totally ordered*

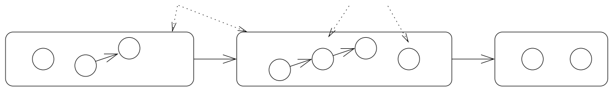


Fig. 4. An Ordered Chain Partition.

A *trivial* chain contains only one vertex, and a *monochain* part contains only one chain. The OCPs generalize the Ordered Partitions since in the latter ones contain only trivial chains. $\mathcal{C}(x)$ will denote the chain containing x while $\mathcal{P}(x)$ will denote the part of the partition containing x. Each chain $\mathcal{C}$ has a *representative vertex* $r(\mathcal{C}) \in \mathcal{C}$. During the algorithm, the chains will behave as their representative vertex (the tests for the refinement rules are done on the representatives). Notice that chains are possibly merged. In that case, the representative of the new chain is one of the former representatives (indeed it will be the old center). But chains will never be split.

The algorithm still uses the *center* rule and the *pivot* rule (see Lemma 1). The chains are moved by these two rules, according to the adjacency between their representative vertex and the center or the pivot. There is a third rule, the *chaining* rule (line 9 of algorithm of figure 7). Unlike the two first ones, this third rule *removes* comparisons from the order. This rule first concatenates a sequence of monochain parts, that occur consecutively in $\mathcal{O}$, into one chain. Then this new chain is inserted into one of the two parts, say $\mathcal{P}$, neighboring the chain (see Figure 5). The comparisons between the chain and $\mathcal{P}$ are lost. But since the number of chains strictly decreases during the algorithm, the process is guaranteed to end. Finally the above invariant is satisfied.

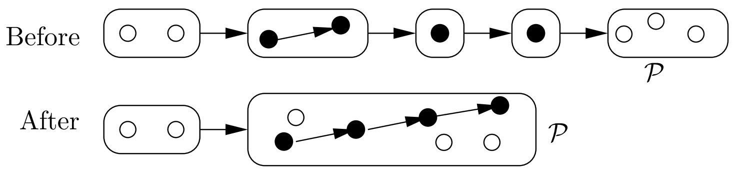


Fig. 5. The chaining rule, chaining the black vertices into $\mathcal{P}$.

Invariant 1 *The ordered chain partition $\mathcal{O}$ is an MFPO of $V(G)$ and no chain is overlapped by some strong module.*

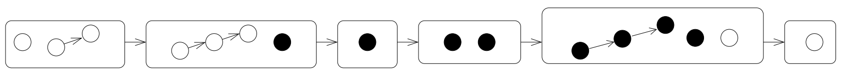


Fig. 6. Position of a strong module M (black vertices) in $\mathcal{O}$ (Invariant 1).

To use any vertex $\mathcal{O}(1)$ times as a pivot, the algorithm picks only *one* vertex per part to extend the OCP (instead of *all* the vertices of the part as [16] did). A chain is *used* if its representative vertex has been yet used as pivot. Similarly a part is used if it contains an used chain. Any used part already has a *pivot*, therefore no other vertex can be chosen as pivot. Unlike Algorithm of Figure 3, when all parts are used, the non-trivial (multichain) parts are *not* necessarily modules. The algorithm chooses a new center and recurses (see line 12 of Algorithm of Figure 7).

```
Refine($G$,$\mathcal{O}$,$[\![i,j]\!]$,$c$)                    /*$[\![i,j]\!]$ denotes the working factor of $\mathcal{O}$
 1.  Split $[\![i,j]\!]$ using the $c$ center rule           /* in which the pivots are chosen
 2.  While some multichain part in $[\![i,j]\!]$ exists Do
 3.      While there is an unused part in $[\![i,j]\!]$ Do
 4.          Select a unused part $\mathcal{P} \subset [\![i,j]\!]$ and a chain $\mathcal{C} \in \mathcal{P}$
 5.          Extend $\mathcal{O}$ using the pivot rule with $p = r(\mathcal{C})$
 6.      End of while
 7.      If some multichain part in $[\![i,j]\!]$ exists Then
 8.          Find the multichain part $\mathcal{P}'$ of the new center $c_n$
 9.          Create the new chain $\mathcal{S}$ containing $c$ and $c_n$ using the chaining rule
10.          Add $\mathcal{C}$ to $\mathcal{P}'$
11.          Extend $\mathcal{O}$ using the pivot rule with $c_n$
12.          Refine($G$,$\mathcal{O}$,$\mathcal{P}(c_n)$,$c_n$)
13.      End of if
14.  End of while
```

Fig. 7. Linear-time algorithm.

