available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cosrev

Survey

A survey of the algorithmic aspects of modular decomposition☆

Michel Habib[a], *Christophe Paul*[b,*]

[a] *LIAFA, Univ. Paris Diderot - Paris VII, France*
[b] *CNRS, LIRMM, Univ. Montpellier II, France*

ARTICLE INFO

Article history:
Received 8 December 2009
Received in revised form
4 January 2010
Accepted 4 January 2010

ABSTRACT

Modular decomposition is a technique that applies to (but is not restricted to) graphs. The notion of a module naturally appears in the proofs of many graph theoretical theorems. Computing the modular decomposition tree is an important preprocessing step to solve a large number of combinatorial optimization problems. Since the first polynomial time algorithm in the early 1970's, the algorithmic of the modular decomposition has known an important development. This paper survey the ideas and techniques that arose from this line of research.

1. Introduction

Modular decomposition is a technique at the crossroads of several domains of combinatorics which applies to many discrete structures such as graphs, 2-structures, hypergraphs, set systems and matroids, among others. As a graph decomposition technique, it was introduced by Gallai [1] to study the structure of comparability graphs (those graphs whose edge set can be transitively oriented). Roughly speaking, a *module* in a graph is a subset M of vertices which share the same neighbourhood outside M. Gallai showed that the family of modules of an undirected graph can be represented by a tree, the *modular decomposition tree*. The notion of a module appeared in the literature as *closed sets* [1], *clans* [2], *autonomous sets* [3], and *clumps* [4], while modular decomposition is also called *substitution decomposition* [5] or *X-join decomposition* [6]. See [7] for an early survey on this topic.

There is a large variety of combinatorial applications of modular decomposition. Modules can help in proving structural results on graphs as Gallai did for comparability graphs. More generally, modular decomposition appears in (but is not limited to) the context of perfect graph theory. Indeed Lovász's proof of the perfect graph theorem [8] involves clique modules. Notice also that a number of perfect graph classes can be characterized by properties of their modular decomposition tree: cographs, P_4-sparse graphs, permutation graphs, interval graphs, etc. Refer to the books of Golumbic [9], Brandstädt et al. [10] for graph classes. We should also mention that the modular decomposition tree is useful for solving optimization problems on graphs or other discrete structures (see [3]). An example of such use is given in the last section.

In the late 1970's, modular decomposition was independently generalized to *partitive set families* [11] and to a combinatorial decomposition theory [12] which applies to graphs, matroids and hypergraphs. More recently, the theory of partitive families and its variants had been the foundation of decomposition schemes for various discrete structures among

☆ Research supported by the project ANR-06-BLAN-148 "*Décomposition des graphes et algorithmes*".
* *Corresponding author.* Tel.: +33 0467418676; fax: +33 0467418676.
E-mail addresses: habib@liafa.jussieu.fr (M. Habib), paul@lirmm.fr (C. Paul).

1574-0137/$ - see front matter © 2010 Published by Elsevier Inc.
doi:10.1016/j.cosrev.2010.01.001

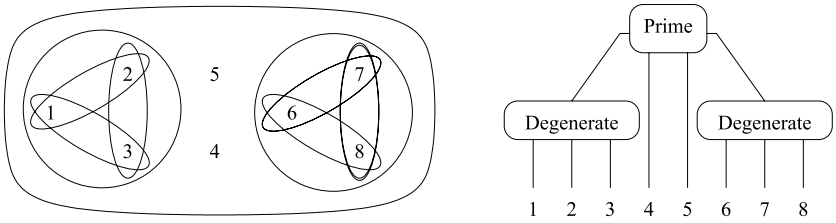

Fig. 1 – The inclusion tree of the strong elements of the family $\mathcal{S} = \{\{1, 2, 3, 4, 5, 6, 7, 8\}, \{1, 2, 3\}, \{6, 7, 8\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{6, 7\}, \{7, 8\}, \{6, 8\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}\}$.

which are 2-structures [13] and permutations [14,15]. Besides, based on efficiently representable set families, different graph decompositions have been proposed. The *split decomposition* of [12] relies on a *bipartitive family* on the vertex set. Refer to [16] for a survey of the recent developments of these techniques.

A good feature of most of these decomposition schemes is that they can be computed in polynomial time. Indeed, since the early 1970's, there have been a number of algorithms for computing the modular decomposition of a graph (or for some variants of this problem). The first polynomial algorithm is due to Cowan et al. [17], and it runs in $O(n^4)$. Successive improvements are due to Habib and Maurer [6], who proposed a cubic time algorithm, and to Müller and Spinrad, who designed a quadratic time algorithm. The first two linear time algorithms appeared independently in 1994 [18,19]. Since then a series of simplified algorithms has been published, some running in linear time [20,21], and others in almost linear time [22–24]. The list is not exhaustive. This line of research yields a series of new interesting algorithmic techniques, which, we believe, could be useful in other applications or topics of computer science. The aim of this paper is to survey the algorithmic theory of modular decomposition.

The paper is organized as follows. The partitive family theory and its application to modular decomposition of graphs is presented in Section 2. As an algorithmic appetizer, Section 3 addresses the special case of totally decomposable graphs, namely the *cographs*, for which a linear time algorithm has been known since 1985 [25]. *Partition refinement* is an algorithmic technique that reveals itself to be really powerful for the modular decomposition problem, but also for other graph applications (see *e.g.* [24,26]). Section 4 is devoted to partition refinement. Section 5 describes the principle of a series of modular decomposition algorithms developed in the mid 1990's. Section 6 explains how modular decomposition can be efficiently computed via the recent concept of *factoring permutation* [27]. Let us mention that we do not discuss the recent linear time algorithm of Tedder et al. [21], even though we believe that this last algorithm provides a positive answer to the problem of finding a simple linear time modular decomposition algorithm. Actually the key to Tedder et al.'s algorithm is to merge the ideas developed in Sections 5 and 6. The purpose of this paper is not to enter into the details of all the algorithmic techniques but rather to present their main lines. Finally, the last section presents three recent applications of modular decomposition in three different domains of computer science, namely pattern matching, computational biology and parameterized complexity.

2. Partitive families

The *modular decomposition theory* has to be understood as a special case of the theory of a *partitive family*, whose study dates back to the early 1980's [11,12]. We briefly present the main concepts and theorems of the partitive family theory. We then introduce the *modular decomposition* of graphs and discuss its elementary algorithmic aspects. This section ends with a discussion of two important classes of graph: indecomposable graphs (the *prime* graphs) and totally decomposable graphs (known as the *cographs*)

2.1. Decomposition theorem of partitive families

The *symmetric difference* between two sets A and B is denoted by $A \triangle B = (A \setminus B) \cup (B \setminus A)$. Two subsets A and B of a set S *overlap* if $A \cap B \neq \emptyset$, $A \setminus B \neq \emptyset$ and $B \setminus A \neq \emptyset$, we write $A \perp B$.

Definition 1. A family $\mathcal{S} \subseteq 2^S$ of subsets of S is *partitive* if:

(1) $S \in \mathcal{S}$, $\emptyset \notin \mathcal{S}$ and for all $x \in S$, $\{x\} \in \mathcal{S}$;
(2) For any pair of subsets $A, B \in \mathcal{S}$ such that $A \perp B$:
  (a) $A \cap B \in \mathcal{S}$;
  (b) $A \setminus B \in \mathcal{S}$ and $B \setminus A \in \mathcal{S}$;
  (c) $A \cup B \in \mathcal{S}$;
  (d) $A \triangle B \in \mathcal{S}$.

A family is *weakly partitive* whenever condition (2.d) is not satisfied. Unless explicitly mentioned, we will only consider partitive families.

Definition 2. An element $F \in \mathcal{S}$ is *strong* if it does not overlap any other element of $\mathcal{S}$. The set of strong elements of $\mathcal{S}$ is denoted $\mathcal{S}_F$.

Obviously any trivial subset of $\mathcal{S}$, namely S or $\{x\}$ (for $x \in S$), is a strong element. Let us remark that $\mathcal{S}_F$ is nested, *i.e.* the transitive reduction of the inclusion order of $\mathcal{S}_F$ is a tree $T_{\mathcal{S}}$, which we call the *strong element tree* (see Fig. 1). It follows that $|\mathcal{S}_F| = O(|S|)$.

Definition 3. Let $f_1^q, \ldots, f_k^q$ be the children of a node q of $T_{\mathcal{S}}$, the strong element tree of $\mathcal{S}$. The node q is *degenerate* if, for every non-empty subset $J \subset [1, k]$, $\cup_{j \in J} f_j^q \in \mathcal{S}$. A node is *prime* if, for every non-empty subset $J \subset [1, k]$, $\cup_{j \in J} f_j^q \notin \mathcal{S}$.

It is not difficult to see that any strong element is either prime or degenerate. Moreover, the following theorem tells us that the tree $T_{\mathcal{S}}$ is a representation of the family $\mathcal{S}$ and the subfamily of strong elements $\mathcal{S}_F$ of $\mathcal{S}$ defines a "basis" of $\mathcal{S}$.

Theorem 1 ([11]). *Let $\mathcal{S}$ be a partitive family on S. The subset $A \subseteq S$ belongs to $\mathcal{S}$ if and only if A is strong or there exists a degenerate strong element A' (or a node of $T_{\mathcal{S}}$) such that A is the union of a strict subset of the children of A' in $T_{\mathcal{S}}$.*

As a consequence, even if a partitive family on a set S can have exponentially many elements, it always admits a representation that is linear in the size of S. Such a representation property is also known for other families of subsets of a set, such as laminar families and cross-free families [28], as well as for some families of bipartitions of a set, such as splits [12]. Recently, a similar result has been shown for *union-difference* families of subsets of a set, i.e. families closed under the union and the difference of its overlapping elements [29]. In this latter case, the size of the representation amounts to $O(|S|^2)$. For a detailed study of these aspects, the reader should refer to [16].

2.2. Factoring permutations

Although the idea of *factoring permutation* implicitly appeared in some early papers (see *e.g.* [30–32]), it has only been formalized in [33,34]. This concept turns out to be central to recent modular decomposition algorithms and other applications.

Let σ be a permutation of a set S of size n. By $\sigma(x)$, we mean the rank i of x in σ, and $\sigma^{-1}(i)$ stands for the i-th element of σ. A subset $I \subseteq S$ is a *factor* or an *interval* of a permutation σ if there exist $i \in [1, n]$ and $j \in [1, n]$ such that $I = \{x \mid x = \sigma^{-1}(k), i \leqslant k \leqslant j\}$. In other words, the elements of I occur consecutively in σ.

Definition 4 ([34]). Let $\mathcal{S}$ be a (weakly) partitive family of a set S and let $\mathcal{S}_F$ be the strong elements of $\mathcal{S}$. A permutation σ of S is *factoring* for $\mathcal{S}$ if, for any $F \in \mathcal{S}_F$, F is a factor of σ.

For example, $\pi = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8$, $\pi_1 = 6\ 7\ 8\ 4\ 3\ 1\ 2\ 5$ and $\pi_2 = 8\ 7\ 6\ 1\ 3\ 2\ 4\ 5$ are three factoring permutations of the family $\mathcal{S}$ depicted in Fig. 1. One can check that, in each of these three permutations, the two non-trivial strong elements of $\mathcal{S}_F$, namely $\{1, 2, 3\} \in \mathcal{S}_F$ and $\{6, 7, 8\} \in \mathcal{S}_F$, are factors.

Given a layout of the strong element tree of a partitive family, a left-to-right enumeration of the leaves results in a factoring permutation. In many cases it is easier to compute a factoring permutation than the strong element tree. We explain in Section 6.3 how to obtain the strong element tree from a factoring permutation.

To conclude this brief introduction on factorizing permutation, we state a lemma which formalizes links between intervals of factoring permutations and partitive families. This lemma somehow guided the development of factoring permutation algorithms.

Lemma 1. *Let σ be a factoring permutation of a partitive family $\mathcal{S}$. Then the set $\mathcal{I}(\mathcal{S}, \sigma)$ of intervals of σ which are elements of $\mathcal{S}$ is a weakly partitive family. Moreover, the strong elements of $\mathcal{I}(\mathcal{S}, \sigma)$ and of $\mathcal{S}$ are the same.*

2.3. Modules of a graph

For the sake of the presentation we only consider undirected, simple and loopless graphs. We use the classical notations (*e.g.* see [10]). The neighbourhood of a vertex x in a graph $G = (V, E)$ is denoted $N_G(x)$ and its non-neighbourhood $\overline{N}_G(x)$ (subscript G will be omitted when the context is clear). The complementary graph of a graph G is denoted by $\overline{G}$. Given a subset of vertices $X \subseteq V$, $G[X]$ is the subgraph induced by X (any edge in G between two vertices in X belongs to $G[X]$).

Let M be a set of vertices of a graph $G = (V, E)$ and x be a vertex of $V \setminus M$. Vertex x *splits* M (or is a *splitter* of M), if there exist $y \in M$ and $z \in M$ such that $xy \in E$ and $xz \notin E$. If x is not a splitter of M, then M is *uniform* or *homogeneous* with respect to x.

Definition 5. Let $G = (V, E)$ be a graph. A set $M \subseteq V$ of vertices is a *module* if M is homogeneous with respect to any $x \notin M$ (i.e. $M \subseteq N(x)$ or $M \cap N(x) = \emptyset$).

Observation 1. *Let S be a subset of vertices of a graph $G = (V, E)$. If S has a splitter x, then any module of G containing S also contains x.*

Aside the singletons and the whole vertex sets, any union of connected components (or of co-connected components) of a graph are simple examples of modules. Let us also note that a graph may have exponentially many modules. Indeed any subset of a complete graph is a clique. Nevertheless, as we shall see with the following lemma, the family of modules has strong combinatorial properties.

