Revisiting T. Uno and M. Yagiura's Algorithm

Binh Minh Bui Xuan Michel Habib Christophe Paul

Juin 2005

Rapport de Recherche Lirmm RR-05049

161, rue Ada - F. 34394 Montpellier cedex 5 - Tél. 33 (0) 4 67 41 85 85 - Fax. 33 (0) 4 67 41 85 00 - www.lirmm.fr

Abstract.
In 2000, T. Uno and M. Yagiura published an algorithm that computes all the K common intervals of two given permutations of length n in $\mathcal{O}(n + K)$ time. Our paper first presents a decomposition approach to obtain a compact encoding for common intervals of d permutations. Then, we revisit T. Uno and M. Yagiura's algorithm to yield a linear time algorithm for finding this encoding. Besides, we adapt the algorithm to obtain a linear time modular decomposition of an undirected graph, and thereby propose a formal invariant-based proof for all these algorithms.

B.-M. Bui Xuan
buixuan@lirmm.fr

M. Habib
habib@lirmm.fr

C. Paul
paul@lirmm.fr

1 Introduction

T. Uno and M. Yagiura's algorithm [27] computes all the K *common intervals* of two permutations of length n in $\mathcal{O}(n+K)$ time. Therein, each genome is regarded as a permutation on a finite set of genes, and a common interval of two genomes refers to a set of genes that are consecutive on each genome. This notion formalises the concept of a gene cluster. Afterwards, F. de Montgolfier pointed out strong relationships between *modules* of a *permutation graph* and common intervals of any of its *realiser* (made of two permutations) [26]. This allows to define the *common interval decomposition tree* for this case. From recent works, the tree turns out to own some important biological meaning [2, 22]. Particularly, common intervals help out with finding evolutionary distances between the corresponding species [3, 4, 16, 22]. Finally, common intervals can be interpreted as pieces of each genome that have been *conserved* all along an evolutionary scenario between the involved species and their common ancestor [2].

The seminal algorithmic result on common intervals is due to T. Uno and M. Yagiura (Fig. 1). This really is a masterpiece among combinatorial algorithms as it uses a unique scan on one of the two permutations and could be seen as a sweep plane paradigm as used in computational geometry [14]. However, its correctness proof is a little hard to understand. Later, S. Heber and J. Stoye pointed out a smaller and generating sub-family, so-called the family of *irreducible* common intervals, and succeeded in adapting T. Uno and M. Yagiura's algorithm to find all irreducible common intervals of d permutations in $\mathcal{O}(d \times n)$ time [20]. Besides, generating all the K common intervals from this sub-family is in $\mathcal{O}(K)$ time [20]. While they used T. Uno and M. Yagiura's scheme as a black box, they did not give further explanations for the correctness proof.

T. Uno and M. Yagiura's general scheme:
1. Let `Potential` be an empty list
2. **For** $i = n$ down to 1 **Do**
3. (Filter): Remove all known boundaries r in `Potential` such that for all $l \leq i$, (l, r) is not a common interval
4. (Add): Add i to the head of `Potential`
5. (Extract): While there still is some boundary r of `Potential` such that (i, r) is a common interval, output (i, r)
6. **End of for**

Fig. 1. A list `Potential` is used. It contains at each step i all boundaries $r \geq i$ such that there is some $l \leq i$ with (l, r) a common interval. Then, `Potential` is (partially) traced to output the common intervals of the form (i, r). Though `Potential` is filtered as many time as possible, the main difficulty of this approach relies on the linear time complexity while the idea is based on a double iteration.

In this paper, we first show how T. Uno and M. Yagiura's algorithm can easily be adapted to compute in $\mathcal{O}(n)$ time a tree representation of all common intervals of two permutations on n elements. As a by-product we propose a complete invariant-based proof of the algorithm, and detailed the complexity analysis. In Section 3, we generalise T. Uno and M. Yagiura's algorithm to be a central step for modular decomposition algorithms of undirected graphs. To close the paper, we discuss on the important role that this combinatorial algorithm could play in the near future.

2 Common Interval Decomposition

Let us denote $\mathbb{N}_n = [\![1, n]\!] = \{1, 2, \ldots, n\}$. A permutation π on a finite set V is regarded indifferently as a bijection from $\mathbb{N}_{|V|}$ to V, a total order on V, or a word in V^* without multiple occurrence. The support of a factor of π is called an *interval of* π, noted $\pi([\![l, r]\!])$ with $l, r \in \mathbb{N}_{|V|}$ its left and right boundaries. A *common interval* of two permutations on V is interval of each (see Figure 2). There could be a quadratic number of those, e.g. when the permutations are identical. The decomposition addressed in this paper is based on the seminal works on weakly partitive families [9, 25]. Let us recall some useful formalisms.