Choice of the new center (line 8 of Algorithm of Figure 7) As already seen in the algorithm of [16], the center plays an important role (see Lemma 1). The rule described below was already defined and proved in the context of cographs [17]. Indeed the following invariant is the basis of the correctness proof:

Invariant 2 *Let M be a strong module and c be the center. Then either c belongs to M, or any parts intersecting M is monochain, or a part $\mathcal{P}$ such that $M \subseteq \mathcal{P}$ exists.*

The new center c_n must fulfill Invariant 2, as the old center c did. If all the strong modules containing c but not c_n are included in $\mathcal{P}(c)$, then Invariant 2 holds. Let $\mathcal{P}_L$ (resp. $\mathcal{P}_R$) be the rightmost (resp. leftmost) multichain part that precedes (resp. follows) c. As both parts are *used*, their pivots p_L and p_R are defined. One of them is chosen for the recursive call, and its pivot becomes the new center. Only one pivot among p_L and p_R distinguishes the other from the center c. The rule chooses that pivot (wlog. say p_L see Figure 8.b) as new center. A simple adjacency test between p_L and p_R is enough to implement that choice. Assume the other choice is made: ie. p_L distinguishes c and p_R and p_R has been chosen as the new center. It could exist a module containing c and p_L but not p_R: such a module would violate Invariant 2, a contradiction.

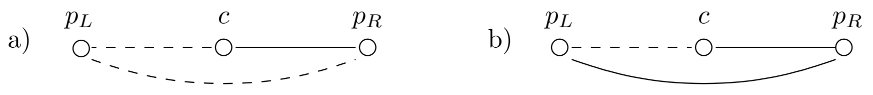


Fig. 8. In case a), the new center is p_R, in case b) p_L is chosen.

Center and pivot rules modified The pivot rule works as described in Lemma 1 (rule 2). The center rule should be handled carefully. It breaks $\mathcal{P}(c)$, the part containing the new center c, into three parts (see Lemma 1, rule 1). The case where $\mathcal{P}(c) \cap N(c)$ or $\mathcal{P}(c) \cap \overline{N}(c)$ is empty, could hinder the algorithm since the number of parts does not increase (a similar problem was observed in [17] for the cograph recognition). This case occurs when c and the previous center c_p are both adjacent (respectively nonadjacent) to the other representatives of $\mathcal{P}(c)$. It can be shown that $\mathcal{P}(c)$ has at least three chains. Therefore if $\mathcal{C}(c_p)$, the chain containing the old center, is put in that empty part, then cycling is avoided and the module-fatorizing property is still valid.

The chaining rule (line 9, Algorithm of Figure 7) When a new center c_n is chosen, there are only monochain parts between c_n and c. The chain to concatenate with $\mathcal{P}(c_n)$, using the chaining rule, starts from $\mathcal{P}(c_n)$, contains c and extends until a certain chain $\mathcal{C}(a)$ that is contained in a monochain part. Wlog. assume that $c_n \prec c$ and define (1) as the property that a part $\mathcal{P}$ in the working factor ($\mathcal{P} \subset [\![i,j]\!]$) fulfills if:

$$\mathcal{P} \succeq_{\mathcal{O}} \mathcal{P}(c) \text{ is a monochain part and } (Pivot[\mathcal{P}], c_n) \notin E \tag{1}$$

Let $\mathcal{P}'$ be the leftmost part (wrt. $\mathcal{O}$) that violates (1) and such that any part between $\mathcal{P}(c)$ and $\mathcal{P}'$ satisfies (1). Since $\mathcal{P}(c)$ fullfills (1), part $\mathcal{P}'$ exists. The chain $\mathcal{S}$ to concatenate with $\mathcal{P}(c_n)$ starts from the part that follows $\mathcal{P}(c_n)$ and extends until $\mathcal{P}'$ (excluded). Lemma 2, required in the proof of Theorem 1, ensures that the strong modules containing c but not c_n are included in $\mathcal{P}(c_n)$. Invariant 2 will be fulfilled, and the algorithm can be relaunched.