Lemma 2 ([11]). *The family $\mathcal{M}$ of modules of a graph is partitive.*

The notions of *trivial* and *strong* module and *degenerate* are defined according to the terminology of Section 2.1. By Lemma 2, if M and M' are overlapping modules, then $M \setminus M'$, $M' \setminus M$, $M \cap M'$, $M \cup M'$ and $M \triangle M'$ are modules of G.

Let M and M' be disjoint sets. We say that M and M' are *adjacent* if any vertex of M is adjacent to all the vertices of M' and *non-adjacent* if the vertices of M are non-adjacent to the vertices of M'.

Observation 2. *Two disjoint modules are either adjacent or non-adjacent.*

A module M is *maximal* with respect to a set S of vertices, if $M \subset S$ and there is no module M' such that $M \subset M' \subset S$. If the set S is not specified, we shall assume that $S = V$.

Definition 6. Let $\mathcal{P} = \{M_1, \ldots, M_k\}$ be a partition of the vertex set of a graph $G = (V, E)$. If, for all i, $1 \leqslant i \leqslant k$, M_i is a module of G, then $\mathcal{P}$ is a *modular partition* (or *congruence* partition) of G.

A non-trivial modular partition $\mathcal{P} = \{M_1, \ldots, M_k\}$ which only contains maximal strong modules is a *maximal modular partition*. Notice that each graph has a unique maximal modular partition. If G (resp. $\overline{G}$) is not connected then its connected (resp. co-connected) components are the elements of the maximal modular partition. From Observation 2, we can define a *quotient graph* whose vertices are the parts (or modules) belonging to the modular partition $\mathcal{P}$.

Definition 7. To a modular partition $\mathcal{P} = \{M_1, \ldots, M_k\}$ of a graph $G = (V, E)$, we associate a *quotient graph* $G_{/\mathcal{P}}$, whose vertices are in one-to-one correspondence with the parts of $\mathcal{P}$. Two vertices v_i and v_j of $G_{/\mathcal{P}}$ are adjacent if and only if the corresponding modules M_i and M_j are adjacent in G.

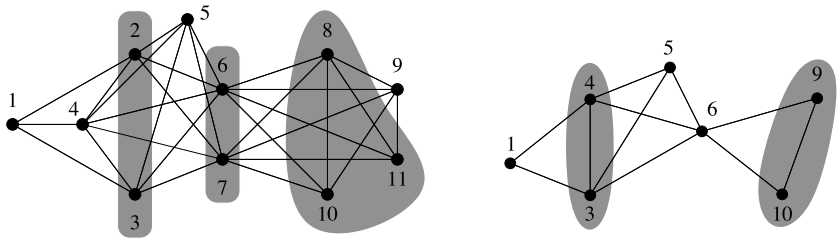

Fig. 2 – On the left, the grey sets are modules of the graph G. $\mathcal{Q} = \{\{1\}, \{2, 3\}, \{4\}, \{5\}, \{6, 7\}, \{9\}, \{8, 10, 11\}\}$ is a modular partition of G. The quotient graph $G_{/\mathcal{Q}}$, depicted on the right with a representative vertex for each module of $\mathcal{Q}$, has two non-trivial modules (the sets $\{3, 4\}$ and $\{9, 10\}$). The maximal modular partition of G is $\mathcal{P} = \{\{1\}, \{2, 3, 4\}, \{5\}, \{6, 7\}, \{8, 9, 10, 11\}\}$ and its quotient graph are represented in Fig. 3 (aside the top node of the tree).

Let us remark that the quotient graph $G_{/\mathcal{P}}$ with $\mathcal{P} = \{M_1, \ldots, M_k\}$ is isomorphic to any subgraph induced by a set $V' \subseteq V$ such that $\forall i \in [1, k]$, $|M_i \cap V'| = 1$. The *representative graph* of a module M is the quotient graph $G[M]_{/\mathcal{P}}$, where $\mathcal{P}$ is the maximal modular partition of $G[M]$: it is thereby the subgraph induced by a set containing a unique – *representative* – vertex per maximal strong module of $G[M]$. See Fig. 2. By extension, for a module M, we denote by $G_{/M}$ the graph quotiented by the modular partition $\{M\} \cup \{\{x\} \mid x \notin M\}$.

Before we state the *modular decomposition theorem* (Theorem 2), let us present two more properties of modular partitions and quotient graphs which are central to efficient modular decomposition algorithms (see Section 5).

Lemma 3 ([3]). *Let $\mathcal{P}$ be a modular partition of a graph $G = (V, E)$. Then $\mathcal{X} \subseteq \mathcal{P}$ is a module of $G_{/\mathcal{P}}$ iff $\bigcup_{M \in \mathcal{X}} M$ is a module of G.*

Lemma 3 is illustrated in Fig. 2: for example, the set $\{2, 3, 4\}$ is a module of G; it is the union of modules $\{2, 3\}$ and $\{4\}$ (which representative vertices are respectively 3 and 4 in $G_{/\mathcal{Q}}$) which belongs to partition $\mathcal{Q}$. It can be strengthened in order to observe the correspondence between the strong modules of G and those of $G_{/\mathcal{P}}$.

Lemma 4. *Let $\mathcal{P}$ be a modular partition of a graph $G = (V, E)$. Then $\mathcal{X} \subset \mathcal{P}$ is a non-trivial strong module of $G_{/\mathcal{P}}$ iff $\bigcup_{M \in \mathcal{X}} M$ is a non-trivial strong module of G.*

The inclusion tree of the strong modules of G, denoted $MD(G)$, entirely represents the graph if the representative graph of each strong module is attached to each of its nodes (see Fig. 3). Indeed any adjacency of G can be retrieved from $MD(G)$. Let x and y be two vertices of G and let G_N be the representative graph of node N, their least common ancestor. Then x and y are adjacent in G if and only if their representative vertices in G_N are adjacent.

Let us recall that a graph is *prime* if it only contains trivial modules.

Theorem 2 (*Modular Decomposition Theorem* [1,11]). *For any graph $G = (V, E)$, one of the following three conditions is satisfied:*

(1) *G is not connected;*
(2) *$\overline{G}$ is not connected;*
(3) *G and $\overline{G}$ are connected and the quotient graph $G_{/\mathcal{P}}$, with $\mathcal{P}$ the maximal modular partition of G, is a prime graph.*

What does the modular decomposition theorem say is twofold. First, the quotient graphs associated with the nodes of the inclusion tree $MD(G)$ of the strong modules are of three types: an *independent set* if G is not connected (the node is labelled *parallel*); a *clique* (complete graph) if $\overline{G}$ is not connected (the node is labelled *series*); a prime graph otherwise. It also follows that $MD(G)$ is unique and does not contain two consecutive series nodes or two consecutive parallel nodes. Parallel and series nodes of $MD(G)$ are also called *degenerate* nodes.

The tree $MD(G)$ is called the *modular decomposition tree*. Theorem 2 yields a natural polynomial time recursive algorithm to compute $MD(G)$: (1) compute the maximal modular partition $\mathcal{P}$ of G; (2) label the root node according to the parallel, series or prime type of G; (3) for each module M of $\mathcal{P}$, compute $MD(G[M])$ and attach it to the root node. A subproblem central to the computation of $MD(G)$ is to compute the maximal modular partition, a task which can be avoided if a non-trivial module M is identified. This yields another natural algorithm scheme: by Lemmas 3 and 4, it suffices to recursively compute $MD(G[M])$ and $MD(G_{/M})$, and then to paste $MD(G[M])$ on the leaf of $MD(G_{/M})$ corresponding to the representative vertex of M. As suggested by Cowan et al. [17], a naive way to compute a non-trivial module is to follow the definition of a module and Observation 1. Assume that the graph G contains a non-trivial module M. Then M contains a pair of vertices $\{x, y\}$ and as a module is closed under adding splitters. Such an algorithm would find a non-trivial module, if any, in time $O(n^2(n+m))$. We should note that for some generalizations of modular decomposition, no better algorithm than this "closure by splitter" approach is known (see e.g. [35]).

Before we present some structural properties of prime and totally decomposable graphs, let us introduce some notations and briefly discuss the composition view of the theory of modules in graphs.

Notation 3. *For a node p of $MD(G)$, its corresponding strong module is denoted by $M(p)$ (or P). In fact $M(p)$ is the union of all singletons which are leaves of the subtree of $M(p)$ rooted in p.*

The minimal strong module containing two vertices x and y is denoted by $m(x, y)$, while the maximal strong module containing x but not y, for any two different vertices x, y of G, is denoted by $M(x, \overline{y})$.

The *substitution* operation is the reverse of the quotient operation. It consists of replacing a vertex x of G by a

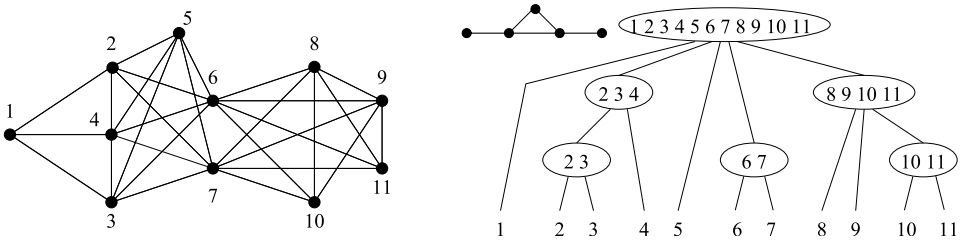

Fig. 3 – The inclusion tree $MD(G)$ of the strong modules of G. The representative graph associated to the root is $G_{/\mathcal{P}}$ with $\mathcal{P} = \{\{1\}, \{2, 3, 4\}, \{5\}, \{6, 7\}, \{8, 9, 10, 11\}\}$, the parts of which correspond to the children of the root.

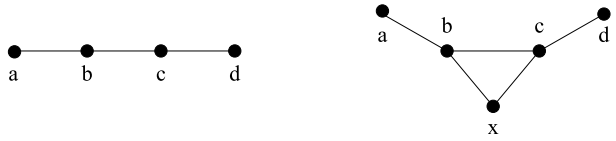

Fig. 4 – The vertices a, b, c, d form a P_4 whose *extremities* are a and d, and midpoints are b and c. The graph on the right is the *bull* whose "*nose*" is vertex x.

graph $H = (V', E')$ while preserving the neighbourhood. The resulting graph is

$$G_{x \to H} = ((V \setminus \{x\}) \cup V', (E \setminus \{xy \in E\}) \cup E' \cup \{yz : xy \in E \text{ et } z \in V'\}).$$

The *parallel composition* or *disjoint union* of k connected graphs $G_1, \ldots G_k$ defines a graph whose connected components are the graphs $G_1, \ldots, G_k$. This composition operation is usually denoted $G_1 \oplus \cdots \oplus G_k$.

The *series composition* of k co-connected graphs $G_1, \ldots, G_k$ defines a graph whose co-connected components are the graphs $G_1, \ldots, G_k$ (for any pair x, y of vertices belonging to different graphs G_i and G_j, the edge xy has been added). The series composition is generally denoted $G_1 \otimes \cdots \otimes G_k$.

These three operations are classical graph operations that have been widely used in various contexts among which is the clique-width theory [36].

2.4. *Prime graphs*

The structure of prime graphs has been extensively studied (*e.g.* see [37–39]). For example, it is easy to check that the smallest prime graph is the P_4, the path on four vertices (see Fig. 4). As witnessed by the following result, P_4's play an important role in the structure of prime graphs.

Lemma 5 ([39]). *Let G, with $|G| \geq 4$, be a prime graph. Then any vertex, but at most one, is contained in an induced P_4. A vertex not contained in any P_4 is called the "nose of the bull" (see Fig. 4).*

The next property shows that one can always remove one or two vertices from a large enough prime graph to obtain a new prime graph.

Lemma 6 ([37,38]). *Let $G = (V, E)$ be a prime graph with at least five vertices. Then there exists a subset of vertices X such that $|V| - 2 \leqslant |X| \leqslant |V| - 1$ and $G[X]$ is prime.*

Jamison and Olariu proposed an extension of Theorem 2 by considering the structure of prime graphs [40]. A subset C of vertices of a graph $G = (V, E)$ is *P-connected* if, for any bipartition $\{A, B\}$ of C, there is an induced P_4 intersecting both A and B. For example, the bull is not P-connected (consider the vertex partition $\{\{x\}, \{a, b, c, d\}\}$). A P-connected component is a maximal P-connected set of vertices. The set of P-connected components defines a partition of the vertices. A P-connected component H is *separable* if there is a bipartition (H_1, H_2) of H such that, for any P_4 intersecting H_1 and H_2, the extremities are in H_1 and the mid-vertices are in H_2.

Theorem 4 ([40]). *Let $G = (V, E)$ be a connected graph such that $\overline{G}$ is connected. Then G is either P-connected or there exists a unique P-connected component H which is separable in (H_1, H_2) such that, for any vertex $x \notin H$, $H_1 \subseteq N(x)$ and $H_2 \cap N(x) = \emptyset$.*

A hierarchy of graph families has been proposed based on the above Theorem 4 by restricting the number of induced P_4's in small subgraphs (or equivalently by restricting the structure of prime graphs). For example, P_4-sparse graphs are defined as the graphs for which there is at most one P_4 in any induced subgraph on five vertices [41,42]. Let us also mention the P_4-reducible graphs [40]. See [10] for a complete presentation of these graph families.

2.5. *Totally decomposable graphs*

A graph is *totally decomposable* if any induced subgraph of size at least 4 has a non-trivial module. As any prime graph contains a P_4, it follows from Theorem 1 that any node of the modular decomposition tree $MD(G)$ of a totally decomposable graph G is degenerate.