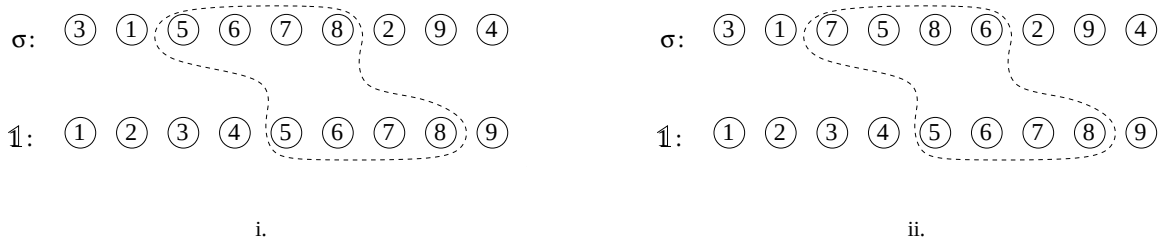


Fig. 2. In both examples, $\sigma([\![3, 6]\!]) = \{5, 6, 7, 8\} = \mathbb{1}([\![5, 8]\!])$ is a common interval.

2.1 Combinatorial Decomposition Aspects

Let be given a finite set V. By convention, a member of a family $\mathcal{F} \subseteq 2^V$ is referred to as a *subset (of V) in* $\mathcal{F}$: a subset A in $\mathcal{F}$ satisfies $A \in \mathcal{F}$ while a subset $\mathcal{S}$ of $\mathcal{F}$ is such that $\mathcal{S} \subseteq \mathcal{F}$. Two subsets of V *overlap* when none of their intersection and differences is empty. A family $\mathcal{F} \subseteq 2^V$ is *weakly partitive* if and only if $\emptyset \notin \mathcal{F}$, $\mathcal{F}$ contains the trivial subsets (singletons and V), and $\mathcal{F}$ is closed by intersection, union, and differences on overlapping subsets. It is *partitive* if weakly partitive and closed by symmetric difference on overlapping subsets [9, 25]. A weakly partitive family can have $\mathcal{O}(n!)$ members, e.g. with $\mathcal{F} = 2^V$.

Let $\mathcal{F} \subseteq 2^V$ be weakly partitive. A member $S \in \mathcal{F}$ is *strong* when it does not overlap any other $F \in \mathcal{F}$. The subset of $\mathcal{F}$ containing all strong members of $\mathcal{F}$ is denoted $\mathcal{S}_\mathcal{F}$. The members of $\mathcal{S}_\mathcal{F}$ can be organised by inclusion order in a tree, so-called the *decomposition tree* and noted $\mathcal{T}_\mathcal{F}$. The size of $\mathcal{T}_\mathcal{F}$ is $\mathcal{O}(n)$.

Theorem 1. *[9, 25] Except for binary nodes, an internal node in $\mathcal{T}_\mathcal{F}$ satisfies one and only one of the following: (*Prime *node) no union of children belongs to $\mathcal{F}$, except for the node itself; (*Degenerate *node) all union of children belongs to $\mathcal{F}$; (*Linear *node) there is a children ordering such that a union of children belongs to $\mathcal{F}$ if and only if they are consecutive in this order.*

Roughly, the tree $\mathcal{T}_\mathcal{F}$ is a (compact) encoding of $\mathcal{F}$ from which all members of $\mathcal{F}$ can easily be generated. A permutation σ is *factorising* for $\mathcal{F}$ if and only if any strong subset $S \in \mathcal{S}_\mathcal{F}$ is an interval of σ [8]. In other words, a factorising permutation is a visit-order of the leaves of $\mathcal{T}_\mathcal{F}$ by a depth-first graph search. Though the following property is trivial, it yields a formal decomposition framework for common intervals. Fig. 3 exemplifies the common interval decomposition.

Fig. 3. Common interval decomposition tree. "L" stands for *Linear* and "P" for *Prime*.

Property 1 *The family $\mathcal{CI}$ of common intervals of two permutations σ_1 and σ_2 satisfies three following properties: $\mathcal{CI}$ is weakly partitive; $\mathcal{T}_{\mathcal{CI}}$ has no* Degenerate *nodes; and both σ_1 and σ_2 are factorising.*

A common interval is *reducible* if it is union of consecutively overlapping non-trivial common intervals, and is *irreducible* when not reducible [20].