Lemma 2. *Let $\mathcal{P}_c$ be the part resulting from the concatenation of $\mathcal{P}(c_n)$ and $\mathcal{S}$ by the chaining rule. Every module containing c but not c_n is included in $\mathcal{P}_c$.*

Notice that the part $\mathcal{P}(c)$ now contains *two* used chains, $\mathcal{C}(c)$ and $\mathcal{C}(c_n)$. But the center rule, at next recursive call, will separate. Then the invariant property (usefull for proving time complexity) that every part contains at most one used chain, holds.

The problem of bad pivots, and the working factor In [16], it has been shown that the refinement rules (center and pivot rule, see Lemma 1) can be applied as long as any pivot that precedes the center c is non-adjacent with it, while a pivot that follows the center is one of its neighbors. In the new algorithm, the choice of a new pivot could hinder that property. A vertex $x \in \mathcal{P}$ is said to be *bad* if:

$$\text{either } x \prec_{\mathcal{O}} c \text{ and } x \in N(c); \text{ or } x \succ_{\mathcal{O}} c \text{ and } x \in \overline{N}(c)$$

A chain is bad if its representative is bad; and a part is bad if it contains a bad part. Notice that the choice of pivot is restricted to a *working factor*. But any refinement rule applies on any part (even those that do not belong to the working factor). The vertices of the working factor are those that are in $\mathcal{P}(c)$ when c becomes the new center. They are exactly the scope of the *center* rule that is used with c. Even after some split of $\mathcal{P}(c)$, it remains a factor of $\mathcal{O}$. It follows that the working factor contains no bad vertex wrt. the current center. The working factor is denoted $[\![i,j]\!]$, where i and j are two integers such that, for any linear extension σ of $\mathcal{O}$, x is in the working factor iff $i \leq \sigma(x) \leq j$.

The following invariant shows that the bad parts are "almost" modules (indeed they are the union of some strong modules) and explains the role of the working factor.

Invariant 3

1. *Let $x \in \mathcal{P}$ be a bad vertex. If a part $\mathcal{P}$ is bad, then all of its chains are bad, no strong module overlaps $\mathcal{P}$ and $\mathcal{P} \cap [\![i,j]\!] = \emptyset$.*
2. *Let $[\![i,j]\!]$ be the working factor and c the center. No part overlaps $[\![i,j]\!]$. If a strong module M overlaps $[\![i,j]\!]$ then $c \in M$.*

Line 11 of Algorithm of Figure 7 uses the new center once more as a pivot in order to avoid the existence of bad chain in the incoming working factor. It is worth to remark that the working factors are nested. Moreover the working factor returned by any recursive call only contains monochain parts (a total order on its vertices). As the whole vertex-set is the working factor of the main (initial) call, when it ends, V is linearly ordered in a factorizing permutation. Thus we have:

Theorem 1. *Algorithm of Figure 7 computes a factorizing permutation of a graph G.*

5 Linear-time implementation

The algorithm described above can be implemented to run in $O(n+m)$ time. It uses a simple implementation of the OCP presented below. Some hints of the complexity analysis are explained.

Ordered Chain Partition (see Figure 9) The *parts* of the OCP form a doubly linked list. A part itself has a doubly linked list of its representatives vertices. The order inside this list does not matter. The part of a representative vertex x is explicitly mentioned using an field $Part[x]$. Finally, each representative vertex points its part, and maintains and ordered list of the chain $\mathcal{C}(x)$, with pointers towards heads and tails. The concatenation of two chains can thus be performed in $O(1)$.