The family $\mathcal{F}$ of totally decomposable graphs is natural, and it arose in many different contexts (see [25,43,44] for references) even recently (see [45,46]) as any graph of $\mathcal{F}$ can be obtained by a sequence of disjoint and series compositions starting from a single vertex graph. Let us remark that if G is totally decomposable then its complement also is. The family of totally decomposable graphs is also known as the *cographs* for *complement reducible graphs* [43,44]. From definition, the cograph family is *hereditary* (any induced subgraph of a cograph is a cograph). It also has a very simple forbidden subgraph characterization.

Theorem 5 ([43]). *The cographs are exactly the P_4-free graphs (Fig. 5).*

The following lemma states the classical properties of cographs whose proofs (left to the reader) are good exercises to understand the structure of cographs.

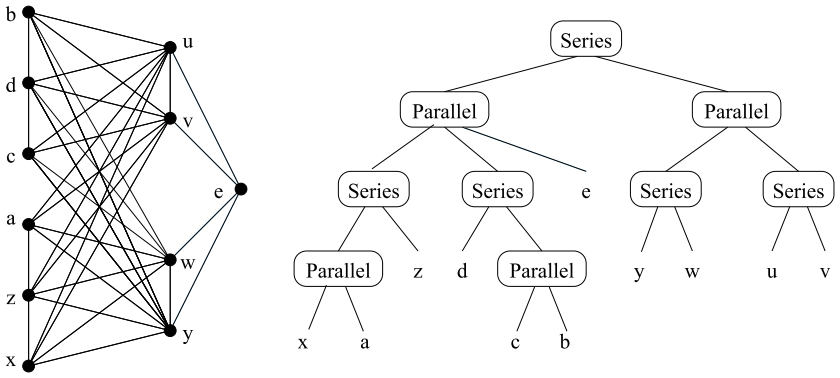

Fig. 5 – A cograph and its modular decomposition tree (also called a cotree).

Lemma 7. *Let x, y and v be vertices of a cograph $G = (V, E)$.*

(1) *If $xv \in E$, $yv \notin E$ and $xy \in E$, then $m(v, y) \subseteq M(v, \overline{x})$.*
(2) *If $xv \in E$, $yv \in E$ and $xy \notin E$, then $M(v, \overline{x}) = M(v, \overline{y})$ and $m(v, x) = m(v, y)$.*

Using Theorem 5, one can propose a naive cograph recognition algorithm by searching for an induced P_4. But so far, most of the linear time cograph recognition algorithms construct the modular decomposition tree and exhibit a P_4 in the case of failure.

The first linear time cograph recognition algorithm was proposed in 1985 by Corneil et al. [25]. It incrementally constructs the modular decomposition tree, also called the *cotree* when restricted to cographs, as long as the graph induced by the processed vertices is a cograph. Even if alternative recognition algorithms have recently been proposed [47–49], the seminal algorithm of [25] is a cornerstone in the algorithmic of the modular decomposition and turns out to have a large impact even for other decomposition technics (e.g. for the split decomposition [50]). We present Corneil et al.'s algorithm in Section 3.

2.6. *Bibliographic notes*

The seminal paper on modular decomposition of graphs is probably Gallai's one [1] on transitive orientation. To knowledge, the only survey paper is due to Möhring and Radermacher [7]. More recently, Ehrenfeucht et al. [13] published a book on the decomposition of 2-structures (a generalization of graphs) which presents the modular decomposition in a more general framework. In his PhD thesis [16], Bui Xuan proposes a survey as well as original results on the representation of set families. Many graph families are well-structured with respect to the modular decomposition, e.g. comparability graphs, permutation graphs, cographs, etc. For these aspects, the reader should refer to the books of Golumbic [9] and more recently [10,51]. The algorithmic aspects are particularly developed in [9,51].

We saw that the family of modules in a graph is partitive. If we move to directed graphs, then we obtain a weakly partitive family. The related decomposition of a bipartite graph into bimodules also yields a weakly partitive family [52]. In order to formalize split decomposition [12], bipartitive families have been introduced [12,53]. For a recent survey of all kinds of variations of modular decomposition, the reader should refer to [16].

3. Cographs recognition algorithms as an appetizer

We first study in detail Corneil et al.'s algorithm [25]. If the input graph is a cograph, this vertex-incremental algorithm builds the cotree by adding the vertices one by one in an arbitrary order. Then, we sketch how the cotree of a cograph can be updated under edge modification; the result is due to Shamir and Sharan [54].

3.1. *Adding a vertex to a cograph*

Consider the following subproblem: given a cograph $G = (V, E)$ together with its cotree $MD(G)$, a vertex x and a subset of vertices $S \subseteq V$, test whether the graph $G + (x, S) = (V \cup \{x\}, E \cup \{xy \mid y \in S\})$ is a cograph, and if so output the cotree $MD(G+x)$. Corneil et al. [25] showed that whether $G+x$ is a cograph or not can be characterized by a labelling of the nodes of the cotree $MD(G)$. A node p receives the label *empty* if the corresponding module $M(p)$ does not intersect S; *adjacent* if $M(p) \subseteq S$; and *mixed* otherwise. We remark that by definition any child of a node labelled adjacent (resp. empty) is also labelled adjacent (resp. empty).

Lemma 8 ([25]). *Let G be a cograph, x a vertex of V and $S \subseteq V$. The graph $G + (x, S)$ is a cograph iff*

(1) *either none of the nodes of the cotree $MD(G)$ is mixed;*
(2) *or the set of mixed nodes induces a path π from the root of $MD(G)$ to some node p and*
   (a) *the children of the series nodes of π different than p are all adjacent;*
   (b) *the children of the parallel nodes of π different than p are all empty.*

The main idea expressed by the conditions of Lemma 8 is that the modifications of the cotree implied by the insertion of vertex x are localized in the subtree of $MD(G)$ rooted at node p. Indeed any module disjoint from $M(p)$ is not affected by x's insertion (the corresponding nodes are labelled empty or adjacent). In a sense, node p should be considered as the

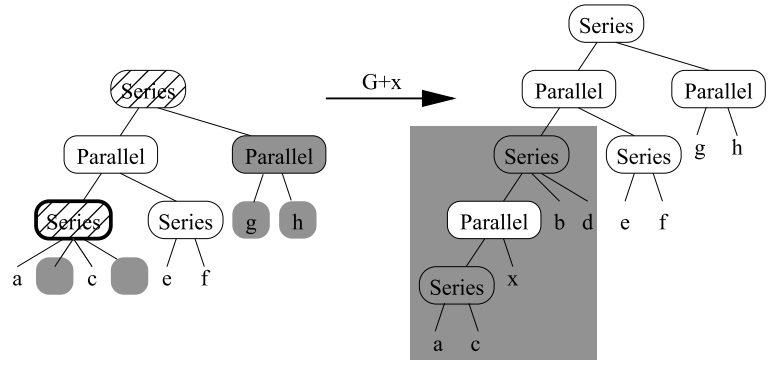

Fig. 6 – Insertion of the vertex x adjacent to $S = \{b, d, g, h\}$. Grey nodes are the adjacent labelled nodes and dashed nodes are the mixed nodes. The insertion node p is the bold series node (father of a, b, c, d).

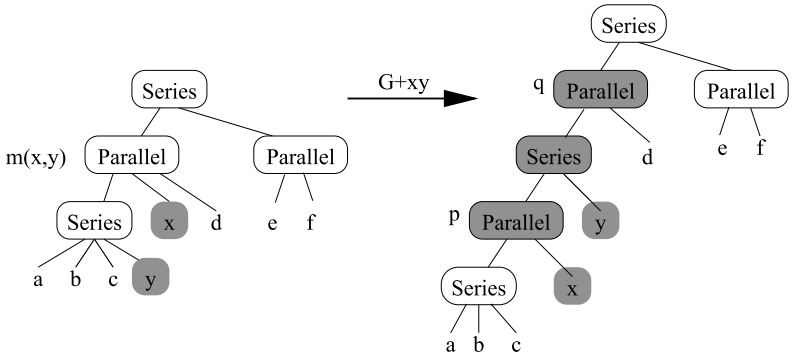

Fig. 7 – Update of the cotree to insert the edge xy in a cograph. The node $m(x, y)$ is split into two parallel nodes, say p and q, one being the father of x, the other the father of the other children of $m(x, y)$. Then leaf y is extracted from the cotree and attached to a new series node inserted between nodes p and q.

insertion node. The cotree updates only depend on node p (*e.g.* whether it is mixed or adjacent). An example is depicted in Fig. 6.

The algorithm first labels the cotree in a bottom-up manner. The leaves corresponding to vertices of S are labelled adjacent. A node labelled adjacent forwards a partial mark to its father. When a node has received a mark from each of its children, it is labelled adjacent. At the end of this process the empty nodes have never been searched, while the partially marked nodes correspond, if $G + x$ is a cograph, to the parallel nodes of the path π from the insertion node to the root of $MD(G)$. It is not difficult to see that the number of marked nodes is linear in the size of S, meaning that the labelling process runs in time $O(|S|)$. Testing the condition of the above lemma can be done within the same complexity as well.

Theorem 6 ([25]). *The family of cographs can be recognized in linear time.*

3.2. *Edge modification algorithms for cographs*

Let us now turn to the edge modification problem which consists in updating the cotree of a cograph G under an edge insertion or deletion. Since the cotree of a cograph can be obtained from the cotree of its complement by flipping the parallel and the series nodes, deleting or inserting an edge in a cograph are equivalent problems.

Lemma 9 ([54]). *Let x and y be two non-adjacent vertices of a cograph $G = (V, E)$. Then $G + xy = (V, E \cup \{xy\})$ is a cograph iff x is a child of $m(x, y)$ and $M(y, \overline{x}) \subseteq N(x)$.*

Let us sketch the argument proof. As $xy \notin E$, the module $m(x, y)$ is represented by a parallel node. Assume that the conditions of Lemma 9 do not hold. Then the path in the cotree from $m(x, y)$ to x (resp. y) contains a series nodes p_x (resp. p_y) which is the least common ancestor of x (resp. y) and some leaf u_x (resp. u_y). Then the vertices $\{u_x, x, y, u_y\}$ induce a P_4 in the graph $G + xy$ (Fig. 7).

It follows from Lemma 9 that, as long as the modified graph remains a cograph, the modifications in the cotree are local and can be done in constant time. From the results presented in this section, we obtain the following.

Theorem 7 ([25,54]). *There exists an algorithm maintaining the modular decomposition tree of a cograph which runs in time $O(d)$ per modification (edge or vertex insertion and deletion), where d is the number involved in the modification.*

Such an algorithm is known in the literature as a *fully dynamic* algorithm.

3.3. *Bibliographic notes*

In the late 1980's, Müller and Spinrad generalized Corneil et al.'s algorithm to the first quadratic modular decomposition

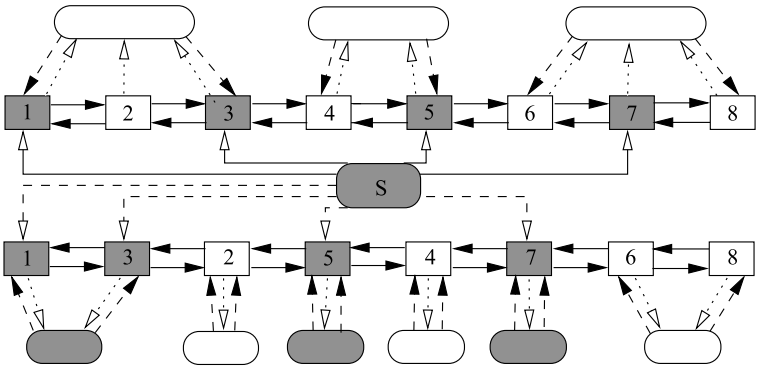

Fig. 8 – $\mathcal{P}' = \text{Refine}(\mathcal{P}, S)$.

algorithm of graphs [55]. Their algorithm is also incremental, but unlike in Corneil et al.'s algorithm, the whole graph has to be known at the beginning of the algorithm. This restriction is required for the sake of adjacency tests.

Concerning the cograph recognition problem, new algorithms have also appeared recently. Habib and Paul [48] proposed a partition refinement based algorithm (see Section 4) and Bretscher et al. [56] discovered a simple Lexicographic Breadth First Search [57] based algorithm.

As well as the two cograph algorithmic results presented above, fully dynamic algorithms have recently been proposed to maintain a representation based on the modular decomposition tree under vertex and edge modifications for various graph classes: permutation graphs [58], interval graphs [59,60], etc. The fully dynamic representation problem has also been solved for other families of graphs, *e.g.* proper interval graphs [61], using other decomposition schemes.

In addition, Corneil et al.'s algorithm has been generalized to the split decomposition [12] to obtain an optimal fully dynamic algorithm for the distance hereditary graphs recognition problem [50]. More recently, by the same technique, Gioan et al. derived an almost linear time split decomposition algorithm [62] and the first subquadratic circle graph recognition algorithm [63].

4. Partition refinement

Partition refinement, as an algorithmic technique, has been used in a number of problems, the first of which is probably the deterministic automata minimization [64]. Paigue and Tarjan [26] wrote a synthesis paper on this technique. Since then, the number of problems solved by partition refinement keeps increasing: interval graph recognition [24] and completion [65], transitive orientation, consecutive ones property for boolean matrices [66] are examples, among others. As we will see, this technique turns out to be a powerful and simple algorithmic paradigm that plays an important role in the context of modular decomposition.