Property 2 *The irreducible common intervals of two permutations exactly are* Prime *nodes and pairs of consecutive children of* Linear *nodes.*

Property 2 is straightforward from the definitions, and establishes a reciprocal link between the decomposition tree and irreducible common intervals. In particular, it states that one can compute in $\mathcal{O}(n)$ time the tree from this family and conversely. Both the notion of irreducibility and Property 2 can easily be generalised to any weakly partitive family.

In the following, we give an alternative for computing the common interval decomposition tree, based on the notion of *right-strong intervals*. Let it be emphasised that both notions of irreducibility and right-strong intervals merely are combinatorial tools to remove the term "K" in the raw common intervals $\mathcal{O}(n + K)$ computing time. Fortunately, both notions can be adapted in the sweep paradigm of T. Uno and M. Yagiura's algorithm.

2.2 Right-Strong Intervals

Let $\sigma = \sigma_1$ and σ_2 be two permutations on V. Let $\mathcal{CI}$ refer to the family of their common intervals. Then, σ is factorising for $\mathcal{CI}$. W.l.o.g., from now on, *intervals* will stand for intervals of σ. By definition, a common interval is an interval.

Definition 1 (Right-Strong Interval). *Given a factorising permutation σ for a (weakly) partitive family $\mathcal{F} \subseteq 2^V$, an interval $\sigma(\llbracket i, j \rrbracket) \in \mathcal{F}$ is* right-strong *if and only if it does not overlap on its right any other interval of σ that belongs to $\mathcal{F}$, namely if and only if $i < i' \leq j < j'$ implies $\sigma(\llbracket i', j' \rrbracket) \notin \mathcal{F}$.*

Roughly, a right-strong interval of $\mathcal{CI}$ is a member of $\mathcal{CI}$ that does not overlap any other member of $\mathcal{CI}$ on its right in the order σ. Their number is bounded by $2 \times n$ from Corollary 1 below. To formalise their computation, let us define $\texttt{Select}(i)$ as $\forall n \geq i \geq 1$, $\texttt{Select}(i) = \{j \mid \sigma(\llbracket i, j \rrbracket)$ is a right-strong interval $\}$.

Definition 2 (Useless Boundary). *While inspecting σ from n down to 1, $\sigma(\llbracket l, r \rrbracket)$ is* visited *at step i if $i < l$,* unvisited *otherwise. Then, $r \in \llbracket i, n \rrbracket$ is* useless *w.r.t. i if none of the unvisited right-strong intervals is of the form $\sigma(\llbracket l, r \rrbracket)$.*

Lemma 1. *Let m_i be the maximum boundary such that $\sigma([\![i, m_i]\!]) \in \mathcal{F}$. Then, $m_i = \max \texttt{Select}(i)$ and for all $i < r < m_{i+1}$, r is useless w.r.t. i.*

Proof. If $\sigma([\![i, m_i]\!])$ overlaps $\sigma([\![i', m']\!])$ on its right, then $\sigma([\![i, m']\!]) \in \mathcal{F}$ (partitivity) and m_i is not maximum. Therefore, $m_i \in \texttt{Select}(i)$. Then, $m_i = \max \texttt{Select}(i)$ is trivial. Besides, for all $l < i + 1 \leq r < m_{i+1}$, $\sigma([\![l, r]\!])$ overlaps $\sigma([\![i + 1, m_{i+1}]\!])$ on its right. □

Corollary 1. $|\texttt{Select}(1)| \ + \ldots + \ |\texttt{Select}(n)| \leq 2 \times n$.

Proof. From Lemma 1, the sets $\texttt{Select}(i) \setminus \{\max \texttt{Select}(i)\}$ $(1 \leq i \leq n)$ are pairwise disjunctive and their total cardinal is bounded by n. □

2.3 Right-Strong Intervals of Two Permutations Computation

With a slight modification, namely by adding an one-line routine, T. Uno and M. Yagiura's algorithm computes in $\mathcal{O}(n)$ time the family of *right-strong intervals* of two permutations $\sigma = \sigma_1$ and σ_2 on V, where $n = |V|$. However, we will detail its correctness, since the original version is tough to understand. In this algorithm, the sets $\texttt{Select}(i)$ $(n \geq i \geq 1)$ are computed using a list `Potential`. At each iteration step i, this list contains the right boundaries $r \geq i$ of all unvisited right-strong intervals.