Implementation of Refine Procedure First, to *choose a pivot* in a given part, the algorithm simply selects the head of this list (line 4, Algorithm of Figure 7). Moreover, the *choice of an unused part* (line 4) can be done in $O(1)$ time,

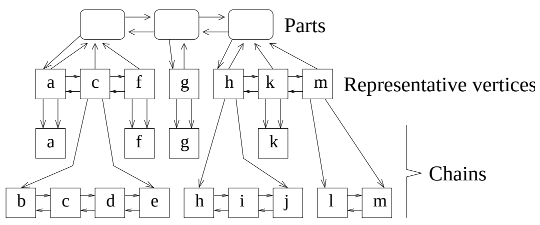


Fig. 9. Implementation of an Ordered Chain Partition.

within the working factor. Indeed it suffices to manage one stack per recursive call that contains such parts. Each time a new unused part is created (when a part is split), it is pushed in the corresponding stack. Finally, a search of the working factor from the part containing the current center, find, if they exist, the parts $\mathcal{P}_L$ and $\mathcal{P}_R$ (line 8, Algorithm of Figure 7). If the working factor is completely visited, then the recursion stop since no such part exists.

From the above discussion and Lemma 3, it follows that the overall running time of the recursive calls, apart the time spent by the refinement rule, is $O(n)$.

Lemma 3. *A given vertex can be used at most once as center and twice to extend the OCP.*

Implementation of refinement rules As in [16], the center rule (line 1), Algorithm of Figure 7) and the pivot rule (line 5) can be processed in $O(|N(x)|)$ time, where x is either the center or the pivot. For the chaining rule (line 9), the "closest" part that does not satisfies Property 1 should be find. A search in the list of parts is necessary and an adjacency test that should be done between the new center and the pivot of the current part. It is possible to show that these tests can be done in $O(1)$ amortized time.

Theorem 2. *Given a graph $G = (V, E)$, the time bound of Algorithm of Figure 7 is $O(n + m)$.*

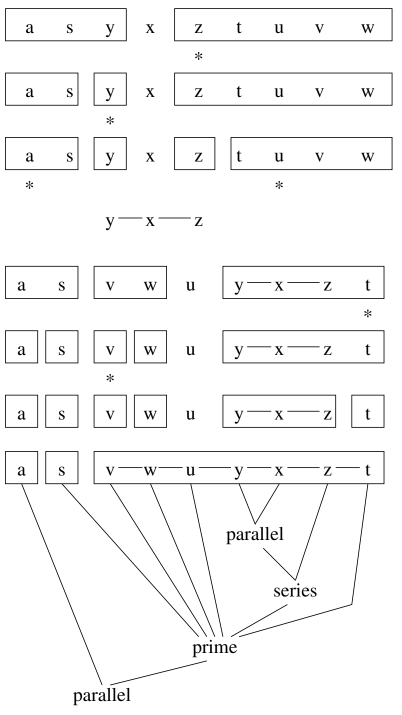
References

1. K.S. Booth and G.S. Lueker. Testing for the consecutive ones properties, interval graphs and graph planarity using PQ-tree algorithm. *J. Comput. Syst. Sci.*, 13:335–379, 1976.
2. A. Brandstädt, V.B. Le, and J. Spinrad. *Graph Classes: a Survey.* SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 1999.
3. C. Capelle and M. Habib. Graph Decompositions and Factorizing Permutations. In *proceedings of ISTCS'97*, pages 132–143, Ramat Gan (Israel), June 1997. IEEE.
4. C. Capelle, M. Habib, and F. de Montgolfier. Graph decomposition and factorizing permutations. *Discrete Mathematics and Theoretical Computer Sciences*, 5(1):55–70, 2002.