We first present the data-structure and the elementary operation, namely the *refine* operation, of the partition refinement technique. Then, we illustrate this technique with an algorithm that computes a modular partition of a graph. Let us mention that this algorithm really follows the lines of Hopcroft's deterministic automaton minimization algorithm [64].

4.1. Data-structures and algorithmic scheme

Let $\mathcal{P}$ and $\mathcal{P}'$ be two partitions of the same set V. The partition $\mathcal{P}$ is *smaller* than $\mathcal{P}'$, denoted $\mathcal{P} \lhd \mathcal{P}'$, if $\mathcal{P} \neq \mathcal{P}'$ and any part of $\mathcal{P}$ is a subset of some part of $\mathcal{P}'$. The partition $\mathcal{P}$ is *stable* with respect to a set S if none of the parts of $\mathcal{P}$ overlaps S.

Partition refinement consists of repeating, as long as needed, the operation described in Algorithm 1. The initial partition and the sequence of *pivot sets* used in the successive refinement steps have a large impact on the whole complexity of the algorithm. Partitioning the vertex set of a graph with respect to the neighbourhood of some vertex is a common operation in graph algorithms. Indeed, in our examples, all pivot sets considered correspond to the neighbourhood of some vertex.

Algorithm 1: Refine($\mathcal{P}$, S)

Input: A partition $\mathcal{P}$ of a set V and a subset $S \subseteq V$, called *pivot set*
Output: The coarsest partition refining $\mathcal{P}$ and stable for S
begin
  foreach *part* $\mathcal{X} \in \mathcal{P}$ **do**
    if $\mathcal{X} \cap S \neq \emptyset$ *and* $\mathcal{X} \cap S \neq \mathcal{X}$ **then** replace $\mathcal{X}$ by $\mathcal{X} \cap S$ and $\mathcal{X} \setminus S$
end

Let us briefly describe a very useful data-structure, namely the *standard partition data structure* (see Fig. 8). The elements of the set V to be partitioned are stored in a doubly linked list. Each element of V is assigned a pointer towards the part it belongs to. The elements of a part $\mathcal{X}$ remain consecutive in the doubly linked list (they form an interval). So each part maintains a pointer towards its first and its last element in the list.

Notation 8. *The data-structure implicitly represents an ordered partition: the parts are totally ordered. Depending on the application, this aspect may or may not be important. In order to distinguish*

the two different cases, an ordered partition will be denoted by $\mathcal{P} = [\mathcal{X}_1, \ldots, \mathcal{X}_k]$ *while a non-ordered partition will be denoted by* $\mathcal{P} = \{\mathcal{X}_1, \ldots, \mathcal{X}_k\}$.

Given a subset $S \subseteq V$, using this standard partition data structure, one can build a list L containing the parts of $\mathcal{P}$ intersecting S, such that in each of these parts the elements of S occur first. Then, using L, one can split every part into $\mathcal{X} \cap S$ and $\mathcal{X} \setminus S$. A careful complexity analysis shows the following result.

Lemma 10. *The time complexity of the operation Refine*$(\mathcal{P}, S)$ *is* $O(|S|)$.

We conclude this brief introduction with a few remarks. Refining a partition by a subset S or its complement $\overline{S} = V \setminus S$ are equivalent operations: Refine$(\mathcal{X}, S)$ = Refine$(\mathcal{X} \cap S, V \setminus S)$. It is thereby possible to deal with the complement of the input graph without explicitly storing its edge set. Partition refinement is usually used either to compute a total ordering of the vertices (*e.g.* LexBFS) or the equivalence classes of some equivalence relations (*e.g.* maximal set of twin vertices). McConnell and Spinrad [23] showed how to augment the data-structure in order to extract within the same complexity, at each refinement step, the edges incident to vertices belonging to different parts. This operation is useful to efficiently compute the quotient graph associated to a modular partition. For a more detailed presentation of partition refinement, refer to [24,26,66,67].

Of course many variations of the standard partition data structure have been introduced, as for example changing the doubly linked list into an array of size $|V|$. A further requirement can be that the elements of every part $\mathcal{X}$ of $\mathcal{P}$ are maintained sorted according to a given an initial ordering τ of V. This can be done within the same complexity and is very useful for example when dealing with LexBFS multi-sweep algorithms. The ordering given by some previous LexBFS can be used as a tie-break rule for another LexBFS [56,68,69].

4.2. *Hopcroft's rule and computation of a modular partition*

Partition refinement is the right tool to compute a modular partition, an important subproblem towards efficient modular decomposition algorithms. In this section, we focus on the problem of computing the *coarsest modular partition* (see Definition 8) of a given vertex partition. The algorithm we present runs in time $O(n + m \log n)$ and is based on *Hopcroft's rule*, which is used in various simple quasi-linear time modular decomposition algorithms.

Definition 8. Let $\mathcal{P}$ be a partition of the vertices of a graph $G = (V, E)$. The *coarsest modular partition of* G *with respect to* $\mathcal{P}$ is the largest modular partition $\mathcal{Q}$ such that $\mathcal{Q} \lhd \mathcal{P}$.

The main idea of the algorithm is the following: as long as there is a part $\mathcal{X}$ which is not uniform for some vertex $x \notin \mathcal{X}$, the current partition $\mathcal{P}$ is refined with the neighbourhood $N(x)$. When the algorithm ends, all the parts are modules. Finding, at each step, a vertex x whose neighbourhood strictly refines the partition $\mathcal{P}$ is the usual barrier to linear time complexity. However, using the so-called *Hopcroft rule*, one gets a fairly simple solution that uses the neighbourhood of each vertex at most $\log n$ times.

Lemma 11. *Let* $\mathcal{P}$ *be a partition of the vertices of a graph* $G = (V, E)$ *and* x *be a vertex of some part* $\mathcal{X}$. *If* $\mathcal{P}$ *is stable with respect to* $N(y)$, $\forall y \notin \mathcal{X}$, *then* $\mathcal{X}$ *is a module of* G *and the partition* $\mathcal{Q} = Refine(\mathcal{P}, N(x))$ *is stable with respect to* $N(x')$, $\forall x' \in \mathcal{X}$.

The above lemma (which is a direct consequence of the definition of module) shows that using as pivots the vertices of all the parts of $\mathcal{P}$ but one, say $\mathcal{Z}$, plus one vertex z of $\mathcal{Z}$ is enough. For complexity issues, the avoided part $\mathcal{Z}$ has to be chosen as the largest part of $\mathcal{P}$. Similarly, once a part $\mathcal{X}$ has been split, the process continues recursively on the subgraph induced by $\mathcal{X}$ and the resulting largest subpart can be avoided (meaning that only one of its vertices has to be used as pivot). This *"avoid the largest part"* technique is known as *Hopcroft's rule*, and it was first proposed in the deterministic automata minimization algorithm [64].

Algorithm 2: Modular Partition

Input: A partition $\mathcal{P}$ of the vertex set V of a graph G
Output: The coarsest modular partition $\mathcal{Q}$ smaller than $\mathcal{P}$

```
begin
    Let Z be the largest part of P;
    Q ← P; K ← {Z}; L ← {X | X ≠ Z, X ∈ P};
1   while L ∪ K ≠ ∅ do
        if there exists X ∈ L then S ← X and L ← L \ {X};
        else
2           Let X be the first part K and x arbitrarily
            selected in X;
            S ← {x} and K ← K \ {X};
        foreach vertex x ∈ S do
            foreach part Y ≠ X such that N(x) ⊥ Y do
3               Replace in Q, Y by Y1 = Y ∩ N(x) and
                Y2 = Y \ N(x);
                Let Ymin (resp. Ymax) be the smallest part
                (resp. largest) among Y1 and Y2;
                if Y ∈ L then L ← L ∪ {Ymin, Ymax} \ {Y};
                else
                    L ← L ∪ {Ymin};
                    if Y ∈ K then Replace Y by Ymax in K;
                    else Add Ymax at the end of K;
end
```

To implement this rule, the parts are stored in two disjoint lists K and L. The neighbourhoods of all the vertices of parts belonging to L will be used to refine the partition. For the parts belonging to K, only the neighbourhood of one arbitrarily selected vertex is used. Since K is managed with an FIFO priority rule, this guarantees that the first part of the list, when extracted, is a module.

Theorem 9. *Let* $\mathcal{P}$ *be a partition of the vertices of a graph* $G = (V, E)$. *Algorithm 2 computes the coarsest modular partition for* G *and* $\mathcal{P}$ *in time* $O(n + m \log n)$.

The correctness of the algorithm follows from the next three invariant properties. The first invariant shows that a module contains in some part of the given partition cannot be split, while the third one guarantees that the algorithm outputs a modular partition.

(1) If M is a module of G contained in a part $\mathcal{X} \in \mathcal{P}$, then there exists a part $\mathcal{Y}$ of the current partition containing M.

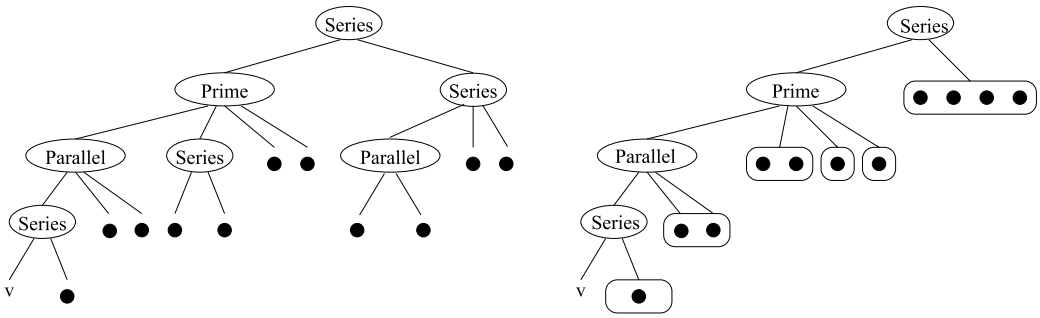

Fig. 9 – On the left, a modular decomposition tree $MD(G)$, and on the right, the modular partition $\mathcal{M}(G, v)$ with the corresponding spine between v and the root of $MD(G)$.

(2) If $L = \emptyset$, then the first part $\mathcal{Y}$ of K is a module.
(3) If the current partition contains a part $\mathcal{X}$ that is not a module, then there exists $\mathcal{Y} \in L \cup K$ different from $\mathcal{X}$ and containing a splitter y for $\mathcal{X}$.

Complexity issues: The main while loop (line 1) manages a set S of vertices whose neighbourhoods have to be used to refine the current partition. The set S is computed from the lists L and K. Since the current part containing a given vertex can be added to L only if its size is smaller than half of the size of the former part containing x, the neighbourhood of each vertex x is guaranteed to be visited at most $\log(|V|)$ times by the algorithm. Furthermore, when a vertex x of a part $\mathcal{X}$ extracted from K is used, neither x nor any of the vertices of $\mathcal{X}$ is used again. This yields to a $O(\sum_{x \in V} \log(|V|).|N(x)|)$ complexity, as claimed.

4.3. *Bibliographic notes*

As already mentioned, the use of partition refinement technique dates to 1971 for the deterministic automata minimization problem [64]. In 1987, Paigue and Tarjan used this technique again to solve three different problems: functional partition, coarsest relational partition problems and doubly lexicographic ordering of a boolean matrix. In the late 1990's, it has been used more systematically in the context of modular decomposition and transitive orientation yielding $O(n + m \log n)$ practical and simple algorithms (see e.g. [23,66]).

5. Recursive computation of the modular decomposition tree

In 1994, Ehrenfeucht et al. [2] proposed a quadratic algorithm for modular decomposition.[1] The principle of this algorithm, which we will call the *skeleton algorithm*, is the basis of a large number of the known subquadratic algorithms proposed in the late 1990's (see *e.g.* [22,23]), which could abusively be considered as a series of different implementations of the skeleton algorithm. The complexity of these implementations is respectively $O(n + m.\alpha(n, m))$ or $O(n + m)$ [22], and finally $O(n + m \log n)$ [23]. We describe the principle of the skeleton algorithm without considering the complexity issues. We then discuss the differences in the time complexity of the known algorithms.

5.1. *The skeleton algorithm*

Let us first mention that the skeleton algorithm computes a *non-reduced* form of the modular decomposition tree $MD(G)$: the resulting tree may contain some series (or parallel) node child of a series (or parallel) node. All the algorithms we describe in this section will do so. This does not impact the complexity issues as a single search of the tree is enough to reduce it in time $O(n)$. In the following, we will abusively denote $MD(G)$ the (non-reduced) decomposition tree returned by these algorithms.

The main idea developed by Ehrenfeucht et al. [2] is to first compute a "*spine*" of the modular decomposition tree $MD(G)$, and then to recursively compute the modular decomposition trees of some induced subgraphs which are eventually padded to the spine. More formally:

Definition 9. Let v be an arbitrary vertex of a graph $G = (V, E)$. The *v-modular partition* is the following modular partition:

$$\mathcal{M}(G, v) = \{v\} \cup \{\, M \mid M \text{ is a maximal module not containing } v\}.$$

We define $spine(G, v)$ as the modular decomposition tree $MD(G_{/\mathcal{M}(G,v)})$.

First we notice that $\mathcal{M}(G, v)$ is easy to compute.