`Potential` is initialised as an empty list. Each iteration step $n \geq i \geq 1$ aims at removing from `Potential` as many useless boundaries w.r.t. i as possible. However, we need some formalisms before coming to the filtering. Let $C_2(i, j)$ refer to the convex hull in σ_2 of the interval $\sigma([\![i, j]\!])$, i.e. $C_2(i, j) = \sigma_2([\![l, r]\!])$ where $l = \min\{k \mid \sigma_2(k) \in \sigma([\![i, j]\!])\}$ and $r = \max\{k \mid \sigma_2(k) \in \sigma([\![i, j]\!])\}$. We define $\mathcal{S}_{\sigma([\![i,j]\!])} = C_2(i, j) \setminus \sigma([\![i, j]\!])$ as the *splitter set* of $\sigma([\![i, j]\!])$. Roughly, a splitter of an interval makes the interval not a common interval. Let $s(\sigma([\![i, j]\!])) = |\mathcal{S}_{\sigma([\![i,j]\!])}| = s_i(j)$ $(j \geq i)$ count the number of interval splitters. For all member p_j of `Potential`, we define $\delta_i(p_j) = s_i(p_{j+1}) - s_i(p_j)$ if p_j has a successor p_{j+1}, $\delta_i(p_j) = +\infty$ otherwise. Then, Theorem 2 below is fundamental and most results thereafter rely on it. However, from our standpoint, the theorem is easier to prove and most comprehensive when generalised to Theorem 4 in Section 3.1.

Property 3 *[27] $\sigma([\![i, j]\!])$ is a common interval if and only if $s_i(j) = 0$.*

Theorem 2. *[27] $\delta_i(p_j) < 0$ implies p_j is useless w.r.t. i.*

At the beginning of step i, assume that some Update-Detect routine provides for any member p_j of `Potential` a pointer to the value of $s_i(p_j)$. Besides, assume that this routine also outputs a list `Detected` of pointers to all members p_j of `Potential` with $\delta_i(p_j) < 0$, and possibly to some other useless boundaries w.r.t. i. Finally, assume that the pointed $p_{j_1} < \ldots < p_{j_h}$ are organised increasingly.

Then, `Potential` is filtered twice. The first filtering (Pre-Filter) is our only addition to the original algorithm to make it compute right-strong intervals instead of all common intervals. It follows from Lemma 1, which states that it is possible to move apart some useless boundaries w.r.t. i even before considering $\sigma(i)$. To do this, from the previous computation of $\texttt{Select}(i+1)$, a pointer to $r_0 = \max \texttt{Select}(i + 1)$ is kept. Then, members of `Potential` strictly before r_0 are removed. For use in Section 2.4, if some boundaries were removed, r_0 receives the mark *Eaten*. Those take linear time on the number of removed boundaries.

The second filtering (Customised Filter) backtracks `Detected` from p_{j_h} down to p_{j_1}. Each p_{j_k} is removed from `Potential` if still there. If some removing makes the next-left boundary p' have $\delta_i(p') < 0$, p' is also removed and so on. Thus, only useless boundaries w.r.t. i are removed, and all remaining boundaries have positive δ_i. This takes linear time on the number of removed boundaries.

The boundary i is then added to the head of `Potential` (Add). Notice that $\delta_i(i) \geq 0$ as $s_i(i) = 0$. Here, the update of `Potential` of step i is complete.

Invariant 1 *After the update of step i, let p_{j_0} be the first member of* `Potential` *with $s_i(p_{j_0}) \neq 0$. Then,* `Select`$(i) = \{r < p_{j_0} \mid r$ *is a member of* `Potential`$\}$.

Proof. After the update, all p_j have $\delta_i(p_j) \geq 0$. If $r \in$ `Select`(i), then $s_i(r) = 0$ and $r < p_{j_0}$. Besides, $\sigma([\![i, r]\!])$ is unvisited at step i. Hence, r still is a member of `Potential`, and it is strictly before p_{j_0}. Conversely, any member $r < p_{j_0}$ of `Potential` after the update hold $s_i(r) = 0$. If $\sigma([\![i, r]\!])$ overlaps some $\sigma([\![i', r']\!])$ on its right, then $i < i' \leq r < r'$, $\sigma([\![i', r]\!]) \in \mathcal{CI}$, $\sigma([\![i', r']\!]) \in \mathcal{CI}$ and the Pre-Filter at step i' would remove r from `Potential` if it was still there. □

Outputting `Selected`(i) from the list `Potential` (Extract) follows from Invariant 1. Its computing time obviously is linear on the size of the output.