5. D.G. Corneil, Y. Perl, and L.K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal of Computing*, 14(4):926–934, 1985.
6. D.G. Corneil, S. Olariu, and L. Stewart. The ultimate interval graph recognition algorithm? In *Proceedings of the ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 175–180, 1998.
7. A. Cournier and M. Habib. A new linear algorithm for modular decomposition. In S. Tison, editor, *Trees in algebra and programming—CAAP 94, 19th International Colloquium, Edinburgh, U.K.*, volume 787 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, April 1994. Springer-Verlag.
8. D.D. Cowan, L.O. James, and R.G. Stanton. Graph decomposition for undirected graphs. In R. B. Levow eds. F. Hoffman, editor, *3rd S-E Conf. Combinatorics, Graph Theory and Computing, Utilitas Math*, pages 281–290, Winnipeg, 1972.
9. E. Dahlhaus, J. Gustedt, and R.M. McConnell. Efficient and practical algorithms for sequential modular decomposition. *Journal of Algorithms*, 41(2):360–387, 2001.
10. R. Ducournau and M. Habib. La multiplicité de l'héritage dans les langages à objects. *Technique et Science Informatique*, 8(1):41–62, 1989.
11. T. Gallai. Transitiv orientierbare Graphen. *Acta Math. Acad. Sci. Hungar.*, 18:25–66, 1967.
12. M.C. Golumbic. *Algorithmic graph theory and perfect graphs.* Academic Press, New-York, 1980.
13. M. Habib, M. Huchard, and J.P. Spinrad. A linear algorithm to decompose inheritance graphs into modules. *Algorithmica*, 13:573–591, 1995.
14. M. Habib, R. McConnell, C. Paul, and L. Viennot. Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234:59–84, 2000.
15. M. Habib, F. de Montgolfier, and C. Paul. A simple linear-time modular decomposition algorithm. Research Report *LIRMM, Université de Montpellier 2*, number RR-LIRMM-03007, April 2003.
`http://www.lirmm.fr/~montgolfier/plublications/`.
16. M. Habib, C. Paul, and L. Viennot. Partition refinement techniques: an interesting algorithmic toolkit. *International Journal of Foundations of Computer Science*, 10(2):147–170, 1999.
17. M. Habib, and C. Paul. A simple linear time algorithm for cograph recognition. *Discrete Mathematics*, To appear in 2004.
18. J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196, New York, 1971. Academic Press.
19. J. Hopcroft and R.E. Tarjan. Efficient planarity testing. *J. Assoc. Mach.*, 21:549–568, 1974.
20. W.-L. Hsu and T.-H. Ma. Substitution decomposition on chordal graphs and applications. In *Proceedings of the 2nd ACM-SIGSAM Internationnal Symposium on Symbolic and Algebraic Computation*, number 557 in LNCS, pages 52–60. Springer-Verlag, 1991.
21. W.-L. Hsu and R.M. McConnell. PC-trees and circular-ones arrangements. *Theoretical Computer Science*, 296:99–116, 2003.
22. R. M. McConnell and J. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms* (Arlington, VA), pages 536–545, New York, 1994. ACM.
23. R.M. McConnell and J.P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201:189–241, 1999.

24. R.M. McConnell and J.P. Spinrad. Ordered vertex partitioning. *Discrete Mathematics and Theoretical Computer Sciences*, 4:45–60, 2000.
25. R.H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In I. Rival, editor, *Graphs and Orders*, pages 41–101. D. Reidel Pub. Comp., 1985.
26. R.H. Möhring and F. J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *Annals of Discrete Mathematics*, 19:257–356, 1984.
27. F. de Montgolfier. Décomposition modulaire des graphes - théorie, extensions et algorithmes. *PhD Thesis, Université de Montpellier*, 2003.
28. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
29. A. Pnueli, S. Even, and A. Lempel. Transitive orientation of graphs and identification of permutation graphs. *Canad. J. Math.*, 23:160–175, 1971.

A Appendix: an example of execution



Vertex x is selcted as the first center.

1. z is used as pivot and split [a,s,y]

2. y is used as pivot and split [z,t,u,v,w]

3. a and u are used, but do not refine anything between u and a, the adjacency test select u as new center

x, y, z are chained and u is the new center

4. the working factor is [v...t]. u is the new center. t is used as pivot and split [a,s] and [v,w] Notice that part {a,s} outside the working factor is split.

5. v is used as pivot and split [y–x–z,t]

Any part of the working factor is monochain The part are linked. The center changes to a and the process stop since a is independant.

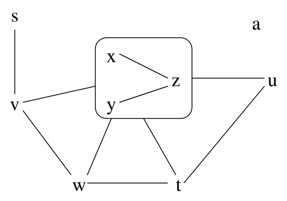


Fig. 10. The resulting factorizing permutation is $a, s, v, w, u, y, x, z, t$