Lemma 12. *The partition $\mathcal{M}(G, v)$ is the coarsest modular partition for G and $\mathcal{P} = \{N(v), v, \overline{N}(v)\}$ and can be computed in time $O(n + m \log n)$ (Fig. 9).*

Let us notice that any degenerate strong module (series or parallel) containing v will be represented in $spine(G, v)$ by a binary node. The purpose of test of Line 1 in Algorithm 3 is to correctly fixed those binary nodes. The correctness of Algorithm 3 is a consequence of the following properties:

Lemma 13 ([2]). *Let v be a vertex of a graph $G = (V, E)$ and $\mathcal{M}(G, v)$ be the associated modular partition. Then:*

(1) *Any non-trivial module of $G_{/\mathcal{M}(G,v)}$ contains v;*
(2) *A set $\mathcal{X} \subset \mathcal{M}(G, v)$ is a non-trivial strong module of $G_{/\mathcal{M}(G,v)}$ iff $\bigcup_{M \in \mathcal{X}} M$ is an ancestor of v in $MD(G)$;*
(3) *Any module not containing v is a subset of a part $M \in \mathcal{M}(G, v)$.*

[1] This algorithm is designed for 2-structures, a classical generalization of graphs.

Algorithm 3: Ehrenfeucht et al. [2]

Input: An arbitrary vertex v of $G = (V, E)$, $T = spine(G, v)$ and $\{T_X = MD(G[X]) \mid X \in \mathcal{M}(G, v)\}$
Output: The modular decomposition tree $MD(G)$
begin
  foreach *leaf* X *of* T **do**
    Let $T_X = MD(G[X])$ and $p(X)$ be X's father in T;
    Replace X by T_X in T;
1   **if** *the root* $r(T_X)$ *and* $p(X)$ *are both parallel or series* **then**
      Remove $r(T_X)$ and connect the children of $r(T_X)$ to $p(X)$
end

Computing $spine(G, v)$ is a difficult and technical task of the skeleton algorithm; indeed it is its main complexity bottleneck. The solution we present hereafter has been proposed in [2] and yields quadratic running time. Later on, Dahlhaus et al. [22] improved this step and obtained a subquadratic running time (see the discussion in Section 5.3).

5.2. *Computation of spine(G, v)*

Definition 10. A graph $G = (V, E)$ is *nested* if there exists a vertex $v \in V$ which is contained in all the non-trivial modules of G. Such a vertex is called an *inner* vertex of G.

As a direct consequence of Lemma 13, the quotient graph $G_{/\mathcal{M}(G,v)}$ is a nested graph with inner vertex v.

In order to compute the modules of $G_{/\mathcal{M}(G,v)}$ and $spine(G, v)$, Ehrenfeucht et al. [2] introduced an auxiliary *forcing digraph*, the arc set of which guarantees the existence of a directed path from any vertex u to any vertex $w \in m(u, v)$, the smallest module containing u and v. As v belongs to all the modules of $G_{/\mathcal{M}(G,v)}$, a simple search on the forcing graph will suffice to compute $spine(G, v)$.

Definition 11. [2] Let v be an arbitrary vertex of a graph $G = (V, E)$. The *forcing graph* $\mathcal{F}(G, v)$ is a directed graph whose vertex set is $V \setminus \{v\}$. The arc $\overrightarrow{xy}$ exists if y is a splitter for $\{x, v\}$ (Fig. 10).

In other words, if $\overrightarrow{xy}$ exists then y belongs to any module containing v and x.

Lemma 14 ([2]). *If X is the set of vertices that can be reached from vertex x in the forcing graph $\mathcal{F}(G, v)$, then $\{v\} \cup X = m(v, x)$.*

In the following we will only consider the graph $G_{/\mathcal{M}(G,v)}$ and its forcing graph $\mathcal{F}(G_{/\mathcal{M}(G,v)}, v)$. Applying Lemma 14 to $\mathcal{F}(G_{/\mathcal{M}(G,v)}, v)$, we obtain the following property.

Corollary 1 ([2]). *Let M_x be the module of $\mathcal{M}(G, v)$ containing the vertex x. If $\mathcal{X}$ is the set of modules that can be reached from M_x in $\mathcal{F}(G_{/\mathcal{M}(G,v)}, v)$, then $\bigcup_{M \in \mathcal{X}} M = m(v, x)$.*

We now consider the *block graph* $\mathcal{B}(G, v)$ of $\mathcal{F}(G_{/\mathcal{M}(G,v)}, v)$ (see [70]) whose vertices are the strongly connected components of $\mathcal{F}(G_{/\mathcal{M}(G,v)}, v)$, also called the *blocks* of (G, v). An arc of $\mathcal{B}(G, v)$ between the block B and B' exists if the vertices of B' can be reached in $\mathcal{F}(G_{/\mathcal{M}(G,v)}, v)$ from the vertices of B.

Lemma 15 ([2]). *The transitive reduction of the block graph $\mathcal{B}(G, v)$ is a chain.*

A set of vertices of a digraph is a *sink* if it has no out-neighbour. By Lemma 15, any sink set of $\mathcal{F}(\mathcal{M}(G, v))$ is the union of consecutive blocks containing the last one in the transitive reduction of $\mathcal{B}(G, v)$. Each sink set corresponds to a module of $G_{/\mathcal{M}(G,v)}$.

Corollary 2 ([2]). *Let v be a vertex of a graph $G = (V, E)$. A set M of vertices containing v is a module of $G_{/\mathcal{M}(G,v)}$ iff M is the union of $\{v\}$ and the modules of $\mathcal{M}(G, v)$ belonging to a sink set X of $\mathcal{B}(G, v)$.*

Thereby the forcing graph $\mathcal{F}(G_{/\mathcal{M}(G,v)}, v)$ describes the modules of $G_{/\mathcal{M}(G,v)}$ and the block graph $\mathcal{B}(G, v)$ allows us to compute $spine(G, v)$. Finally, $MD(G)$ is obtained recursively by following the lines of Lemma 13.

5.3. *Complexity issues*

Rather than detailing the complexity analysis, we point out the differences between the original skeleton algorithm presented in [2] and its later versions improved in [22]. The interested reader should access the original papers for details. As already mentioned, a quadratic time complexity analysis is proposed in [2]. The main bottlenecks are the computation of the partition $\mathcal{M}(G, v)$ and the construction of $MD(G_{/\mathcal{M}(G,v)})$.

Two new versions of the skeleton algorithm proposed by Dahlhaus et al. [22], respectively run in $O(n + m.\alpha(n, m))$ time and in linear time. To improve the time complexity, the authors of [22] borrowed from [47] the idea to first recursively compute the modular decomposition trees of the subgraphs induced by $N(v)$ and by $\overline{N}(v)$. It follows from the next lemma that $\mathcal{M}(G, v)$ is easy to retrieve from those trees.

Lemma 16. *If $\mathcal{X}$ is a module of $\mathcal{M}(G, v)$, then $\mathcal{X}$ is either a module of $G[N(v)]$ or a module of $G[\overline{N}(v)]$.*

As in [2], the technique used to compute $spine(G, v)$ relies on a forcing digraph. Recall that the vertices of $\mathcal{F}(\mathcal{M}(G, v))$ are the modules of G (indeed the modules of $\mathcal{M}(G, v)$), which turns out to be a too strong condition for time complexity issues. In [22], the forcing digraph is rather defined with the help of an equivalence relation. The idea is that each equivalence class gathers vertices of $N(v)$ or of $\overline{N}(v)$ which appear in a set of sibling modules of some ancestor node of v in $MD(G)$ (or $spine(G, v)$). The partition defined by the equivalence classes is a coarser partition than $\mathcal{M}(G, v)$.

The final trick is that, given $MD(G[N(v)])$ and $MD(G[\overline{N}(v)])$, the computation of $\mathcal{M}(G, v)$, $spine(G, v)$ and finally $MD(G)$ has to be done in time linear in the number of *active edges*, *i.e.* the edges incident to v and the edges linking vertices of $N(v)$ and $\overline{N}(v)$. The $\alpha(n, m)$ factor in the first version of the skeleton algorithm presented in [22] is due to the use of some *union-find* data-structures required to update the current tree. A clever time complexity analysis yields linear time if a careful preprocessing step is used to fix the recursion tree.

[2] The definition proposed here differs slightly from the original one of [2]. This modification simplifies the relationships with the results of [22].

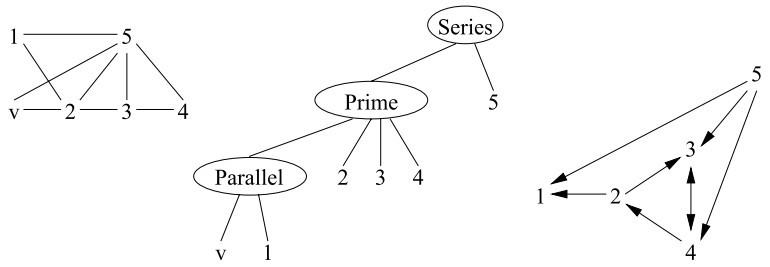

Fig. 10 – A nested graph $G = (V, E)$ together with its modular decomposition tree $MD(G)$ and on its right the forcing graph $\mathcal{F}(G, v)$. The strongly connected components of $\mathcal{F}(G, v)$ are {1}, {2, 3, 4}, {5}. Any module of G containing 3 and v also contains {1, 2, 4}, the vertices that can be reached from vertex 3 in $\mathcal{F}(G, v)$.

5.4. *Bibliographic notes*

Let us mention that the problem of finding a simple linear time algorithm for the modular decomposition is presented in [23] or [51] as an open problem. In his book [51], Spinrad wrote (p. 149):

"*I hope and believe that in a number of years the linear algorithm can be simplified as well*"

Based on partition refinement techniques, a simplified $O(n + m \log n)$ version of the skeleton algorithm has been developed in [23].

6. Factoring permutation algorithm

In his PhD thesis, Capelle [34] proved that computing the modular decomposition tree of a graph and computing a *factoring permutation* (see Definition 4 and Figures 3, 12) are two equivalent tasks, as one can be retrieved from each another in linear time [27]. It follows that computing the modular decomposition of a graph can be divided into two different steps: (1) computation of a factoring permutation; (2) computation of the modular decomposition tree given the factoring permutation. The main interest of such a strategy is to obtain an algorithm that avoids the auxiliary data-structures needed to compute the *union-find* and *least common ancestor* operations, as used in [22], for example. Moreover, in some recent applications (e.g. comparative genomics [14,71, 72]), the given data is not the graph or the partitive family but rather a factoring permutation. This concept turns out to be of interest by itself.

As noticed by Capelle [34], this strategy was already used in few cases such as the computation of the modular decomposition tree of chordal graph [30] and the block tree of inheritance graphs [32]. In [24,67], a partition refinement algorithm is proposed to compute a factoring permutation of a graph in time $O(n + m \log n)$. Restricted to cographs, the complexity can be improved down to linear time [48].

We will first revisit Algorithm 1 of [67] and show how it can be adapted to compute a factoring permutation in time $O(n + m \log n)$. This algorithm has to be compared to McConnell and Spinrad's implementation [23] of Ehrenfeucht et al.'s algorithm. The main differences are that the modular decomposition tree is never built and the relative order between the different parts of the partition is important.

There exist several linear time algorithms that, given a factoring permutation of a graph, compute its modular decomposition tree. A recent one is proposed in [15,73]. We describe the principle of the first one due to Capelle et al. [27].

6.1. *Computing a factoring permutation*

An ordered partition $\mathcal{P} = [\mathcal{X}_1, \ldots, \mathcal{X}_k]$ of a set $\mathcal{E}$ defines a partial order on $\mathcal{E}$, the maximal antichains of which are exactly the parts of $\mathcal{P}$. In other words, we have $x_i <_{\mathcal{P}} x_j$ iff $x_i \in \mathcal{X}_i$, $x_j \in \mathcal{X}_j$ and $i < j$. Thereby, refining an ordered partition could be understood as computing an extension of the corresponding partial order.

We will abusively write $x <_{\mathcal{P}} M$, for $x \in \mathcal{E}$ and $M \subset \mathcal{E}$, if $x <_{\mathcal{P}} y$ for all $y \in M$. To prove the correctness of the algorithm, we need to generalize the definition of interval of permutations to ordered partitions.

Definition 12. Let $\mathcal{P}$ be an ordered partition of a set $\mathcal{E}$. A subset $S \subseteq \mathcal{E}$ is an *interval* of $\mathcal{P}$ iff there are two parts $\mathcal{L} \in \mathcal{P}$ and $\mathcal{R} \in \mathcal{P}$ (not necessarily distinct) intersecting S such that for any part $\mathcal{X}$:

- if $\mathcal{L} <_{\mathcal{P}} \mathcal{X} <_{\mathcal{P}} \mathcal{R}$, then $\mathcal{X} \subset S$;
- if $\mathcal{X} <_{\mathcal{P}} \mathcal{L}$ or $\mathcal{R} <_{\mathcal{P}} \mathcal{X}$, then $\mathcal{X} \cap S = \emptyset$.

To compute a factoring permutation, the main steps of the algorithm we present are: (1) computation of an ordered partition that is a modular partition $\mathcal{M}(G, v)$ such that the strong modules containing a vertex v are intervals of $\mathcal{M}(G, v)$; and (2) recursive computation of a factoring permutation of each of the subgraphs induced by a module $M \in \mathcal{M}(G, v)$.

Theorem 10. *Algorithm 4 computes in time $O(n + m \log n)$ a factoring permutation of a graph $G = (V, E)$.*

Proof. Using Lemma 12, $\mathcal{M}(G, v)$ can be computed in $O(n + m \log n)$. By Lemma 13, any module not containing v is a subset of some module of $\mathcal{M}(G, v)$. It thereby suffices to prove that the following invariant is satisfied by Algorithm 4 (see Fig. 11):

Π = any strong module containing v is an interval of the current partition.