As for complexity issues, Corollary 1 and the fact that each boundary is inserted and removed exactly once in `Potential` imply the following.

Result 1 *The right-strong intervals computing time is $\mathcal{O}(n)$ if Update-Detect runs in linear time on the size of the output* `Detected` *at each iteration step i.*

T. Uno and M. Yagiura's algorithm revisited:
1. Let `Potential` be an empty list and `Select`$(n+1) = \emptyset$
2. **For** $i = n$ down to 1 **Do**
3. (Update-Detect): Collect all known useless boundaries w.r.t. i
4. (Pre-Filter): If there are some $r < r_0 (= \max$ `Select`$(i+1))$ in `Potential`, remove them and mark r_0 as *Eaten*
5. (Customised Filter): Remove all known useless boundaries w.r.t. i
6. (Add): Add the boundary i to the head of `Potential`
7. (Extract): Find the right-most r_q in `Potential` with $s_i(r_q) = 0$ and output `Select`$(i) = \{r_1 \dots r_q\}$
8. **End of for**

Fig. 4. A sketch of the computation of right-strong intervals of two permutations.

The Update-Detect routine in the case of common intervals can be implemented as follows [27]. Let `Potential` $= [p_1 (= i+1), \dots, p_l]$ at the beginning of step i. The routine aims at updating two lists `Min` $= [Min_1, \dots, Min_s]$ and `Max` $= [Max_1, \dots, Max_t]$. Each Min_j is a boundary between 1 and n with two pointers `first`(Min_j) and `last`(Min_j) to two members of `Potential`. All p_j between these two members have to satisfy $Min_j = \min\{k \mid \sigma_2(k) \in \sigma([\![i, p_j]\!])\}$. It is analogous for `Max`. Besides, each member p_j of `Potential` has two pointers `Min`(p_j) and `Max`(p_j) to the corresponding members of `Min` and `Max`. By supposing $V = [\![1, n]\!]$, computing $s_i(p_j)$ from this data structure is in $\mathcal{O}(1)$ time.

Let $\texttt{Min} = [Min'_1, \ldots, Min'_{s'}]$ and $\texttt{Max} = [Max'_1, \ldots, Max'_{t'}]$ at the beginning of step i. Suppose inductively that $C_2(i+1, p_j) = \sigma_2([\![\texttt{Min}(p_j), \texttt{Max}(p_j)]\!])$ for all p_j and that `Min`, resp. `Max`, is strictly decreasing, resp. increasing. Notice that $\sigma_2(Min'_1) = \sigma_2(Max'_1) = \sigma(i+1)$. Now, i' with $\sigma_2(i') = \sigma(i)$ can be obtained in $\mathcal{O}(1)$ time. Then, either $i' < Min'_1$ and `Max` will be unchanged, or $Max'_1 < i'$ and `Min` unchanged. We trace `Min`, resp. `Max`, from $j = 1$ until finding the first j^* with $Min'_{j^*} \leq i' < Max'_1$, resp. $Min'_1 < i' \leq Max'_{j^*}$. Notice that $j^* > 1$ and let $p_{j_0} = \texttt{first}(Min'_{j^*-1})$, resp. $p_{j_0} = \texttt{first}(Max'_{j^*-1})$. The index j_0 for members of `Potential` can be much higher than the index j^* for members of `Min` or `Max`.

Lemma 2. *[27] p_j is useless w.r.t. i if $s_i(p_j) - s_{i+1}(p_j) > s_i(p_{j+1}) - s_{i+1}(p_{j+1}) \geq 0$.*

Invariant 2 *(equivalent to Lemma 2) p_j with $1 \leq j < j_0$ is useless w.r.t. i.*

W.l.o.g. $Min'_{j^*} \leq i' < Max'_1$, we set Min'_{j^*-1} to i'; point $\texttt{first}(Min'_{j^*-1})$ to p_1; and for all $1 \leq j < j_0$, point $\texttt{Min}(p_j)$ to Min'_{j^*-1}. Thus, each p_j is such that $C_2(i, p_j) = \sigma_2([\![\texttt{Min}(p_j), \texttt{Max}(p_j)]\!])$. It is trivial to maintain this fact after the insertions and deletions in `Potential` by some simple updates on the data structure. Hence, the inductive hypothesis for the next step holds. Finally, we define the list `Detected` of pointers to respectively $p_1 < \ldots < p_{j_0-1}$. Now, the only member of `Potential` where δ_i can be negative that is not pointed by `Detected` is $p_{j_1} = \texttt{last}(Min_{j^*-1})$. Thus, if $\delta_i(p_{j_1}) < 0$, we add a pointer to p_{j_1} to the end of `Detected`. Then, the routine outputs `Detected`. The running time of this routine is $\mathcal{O}(j_0 + j^*) = \mathcal{O}(j_0) = \mathcal{O}(|\texttt{Detected}|)$.

The running of the whole algorithm is exemplified in Figure 5.

Result 2 *Right-strong intervals of two permutations computing time is $\mathcal{O}(n)$.*

Remark 1. Ideally, at each iteration step i, `Potential` would contain *only* the right boundaries $r \geq i$ of all unvisited right-strong intervals. Could this somehow be proven ?

2.4 Common Interval Decomposition of Two Permutations

After computing the right-strong intervals, a sweep from left-to-right in a symmetric manner generates the strong common intervals in $\mathcal{O}(n)$ time. We recall that those are the nodes of the decomposition tree. Moreover, the sweep organises these nodes by interval inclusion, i.e. filial order in the tree. Hence, constructing the tree is in $\mathcal{O}(n)$ time. Then, the labelling can be in $\mathcal{O}(n)$ time using the following remarks. Since there are only *Prime* and *Linear* nodes, the strong common intervals that are marked *Eaten* by the Pre-Filter routine in the right-strong intervals computation also have this mark in the left-strong intervals computation. Furthermore, the nodes with the mark *Eaten* are *Linear*, the others are *Prime*.

Finally, Property 3, Theorem 2, and Lemma 2 can easily be generalised to the case of d permutations if one replaces $C_2(i, p_j)$ with $C_j = \mathcal{S}_{\sigma([\![i,p_j]\!])} \uplus \sigma([\![i, p_j]\!]) = \cup_{h=2}^{d} C_h(i, p_j)$. Then, at each iteration step i in the new Update-Detect routine, one has to maintain C_j for any member p_j of `Potential` rather than just $C_2(i, p_j)$. The hitch lays on the fact that $C_h(i, p_j)$ $(2 \leq h \leq d)$ are not pairwise disjunctive. However, as an element $\sigma(i')$ can be added to some $C_h(i'', p''_j)$ only once throughout the computation, the total maintenance can be done in $\mathcal{O}(d \times n)$ time.

Result 3 *The common interval decomposing time of d permutations is $\mathcal{O}(d \times n)$.*

$\sigma_2(i) = \mathbb{1}$	1	2	3	4	5	6	7	8	9
$\sigma(i)$	3	1	5	6	7	8	2	9	4
i	1	2	3	4	5	6	7	8	9
`Negative`	—	6	—	—	—	—	8	—	—
`Detected`	2	6	—	—	—	7	8	—	—
`Select`$(i+1)$	2	3, 6	4, 6	5, 6	6	7	8	9	—
removed by Pre-Filter	—	3	4	5	—	—	—	—	—
removed by Customised Filter	2	6	—	—	—	7	8	—	—
`Potential`	1, 9	2, 9	3, 6, 9	4, 6, 9	5, 6, 9	6, 9	7, 9	8, 9	9
`Select`(i)	1, 9	2	3, 6	4, 6	5, 6	6	7	8	9

Fig. 5. The values of some variables at the end of step i of the right-strong intervals computation on the instance of Fig. 3. We define `Negative` containing all member p_j of `Potential` at the *beginning* of step i with $\delta_i(p_j) < 0$.

3 Modular Decomposition

Let be given a loopless simple undirected graph $G = (V, E)$ with $n = |V|$ and $m = |E|$. A vertex $v \in V \setminus X$ exterior to a non-empty vertex subset X is *adjacent to* X if and only if it is adjacent to each vertex of X, *non-adjacent to* X if and only if non-adjacent to each vertex of X. In both cases, v is *uniform to* X. X is a *module* of G if all its exterior vertices are uniform to X. Any exterior vertex that is not uniform to X is defined as a *splitter of* X. Roughly, the family $\mathcal{M}$ of modules of G refers to the set of subgraphs of G that behave as one single vertex. It is well-known that $\mathcal{M}$ is partitive [9, 25].