The property Π is obviously satisfied by the initial partition $[\overline{N}(v), \{v\}, N(v)]$. Assume by induction that Π holds before the current partition $\mathcal{P}$ is refined by $N(x)$ for some vertex x. Let M be a module containing v and $\mathcal{X}$ be a part of $\mathcal{P}$ such that $\mathcal{X} \perp N(x)$. There are two distinct cases:

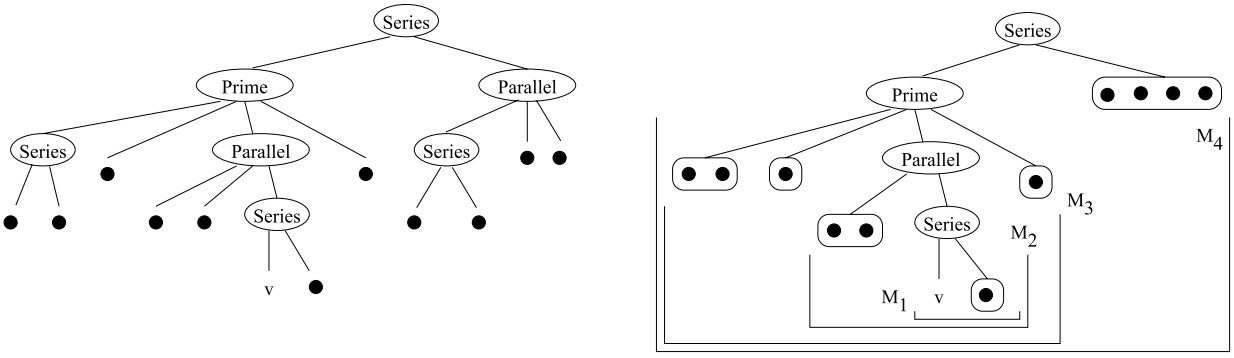

Fig. 11 – Layout of the modular decomposition tree $MD(G)$ such that the neighbours of v are placed on the right of v and the non-neighbours on the left. The right tree enlights the modules of $\mathcal{M}(G, v)$ and the strong modules M_1, M_2, M_3 and M_4 containing v. Algorithm 4 first computes the partition $\mathcal{M}(G, v)$ and then recursively solves the problem on each module of $\mathcal{M}(G, v)$.

Algorithm 4: *Factoring-permutation(G, v)*

Input: A graph $G = (V, E)$ and a vertex $v \in V$
Output: A factoring permutation of G
begin
  Let $\mathcal{P} = [\overline{N}(v), \{v\}, N(v)]$ be an ordered partition;
  Apply Algorithm 2 with the following refinement rule;
  Let x be the current pivot vertex and $\mathcal{Y}$ a part such that $N(x) \perp \mathcal{Y}$;
  if $x \leqslant_{\mathcal{P}} v \leqslant_{\mathcal{P}} \mathcal{Y}$ *or* $\mathcal{Y} \leqslant_{\mathcal{P}} v \leqslant_{\mathcal{P}} x$ **then**
    Substitute $\mathcal{Y}$ by $[\mathcal{Y} \cap \overline{N}(x), \mathcal{Y} \cap N(x)]$;
  else
    Substitute $\mathcal{Y}$ by $[\mathcal{Y} \cap N(x), \mathcal{Y} \cap \overline{N}(x)]$;
  foreach *part* $\mathcal{X} \in \mathcal{M}(G, v)$, *such that* $|\mathcal{X}| > 1$ **do**
    Let x be the last vertex of $\mathcal{X}$ used as pivot;
    $\mathcal{P}_{\mathcal{X}} \leftarrow$ Factoring-permutation($G[\mathcal{X}], x$);
    Substitute $\mathcal{X}$ by $\mathcal{P}_{\mathcal{X}}$;
end

- $x \notin M$: no vertex y of $\mathcal{X} \cap N(x)$ belongs to M, otherwise x would be a splitter for v and y;
- $x \in M$: if $\mathcal{X} \subset N(v)$, then any vertex $y \in \mathcal{X} \cap \overline{N}(x)$ belongs to M, otherwise y would be a splitter for x and v. Similarly, if $\mathcal{X} \subset \overline{N}(v)$, then any vertex $y \in \mathcal{X} \cap N(x)$ belongs to M.

It follows that $\mathcal{P}' = \text{Refine}(\mathcal{P}, N(x))$ also satisfies the invariant Π. The complexity analysis is similar to the analysis of Algorithm 2. □

6.2. *The case of cographs*

The natural question is how to get rid of the $\log n$ factor in the complexity of Algorithm 4. Restricting the problem to cographs (or totally decomposable graphs — see Section 2.5) gives some ideas. The reader should keep in mind that the $\log n$ factor corresponds to the number of times the neighbourhood of a vertex can be used to refine the partition. So, a linear time algorithm should use each vertex as a pivot a constant number of times.

The linear time cograph recognition algorithm proposed in [48] computes a factoring permutation as a preliminary step. It roughly proceeds as follows. It uses at most one vertex per partition part to refine the ordered partition $[\overline{N}(v), \{v\}, N(v)]$. Assuming that the input graph is a cograph, when none of the parts of the current partition is free of a pivot, it can be proved that one of the two non-singleton parts closest to v in the current partition, say $\mathcal{X}$, can be refined into $[\overline{N}(x) \cap \mathcal{X}, \{x\}, N(x) \cap \mathcal{X}]$ (x being the used pivot of $\mathcal{X}$). This step creates at least one new part free of a pivot and thereby relaunches the refining process.

6.3. *From factoring permutation to modular decomposition tree*

As already noticed, a natural idea to compute the modular decomposition tree is to compute for each pair x, y of vertices the set of *splitter* $S(x, y)$. Unfortunately, a linear time algorithm could not afford the computation of all these $O(n^2)$ sets. But if one has in hand a factoring permutation σ, it is then sufficient to consider the pairs of consecutive vertices in σ. Indeed, Capelle et al.'s algorithm [27] only computes for each pair of vertices $x = \sigma(i)$ and $y = \sigma(i + 1)$ ($i \in [1, n - 1]$) the leftmost and the rightmost (in σ) splitter of x and y. These two splitters define two intervals of σ, which are both contained in $m(x, y)$, the smallest module containing both x and y:

- the *left fracture* $F_l(x, y) = [z, x]$ if z is the leftmost splitter of $\{x, y\}$ in $[\sigma(1), y]$ (if any);
- the *right fracture* $F_d(x, y) = [y, z]$ if z is the rightmost splitter of $\{x, y\}$ in $[x, \sigma(n)]$ (if any).

The set of fractures (left and right) defines a parenthesis system. Forgetting the initial pairing of the parenthesis, this system naturally yields a tree, called the *fracture tree* and denoted $FT(G)$ (see Fig. 12). The fracture tree is actually a good estimation of the $MD(G)$ (see Lemma 17) which can be computed in linear time by two traversals of σ: the first traversal computes the fractures, the second builds the tree.

Lemma 17 ([27]). *Let σ be a factoring permutation of a graph G and M be a strong module of G. If M is a prime node of $MD(G)$ and if the father of M is a degenerate, then there exists a node N of the fracture tree $FT(G)$ such that M is the set of leaves of the subtree of $FT(G)$ rooted at N.*

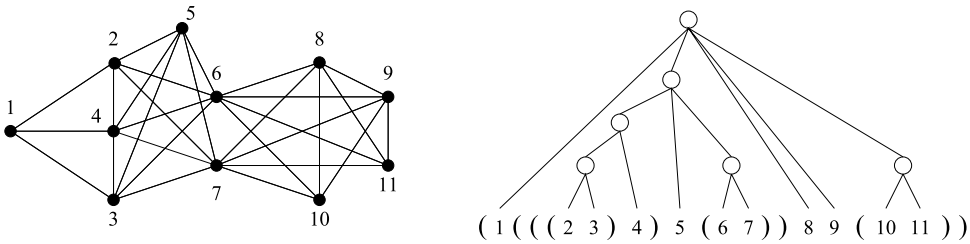

Fig. 12 – A graph $G = (V, E)$ for which $\sigma = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11$ is a factoring permutation (see Definition 4). The right fracture of $(3, 4)$ does not exist but $Fg(3, 4) = [2, 3]$. We also have $Fd(1, 2) = [2, 7] = Fg(7, 8)$.

For example, in Fig. 12, any strong module but $M = \{8, 9, 10, 11\}$ is represented by some node of $FT(G)$. Let us notice that the above lemma does not implies that the strong module $\{2, 3, 4\}$ has a corresponding node in $FT(G)$.

Henceforth, to compute $MD(G)$, the fracture tree $FT(G)$ has to be cleaned. To that aim, Capelle et al. [27] use four extra traversals of the factoring permutation. The first one identifies the strong modules represented by some nodes of $FT(G)$; the second finds the dummy nodes of $FT(G)$; the third searches for strong modules that are merged in a single node of $FT(G)$; and the last one removes the nodes of $FT(G)$ that do not represent strong modules. The complexity of each of these four traversals is linear in the size of G, $O(n + m)$.

6.4. *Bibliographic notes*

An attempt to generalize to arbitrary graphs the linear time algorithm which computes a factoring permutation of a cograph has been proposed in [74]. Unfortunately, the algorithm of [74] contains a flaw. The recent linear time modular decomposition algorithm presented in [21] mixes the ideas from the factoring permutation algorithms and the skeleton algorithm. It generalizes the ordered partition refining technique to tree partition and avoids union-find or least-common ancestor data-structures. In that sense this new algorithm may be considered as a positive answer to Spinrad's comment (see Section 5.4).

7. Three novel applications of the modular decomposition

As mentioned in the Introduction, modular decomposition is used in a number of algorithmic graph theory applications and more generally applies to various discrete structures (see [7]). We conclude this survey with the presentation of three novel applications which are good witnesses of the use of modular decomposition. The first one is a pattern matching problem which is closely related to the concept of factoring permutations. The second one provides an example of dynamic programming on the modular decomposition tree in the context of comparative genomic. Finally, we list a series of parameterized problems for which module based data-reduction rules lead to polynomial size kernels.

7.1. *Pattern matching — Common intervals of two permutations*

Motivated by a series of genetic algorithms for sequencing problems, e.g. the TSP, Uno and Yagiura [14] formalized the concept of the *common interval* of two permutations. As we will see in the next subsection, in the comparative genomic context, common intervals reveal conserved structures in chromosomal material.

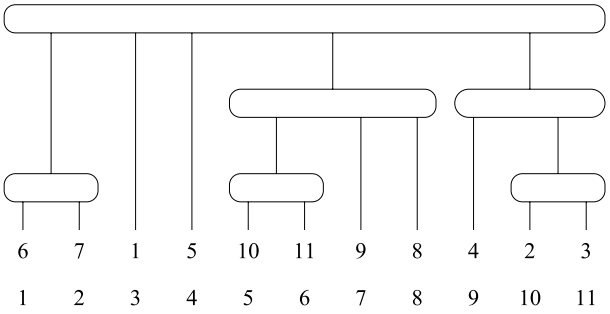

Fig. 13 – The strong interval tree of two permutations. We remark that $\{9, 10, 11\}$ and $\{8, 9\}$ are also a common interval, but they are not strong as they overlap.

Definition 13. A set S of elements is a *common interval* of a set of permutations Σ if, in each permutation $\sigma \in \Sigma$, the elements of S form an interval of σ (see Section 2.2 for the definition of an interval).

It is fairly easy to observe that the family $\mathcal{I}$ of common intervals of two permutations is a weakly partitive family (see Definition 1) and thus all the results from the theory presented in Section 2.1 apply. In particular, the set of strong common intervals is organized into a tree, namely the *strong interval tree* (Fig. 13).

Despite the existence of the (weakly) partitive set theory for more than 30 years, the natural concept of interval substitution and decomposition appeared only very recently in the context of the combinatorial study of permutations (see *e.g.* [75,76]). Atkinson and Stitt [75] (re)discovered the concept of substitution under the name of *wreath product*. In 2005, Albert and Atkinson showed that, if the number of *simple* (*i.e.* prime) permutations in a pattern restricted class of permutations is finite, the class has an algebraic generating function and is defined by a finite set of restrictions. More recently, Bouvel et al. [46,77] used the strong interval tree to solve the longest common pattern problem between two permutations.

Uno and Yagiura [14] proposed the first linear time algorithm to enumerate the common intervals of two permutations. More precisely, it runs in $O(n + K)$ time, where K is

the number of those common intervals (which is possibly quadratic). Alternative algorithms have been recently proposed [15,72]. We sketch Uno and Yagiura's algorithm and discuss how it can be generalized to compute the modules of a graph when a factoring permutation is given.

Without loss of generality, we will consider the problem of computing the common intervals of a permutation σ and the identity permutation $\mathbb{I}_n$. To identify the common intervals of a permutation σ and $\mathbb{I}_n$, the algorithm traverses σ only once. We denote by $[i, j]$ the interval of σ composed by the elements whose indices are between i and j in σ: i.e. $[i, j] = \{x \mid i \leqslant \sigma(x) \leqslant j\}$. An element $x \notin [i, j]$ is a *splitter* of the interval $[i, j]$ if there exist $y \in [i, j]$ and $z \in [i, j]$ such that $y < x < z$. By $s([i, j])$ we denote the number of *splitters* of the interval $[i, j]$. The algorithm uses a list `Potentiel` to filter and extract σ the common intervals of σ and $\mathbb{I}_n$. An element i belongs to the list `Potentiel` as long as it may be the right boundary of a common interval. The step i consists in removing those elements which we know cannot be the left boundary of a common containing. This filtering can be done efficiently by computing $s([i, j])$ (see Lemmas 18 and 19).