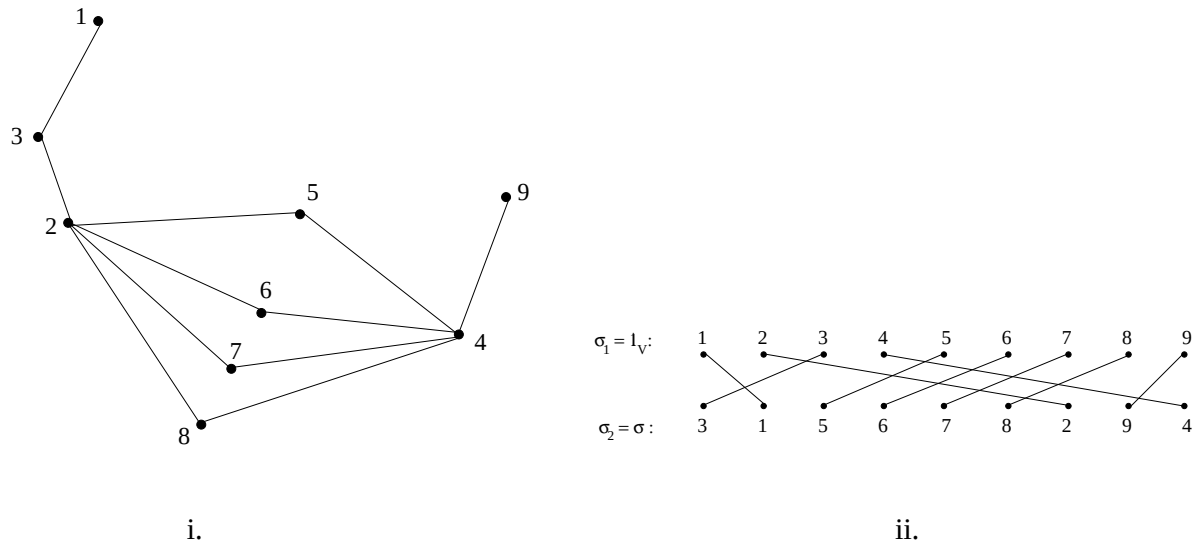
Finding efficient algorithms for computing $\mathcal{T}_{\mathcal{M}}$ from G has been an important challenge of the last two decades [7, 8, 10, 11, 13, 18, 24, 26]. This results in several algorithms reaching linear time performance. The *factorising permutations of a graph* refer to the ones of the family of its modules. Obtaining one such permutation is in linear time for chordal graphs [21], and inheritance graphs [17]. Recently, for arbitrary graphs, this has also been solved in linear time [18]. The decomposition approach conducted by C. Capelle [7] is as follows. First find a *factorising permutation* [18], then construct the modular decomposition tree [8]. Both computations run in $\mathcal{O}(n+m)$ time even if the latter [8] is somewhat heavily fathered.

There is a link between modular decomposition and common interval decomposition as follows.

Theorem 3. *[26] Each permutation of the realiser of a permutation graph is factorising for the family of modules of the graph.*

Corollary 2. *The family $\mathcal{CI}$ of common intervals of two permutations is included in the family $\mathcal{M}$ of modules of the permutation graph that realises this pair of permutations. Furthermore, $\mathcal{CI}$ and $\mathcal{M}$ share the same strong subsets, namely $\mathcal{S}_{\mathcal{CI}} = \mathcal{S}_{\mathcal{M}}$. Finally, $\mathcal{T}_{\mathcal{CI}}$ is the same as $\mathcal{T}_{\mathcal{M}}$ where* Degenerate *labels are replaced by* Linear *labels.*

Proof. directly follows from Theorem 3.



i. ii.

Fig. 6. i. an instance G of permutation graphs. ii. one of the realisers of G.

From Corollary 2, the algorithm of the previous section is equivalent to a $\mathcal{O}(n)$ time modular decomposition algorithm for a permutation graph given by one realiser, yet $m = \theta(n^2)$. In this section, given a factorising permutation σ, we compute the modular decomposition tree of G. Let us first adapt Property 3 and Theorem 2.

3.1 Submodularity on the Size of the Splitter Sets

Let $\mathcal{S}_X$ refer to the splitter set of a vertex subset $X \neq \emptyset$ and $s(X) = |\mathcal{S}_X|$ count the number of its splitters. We extend $s(\emptyset) = -n$. Property 4 and Corollary 3 below are our graph versions of respectively Property 3 and Theorem 2.