Algorithm 5: Uno and Yagiura's algorithm [14]

Input: A permutation σ
Output: The set of intervals common to σ and the identity permutation $\mathbb{I}_n$
begin
  Let `Potentiel` be an empty list;
  for $i = n$ *downto* 1 **do**
    (Filter) Remove from `Potentiel` the boundaries r s.t. $\forall j \leqslant i$, $[j, r]$ is not a **common interval** of σ and $\mathbb{I}_n$;
    (Extraction) Search `Potentiel` to find the boundaries r s.t. $[i, r]$ is a **common interval** of σ and $\mathbb{I}_n$ and output those intervals $[i, r]$;
    (Addition) Add i to `Potentiel`;
end

The following properties are fundamental in the correctness of the algorithm.

Lemma 18 ([14]). *An interval $[i, j]$ of σ is a common interval of σ and $\mathbb{I}_n$ iff $s([i, j]) = 0$.*

Lemma 19 ([14,78]). *If $s([i, j]) > s([i, j + 1])$, then there does not exist $r < i$ such that $[r, j]$ is a common interval of σ and $\mathbb{I}_n$.*

The second lemma above means that if $s([i, j]) > s([i, j + 1])$ then the vertex $\sigma^{-1}(j + 1)$ is a splitter of $[i, j]$. Thereby any common interval containing $[i, j]$ as a subset has to extend up to $\sigma^{-1}(j + 1)$.

Application to factoring permutations of a graph. The most striking link between common intervals and modules of graphs is observed on permutation graphs (see Lemma 20). *Permutation graphs* are defined as the intersection graphs of a set of segments between two parallel lines (see [9,10] for example). It follows that the vertices of a permutation graph $G = (V, E)$ can be numbered from 1 to n such that there exists a permutation σ of $[1, n]$ such that vertex numbered i is adjacent to vertex numbered j iff $i < j$ and $\sigma(j) < \sigma(i)$. The

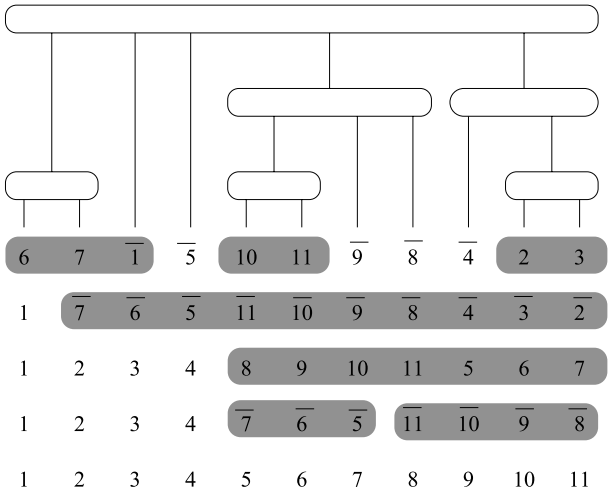

Fig. 14 – A perfect scenario of length 7.

permutations σ and $\mathbb{I}_n$ form the *realizer* of G. As first observed by de Montgolfier, any permutation belonging to a realizer of a permutation graph is a factorizing permutation of that graph. It follows from Lemma 1 that:

Lemma 20 ([79]). *Let $G = (V, E)$ be a permutation graph and $(\mathbb{I}_n, \sigma)$ be its realizer. A set of vertices M is a strong module iff M is a strong common interval of $\mathbb{I}_n$ and σ.*

The permutation graph corresponding to the permutations depicted in Fig. 14 is the graph G of Fig. 3. Notice that the strong interval tree of these two permutations is isomorphic to the modular decomposition tree of G.

It follows from Lemma 20 that, applied to the realizer of a permutation graph, Algorithm 5 computes its strong modules. Though some extra work is required to obtained the modular decomposition tree, the complexity remains linear in time. Moreover, as shown in [78], Uno and Yagiura's algorithm can directly be adapted to compute, given a factoring permutation, the strong modules of a graph. The number $s([i, j])$ becomes the number of splitters (in the sense of the modular decomposition, see Section 2.3) of the vertices contained in the interval $[i, j]$ of the factoring permutation. Now notice that Algorithm 5 does not only output the strong common intervals. In order to restrict the enumeration to strong modules, a slight modification is required. A first traversal computes the strong right modules (i.e. the modules that are intervals of σ and which are not overlapped on their right boundary by any other module). Then a second traversal can detect those modules which are overlapped on the left boundary.

7.2. *Comparative genomic — Perfect sorting by reversals*

A *reversal* in a permutation σ consists in reversing the order of the elements of an interval of σ. When dealing with *signed* permutations (whose elements are positive or negative), a reversal also flips the sign of the element of the reserved interval. Given two (signed) permutations σ and τ,

the problem of *sorting by reversals* asks for a series of reversals (a *scenario*) to transform σ into τ.

Sorting by reversals is used in comparative genomic studies to measure the evolutionary distance between the genomes of two chromosomes, modelled as signed permutations [71]. When comparing two genomic sequences, it can be assumed that the intervals having the same gene content are likely to have been present in their common ancestor and may witness to some functionally interacting proteins. Such a conserved genomic structure in the signed permutation model corresponds to common intervals. So to guess an evolutionary scenario between two genomic sequences represented by signed permutations σ and τ, one could ask for the smallest *perfect scenario*, which is a series of reversals that preserves any common interval of σ and τ. For further details on this topic, the reader could refer to [45,71].

As mentioned in the previous subsection, the set of common intervals of two permutations (signed or not) defines a weakly partitive family. It follows that one can distinguish prime from degenerate strong common intervals. As shown by the following lemma, we can read on the strong interval tree which are the perfect scenarios.

Lemma 21 ([45]). *A reversal scenario for two signed permutations σ and τ is perfect iff any reversed interval is either a prime common interval of σ and τ, or the union of strong common intervals which form a subset of the children of a prime common interval.*

It follows from the previous lemma that the strong interval tree is useful to compute minimum perfect scenarios. Indeed, with some extra technical properties to deal with the signs it can be shown that a simple dynamic programming algorithm on the strong interval tree solves the problem in time $O(2^k \times n\sqrt{n \log n})$, where k is the maximum number of prime nodes which are children of the same prime node. In practice, the parameter k remains very small [80]: e.g. when comparing the chromosome X of the mouse and the rat, we have $k = 0$ [81].

7.3. *Parameterized complexity and kernel reductions — Cluster editing*

The design of parameterized algorithms is, among others, one of the modern techniques to cope with NP-hard problems. A problem Π is *fixed parameter tractable* (FPT) with respect to parameter k if it can be solved in time $f(k).n^{O(1)}$ where n is the input size. The idea behind parameterized algorithms is to find a parameter k, as small as possible, which controls the combinatorial explosion. Many algorithmic techniques have been developed in the context of fixed parameter complexity, among which is *kernelization*. A parameterized problem (Π, k) admits a *polynomial kernel* if there is a polynomial time algorithm (a set of *reduction rules*) that reduces the input instance to an instance whose size is bounded by a polynomial $p(k)$ depending only in k, while preserving the output. The classical example of a parameterized problem having a polynomial kernel is the problem VERTEX COVER parameterized by k, the solution size, which has a $2k$ vertex kernel. For textbooks on this topic, the reader should refer to [82–84].

Recently, modular decomposition appeared in kernalization algorithms for a series of parameterized problems among which are CLUSTER EDITING [83], BICLUSTER EDITING [85], FAST (feedback arc set in tournament) [86], CLOSEST 3-LEAF POWER [87], and FLIP CONSENSUS TREE [88]. We discuss the CLUSTER EDITING problem. Concerning the others, the reader should refer to the original papers.

The parameterized CLUSTER EDITING problem asks whether the edge set of an input graph G can be modified by at most k modifications (deletions or insertions) such that the resulting graph H is a disjoint union of cliques (e.g. clusters). This problem is NP-complete but can be solved in time $O^*(3^k)$ by a simple bounded search tree algorithm [89], which iteratively branches on at most kP_3's. Recent papers [90,91] showed the existence of a linear kernel (the best bound is $4k$). The reduction rules used for these linear kernels are crown rules involving modules. For the sake of simplicity we only present the two basic reduction rules, which leads to a quadratic vertex kernel.

Lemma 22. *Let $G = (V, E)$ be a graph. A quadratic vertex kernel for the* CLUSTER EDITING *problem is obtained by the following reduction rules:*

(1) *Remove from G the connected components which are cliques.*
(2) *If G contains a clique module C of size at least $k+1$, then remove from $|C| - k - 1$ vertex from C.*

It is clear that these rules can be applied in linear time using modular decomposition algorithms. The proof idea works as follows. The first rule is obviously safe. Concerning the second rule, simply observe that to disconnect a clique module of size $k + 1$ from the rest of the graph, at least $k + 1$ edge deletions are required. Now, assuming that G is a positive instance, each cluster of the resulting graph H can be bipartitioned into the vertices non-incident to a modified edge and the other vertices (the *affected vertices*). Finally, k edge modifications can create at most $2k$ clusters, and the total number of affected vertices is bounded by $2k$. This shows that the number of vertices in the reduced graph H is at most $2k^2 + 4k$.

The BICLUSTER EDITING problem edits the edge set of a graph to obtain a disjoint union of complete bipartite graphs. Instead of considering clique modules, we need to consider independent set modules [85]. The proof is then slightly more complicated and relies on a careful analysis of the modification of the modular decomposition under edge insertion or deletion. In the case of FAST, similar rules involving transitive modules also yield a quadratic kernel bound. Note that, for these two problems, linear kernels can be obtained with more sophisticated reduction rules [92,93].

8. Conclusions and perspectives

An important remaining open problem is the proposal of a simple linear time certifying algorithm for modular decomposition. In fact the algorithms described here produce a labelled tree that can be checked in linear time if they are decomposition trees. But for certifying that some decomposition tree is the modular decomposition one must certify all node labels. The bottleneck is the certification of prime nodes.

We have presented above the principles of a fully dynamic algorithm for modular decomposition of cographs; these can be also suitable for permutation graphs and interval graphs using their geometric representation [58,59]. Fully dynamic modular decomposition for the general case is still an open problem.

For some applications one wants to extend the notion of a module to some notion of an *approximative* module, for which we want to extend the notion having the same behaviour outside the module. Several attempts have already been considered [35]. The main difficulty is to find an interesting extension of a module that is polynomially tractable, since many of the natural extensions yield NP-complete problems [94].

REFERENCES

[1] T. Gallai, Transitiv orientierbare graphen, Acta Mathematica Hungarica 18 (1967) 25–66.

[2] A. Ehrenfeucht, H.N. Gabow, R.M. McConnell, S.L. Sullivan, An $O(n^2)$ divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs, Journal of Algorithms 16 (1994) 283–294.

[3] R.H. Möhring, Algorithmic aspect of the substitution decomposition in optimization over relations, set systems and boolean functions, Annals of Operations Research 4 (1985) 195–225.

[4] A. Blass, Graphs with unique maximal clumpings, Journal of Graph Theory 2 (1978) 19–24.

[5] R.H. Möhring, Algorithmic aspect of comparability graphs and interval graphs, Graphs and orders (1985) 42–101.

[6] M. Habib, M.-C. Maurer, On the *X*-join decomposition of undirected graphs, Discrete Applied Mathematics 1 (1979) 201–207.

[7] R.H. Möhring, F.J. Radermacher, Substitution decomposition for discrete structures and connections with combinatorial optimization, Annals of Discrete Mathematics 19 (1984) 257–356.

[8] L. Lovász, A characterization of perfect graphs, Journal of Combinatorial Theory Series B 13 (1972) 95–98.

[9] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, 1980.

[10] A. Brandstädt, VB. Le, J. Spinrad, Graph classes: A survey, in: SIAM Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, 1999.

[11] M. Chein, M. Habib, M.-C. Maurer, Partitive hypergraphs, Discrete Mathematics 37 (1981) 35–50.

[12] W.H. Cunnigham, J. Edmonds, A combinatorial decomposition theory, Canadian Journal of Mathematics 32 (3) (1980) 734–765.

[13] A. Ehrenfeucht, T. Harju, G. Rozenberg, The Theory of 2-Structures, World Scientific, 1999.

[14] T. Uno, M. Yagiura, Fast algorithms to enumerate all common intervals of two permutations, Algorithmica 26 (2) (2000) 290–309.

[15] A. Bergeron, C. Chauve, F. de Montgolfier, M. Raffinot, Computing common intervals of *K* permutations, with applications to modular decomposition of graphs, SIAM Journal on Discrete Mathematics 22 (3) (2008) 1022–1039.

[16] B.M. Bui-Xuan, Tree-representation of set families in graph decompositions and efficient algorithms, Ph.D. Thesis, Univ. de Montpellier II, 2008.

[17] D.D. Cowan, L.O. James, R.G. Stanton, Graph decomposition for undirected graphs, in: S-E Conference on Combinatorics, Graph Theory and Computing, Utilitas Mathematica (1972) 281–290.

[18] A. Cournier, M. Habib, A new linear algorithm of modular decomposition, in: Trees in Algebra and Programming, CAAP, in: Lecture Notes in Computer Science, vol. 787, 1994, pp. 68–84.

[19] R.M. McConnell, J.P. Spinrad, Linear-time modular decomposition and efficient transitive orientation of comparability graphs, in: Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, 1994, pp. 536–545.

[20] R.M. McConnell, J.P. Spinrad, Modular decomposition and transitive orientation, Discrete Mathematics 201 (1999) 189–241.

[21] M. Tedder, D. Corneil, M. Habib, C. Paul, Simpler linear-time modular decomposition via recursive factorizing permutations, in: International Colloquium on Automata, Languages and Programming, ICALP, in: Lecture Notes in Computer Science, vol. 5125 (1), 2008, pp. 634–645.

[22] E. Dahlhaus, J. Gustedt, R.M. McConnell, Efficient and practical algorithm for sequential modular decomposition algorithm, Journal of Algorithms 41 (2) (2001) 360–387.

[23] R.M. McConnell, J.P. Spinrad, Ordered vertex partitioning, Discrete Mathematics and Theoretical Computer Science 4 (2000) 45–60.

[24] M. Habib, C. Paul, L. Viennot, Partition refinement: An interesting algorithmic tool kit, International Journal of Foundation of Computer Science 10 (2) (1999) 147–170.

[25] D.G. Corneil, Y. Perl, L.K. Stewart, A linear time recognition algorithm for cographs, SIAM Journal on Computing 14 (4) (1985) 926–934.

[26] R. Paigue, R.E. Tarjan, Three partition refinement algorithms, SIAM Journal on Computing 16 (6) (1987) 973–989.

[27] C. Capelle, M. Habib, F. de Montgolfier, Graph decompositions and factorizing permutations, Discrete Mathematics and Theoretical Computer Science 5 (2002) 55–70.

[28] J. Edmonds, R. Giles, A min–max relation for submodular functions on graphs, Annals of Discrete Mathematics 1 (1977) 185–204.

[29] B.-M. Bui-Xuan, M. Habib, A representation theorem for union-difference families and application, in: Latin American Symposium on Theoretical Informatics, LATIN, in: Lecture Notes in Computer Science, vol. 4957, 2008, pp. 492–503.

[30] W.-L. Hsu, T.-H. Ma, Substitution decomposition on chordal graphs and applications, in: International Symposium on Algorithms, ISA, in: Lecture Notes in Computer Science, vol. 557, 1991, pp. 52–60.

[31] W.-L. Hsu, A simple test for interval graphs, in: 3rd International Symposium on Algorithms and Computation, ISAAC, in: Lecture Notes in Computer Science, vol. 557, 1992, pp. 459–468.

[32] M. Habib, M. Huchard, J.S. Spinrad, A linear algorithm to decompose inheritance graphs into modules, Algorithmica 13 (1995) 573–591.

[33] C. Capelle, M. Habib, Graph decompositions and factorizing permutations, in: Israel Symposium on Theory of Computing and Systems – ISTCS, 1997, pp. 132–143.

[34] C. Capelle, Décomposition de graphes et permutations factorisantes, Ph.D. Thesis, Univ. de Montpellier II, 1997.

[35] B-M. Bui-Xuan, M. Habib, V. Limouzy, F. de Montgolfier, Algorithmic aspects of a general modular decomposition theory, Discrete Applied Mathematics 157 (9) (2009) 1993–2009.

[36] B. Courcelle, J. Engelfriet, G. Rozenberg, Handle rewriting graph grammars, Journal of Computer and System Science 46 (1993) 218–270.

[37] A. Ehrenfeucht, G. Rozenberg, Primitivity is hereditary for 2-structures, Theoretical Computer Science 70 (3) (1990) 343–358.

[38] J. Schmerl, W. Trotter, Critically indecomposable partially ordered sets, graphs, tournaments and other binary relational structures, Discrete Mathematics 113 (1–3) (1993) 191–205.
[39] A. Cournier, P. Ille, Minimal indecomposable graphs, Discrete Mathematics 183 (1998) 61–80.
[40] B. Jamison, S. Olariu, P-components and the homogeneous decomposition of graphs, SIAM Journal on Discrete Mathematics 8 (3) (1995) 448–463.
[41] B. Jamison, S. Olariu, Recognizing P_4-sparse graphs in linear time, SIAM Journal on Discrete Mathematics 21 (1992) 381–406.
[42] B. Jamison, S. Olariu, A tree representation of P_4-sparse graphs, Discrete Applied Mathematics 35 (1992) 115–129.
[43] D.P. Sumner, Graphs indecomposable with respect to the X-join, Discrete Mathematics 6 (1973) 281–298.
[44] D.G. Corneil, H. Lerchs, L.K. Stewart-Burlingham, Complement reducible graphs, Discrete Applied Mathematics 3 (1) (1981) 163–174.
[45] A. Bergeron, S. Bérard, C. Chauve, C. Paul, Sorting by reversal is not always difficult, in: Workshop on Algorithm for Bio-Informatics, WABI, in: Lecture Notes in Computer Science, vol. 3692, 2005, pp. 228–238.
[46] M. Bouvel, D. Rossin, S. Vialette, Longest common separable pattern among permutations, in: Annual Symposium on Combinatorial Pattern Matching, CPM, in: Lecture Notes in Computer Science, vol. 4580, 2007, pp. 316–327.
[47] E. Dahlhaus, Efficient parallel algorithms for cographs and distance hereditary graphs, Discrete Applied Mathematics 57 (1995) 29–54.
[48] M. Habib, C. Paul, A simple linear time algorithm for cograph recognition, Discrete Applied Mathematics 145 (2) (2005) 183–187.
[49] A. Bretscher, D.G. Corneil, M. Habib, C. Paul, A simple linear time LexBFS cograph recognition algorithm, in: International Workshop on Graph-Theoretic Concepts in Computer Science, WG, in: Lecture Notes in Computer Science, vol. 2880, 2003, pp. 119–130.
[50] E. Gioan, C. Paul, Dynamic distance hereditary graphs using split decomposition, in: International Symposium on Algorithms and Computation, ISAAC, in: Lecture Notes in Computer Science, vol. 4884, 2007.
[51] J.P. Spinrad, Efficient Graph Representation, in: Fields Institute Monographs, vol. 19, American Mathematical Society, 2003.
[52] J.-L. Fouquet, M. Habib, F. de Montgolfier, J.-M. Vanherpe, Bimodular decomposition of bipartite graphs, in: International Workshop on Graph Theoretical Concepts in Computer Science, WG, in: Lecture Notes in Computer Science, vol. 3353, 2004, pp. 117–128.
[53] W.H. Cunningham, Decomposition of directed graphs, SIAM Journal on Algebraic and Discrete Methods 3 (1982) 214–228.
[54] R. Shamir, R. Sharan, A fully dynamic algorithm for modular decomposition and recognition of cographs, Discrete Applied Mathematics 136 (2–3) (2004) 329–340.
[55] J.H. Muller, J.P. Spinrad, Incremental modular decomposition, Journal of the ACM 36 (1) (1989) 1–19.
[56] A. Bretscher, D.G. Corneil, M. Habib, C. Paul, A simple linear time LexBFS cograph recognition algorithm, SIAM Journal on Discrete Mathematics 22 (4) (2008) 1277–1296.
[57] D.J. Rose, R.E. Tarjan, G.S. Lueker, Algorithmic aspects of vertex elimination on graphs, SIAM Journal on Computing 5 (2) (1976) 266–283.
[58] C. Crespelle, C. Paul, Fully-dynamic recognition algorithm and certificate for directed cographs, Discrete Applied Mathematics 154 (12) (2006) 1722–1741.
[59] C. Crespelle, Fully dynamic representation of interval graphs, in: International Workshop on Graph Theoretical Concepts in Computer Science, WG, in: Lecture Notes in Computer Science, vol. 5911, 2009, pp. 77–87.
[60] L. Ibarra, A fully dynamic graph algorithm for recognizing interval graphs, Algorithmica (2009).
[61] P. Hell, R. Shamir, R. Sharan, A fully dynamic algorithm for recognizing and representing proper interval graphs, SIAM Journal on Discrete Mathematics 31 (1) (2001) 289–305.
[62] E. Gioan, C. Paul, M. Tedder, D. Corneil, Practical split-decomposition via graph-labelled trees, 2009 (submitted for publication).
[63] E. Gioan, C. Paul, M. Tedder, D. Corneil, Quasi-linear circle graph recognition, 2009 (submitted for publication).
[64] J. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, Theory of machines and computations (1971) 189–196.
[65] I. Rapaport, K. Suchan, I. Todinca, Minimal proper interval completions, Information Processing Letters 106 (5) (2008) 195–202.
[66] M. Habib, R.M. McConnell, C. Paul, L. Viennot, Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing, Theoretical Computer Science 234 (2000) 59–84.
[67] M. Habib, C. Paul, L. Viennot, A synthesis on partition refinement: A useful routine for strings, graphs, boolean matrices and automata, in: Symposium on Theoretical Aspect of Computer Science, STACS, in: Lecture Notes in Computer Science, vol. 1373, 1998, pp. 25–38.
[68] D.G. Corneil, Lexicographic breadth first search — A survey, in: International Workshop on Graph-Theoretic Concepts in Computer Science, WG, in: Lecture Notes in Computer Science, vol. 3353, 2004, pp. 1–19.
[69] D.G. Corneil, A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs, Discrete Applied Mathematics 138 (3) (2004) 371–379.
[70] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Algorithms, MIT Press, 1990.
[71] A. Bergeron, S. Herber, J. Stoye, Common intervals and sorting by reversals: A marriage of necessity, in: European Conference on Computational Biology, Bioinformatics (2002) 54–63.
[72] S. Herber, R. Mayr, J. Stoye, Common intervals of multiple permutations, Algorithmica (2009).
[73] A. Bergeron, C. Chauve, F. de Montgolfier, M. Raffinot, Computing common intervals of k permutations, with applications to modular decomposition of graphs, in: European Symposium on Algorithms, ESA, in: Lecture Notes in Computer Science, vol. 3669, 2005, pp. 779–790.
[74] M. Habib, F. de Montgolfier, C. Paul, A simple linear-time modular decomposition algorithm, in: Scandinavian Workshop on Algorithm Theory, SWAT, in: Lecture Notes in Computer Science, vol. 3111, 2004, pp. 187–198.
[75] M. Atkinson, T. Stitt, Restricted permutations and the wreath product, Discrete Mathematics 259 (1–3) (2002) 19–36.
[76] M. Albert, M. Atkinson, Simple permutations and pattern restricted permutations, Discrete Mathematics 300 (1–3) (2005) 1–15.
[77] M. Bouvel, D. Rossin, The longest common pattern problem for two permutations, Pure Mathematics and Applications 17 (1-2) (2006) 55–69.
[78] B.-M. Bui-Xuan, M. Habib, C. Paul, Revisiting Uno and Yagiura's algorithm, in: International Symposium on Algorithms and Computation, ISAAC, in: Lecture Notes in Computer Science, vol. 3827, 2005, pp. 146–155.
[79] F. de Mongolfier, Décomposition modulaire des graphes - Théorie, extensions et algorithmes, Ph.D. Thesis, Univ. de Montpellier II, 2003.
[80] S. Bérard, C. Chauve, C. Paul, A more efficient algorithm for perfect sorting by reversals, Information Processing Letters 106 (2008) 90–95.

[81] S. Bérard, A. Bergeron, C. Chauve, C. Paul, Perfect sorting by reversals is not always difficult, IEEE/ACM Transaction on Computational Biology and Bioinformatics 4 (1) (2007) 4–16.
[82] R.G. Downey, M.R. Fellows, Parameterized Complexity, Springer, 1999.
[83] R. Niedermeier, Invitation to Fixed Parameter Algorithms, in: Oxford Lectures Series in Mathematics and its Applications, vol. 31, Oxford University Press, 2006.
[84] J. Flum, M. Grohe, Parameterized complexity theory, in: Texts in Theoretical Computer Science, Springer, 2006.
[85] F. Protti, M. Dantas da Silva, J.L. Szwarcfiter, Applying modular decomposition to parameterized cluster editing problems, Theory of Computing Systems 44 (1) (2009) 91–104.
[86] M. Dom, J. Guo, F. Hüffner, R. Niedermeier, A. Truss, Fixed-parameter tractability results for feedback set problems in tournaments, in: Italian Conference on Algorithms and Complexity, CIAC, in: Lecture Notes in Computer Science, vol. 3998, 2006, pp. 320–331.
[87] S. Bessy, C. Paul, A. Perez, Polynomial kernels for 3-leaf power graph modification problems, in: International Workshop on Combinatorial Algorithm, IWOCA, in: Lecture Notes in Computer Science, vol. 5874, 2009, pp. 72–82.
[88] S. Böcker, Q.B. Anh Bui, A. Truß, An improved fixed-parameter algorithm for minimum-flip consensus trees, in: International Workshop on Parameterized and Exact Computation, IWPEC, in: Lecture Notes in Computer Science, vol. 5018, 2008, pp. 43–54.
[89] L. Cai, Fixed-parameter tractability of graph modification problems for hereditary properties, Information Processing Letters 58 (4) (1996) 171–176.
[90] J. Guo, A more effective linear kernelization for cluster editing, in: International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, ESCAPE, in: Lecture Notes in Computer Science, vol. 4614, 2007, pp. 36–47.
[91] M.R. Fellows, M. Langston, F. Rosamond, P. Shaw, Efficient parameterized preprocessing for cluster editing, in: International Symposium on Fundamentals of Computation Theory, FCT, in: Lecture Notes in Computer Science, vol. 4639, 2007, pp. 312–321.
[92] J. Guo, F. Hüffner, C. Komusiewicz, Y. Zhang, Improved algorithms for bicluster editing, in: Annual Conference on Theory and Applications of Models of Computation, TAMC, in: Lecture Notes in Computer Science, vol. 4978, 2008, pp. 445–456.
[93] S. Bessy, F. Fomin, S. Gaspers, C. Paul, A. Perez, S. Saurabh, S. Thomassé, Kernels for feedback arc set in tournaments, in: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 4, 2009, pp. 37–47.
[94] J. Fiala, D. Paulusma, The computational complexity of the role assignment problem, in: International Colloquium on Automata, Languages and Programming, ICALP, in: Lecture Notes in Computer Science, vol. 2719, 2003, pp. 817–828.