Property 4 *$\sigma([\![i,j]\!])$ is a module if and only if $s_i(j) = s(\sigma([\![i,j]\!])) = 0$.*

Definition 3 (Submodularity). *(see e.g. [28]) A set function $\mu : 2^V \to \mathbb{R}$ is* submodular *when $\mu(X) + \mu(Y) \geq \mu(X \cup Y) + \mu(X \cap Y)$, for all $X, Y \subseteq V$.*

Theorem 4 (Submodularity). *The function s counting the splitters of modules of a graph is submodular.*

Proof. Since $s(\emptyset) = -n$ and $s(X) \leq n$ for $X \subseteq V$ non-empty, the only tricky issue consists of proving the submodular inequality for a pair (X, Y) of overlapping subsets of V. To do this,

we first notice that $\mathcal{S}_{X\cap Y} = (\mathcal{S}_{X\cap Y} \setminus Y, \mathcal{S}_{X\cap Y} \cap Y)$. Besides, $\mathcal{S}_{X\cup Y} = (\mathcal{S}_{X\cup Y} \setminus \mathcal{S}_X, \mathcal{S}_{X\cup Y} \cap \mathcal{S}_X)$ can be reduced by definition of splitters to $\mathcal{S}_{X\cup Y} = (\mathcal{S}_{X\cup Y} \setminus \mathcal{S}_X, \mathcal{S}_X \setminus (X \cup Y))$. Similarly, $\mathcal{S}_Y = (\mathcal{S}_Y \setminus \mathcal{S}_{X\cap Y}, \mathcal{S}_{X\cap Y} \setminus Y)$. Finally, $\mathcal{S}_X = (\mathcal{S}_X \setminus Y, (\mathcal{S}_X \cap Y) \setminus \mathcal{S}_{X\cap Y}, (\mathcal{S}_X \cap Y) \cap \mathcal{S}_{X\cap Y})$ can be reduced to $\mathcal{S}_X = (\mathcal{S}_X \setminus (X \cup Y), (\mathcal{S}_X \cap Y) \setminus \mathcal{S}_{X\cap Y}, \mathcal{S}_{X\cap Y} \cap Y)$. Hence, $|\mathcal{S}_X| + |\mathcal{S}_Y| - |\mathcal{S}_{X\cup Y}| - |\mathcal{S}_{X\cap Y}| = |(\mathcal{S}_X \cap Y) \setminus \mathcal{S}_{X\cap Y}| + |\mathcal{S}_Y \setminus \mathcal{S}_{X\cap Y}| - |\mathcal{S}_{X\cup Y} \setminus \mathcal{S}_X|$.

To achieve proving the lemma, we prove that $\mathcal{S}_Y \setminus \mathcal{S}_{X\cap Y} \supseteq \mathcal{S}_{X\cup Y} \setminus \mathcal{S}_X$. Indeed, let $v \in \mathcal{S}_{X\cup Y} \setminus \mathcal{S}_X$. Then, v is exterior to X and is uniform to X. By symmetry, we suppose w.l.o.g. that $v \in N_X$. Since $X \cap Y \neq \emptyset$, there exists $w \in X \cap Y$ with $(v, w) \in E$. Now, v is a splitter of $X \cup Y$, implying $u \in Y \setminus X$ such that $(v, u) \notin E$. Hence, $v \in \mathcal{S}_Y$. It is trivial by $v \in N_X$ that $v \notin \mathcal{S}_{X\cap Y}$. □

Corollary 3. *Let $i \leq p_j < p_{j+1}$, and $\delta_i(p_j) = s_i(p_{j+1}) - s_i(p_j)$. Then, $\delta_i(p_j) < 0$ implies there is no $k \leq i$ such that $\sigma([\![k, p_j]\!])$ is a module.*

Proof. If $\delta_i(p_j) < 0$, then the submodularity on the subsets $\sigma([\![k, p_j]\!])$ and $\sigma([\![i, p_{j+1}]\!])$ for all $k \leq i$ implies that $s_k(p_j) > s_k(p_{j+1}) \geq 0$. □

3.2 Modular Decomposition Algorithm

Except for the Update-Detect routine and for labelling the decomposition tree, the previous scheme can easily be adapted to the case of modules. As the Update-Detect routine in this case uses a particular data structure, its whole implementation will be depicted in Section 3.3 below. The routine computes at each step i in $\mathcal{O}(d(i))$ with $d(i)$ the degree of the vertex $\sigma(i)$. Thus, the non-labelled tree is found in $\mathcal{O}(n+m)$ time. Now, it is well-known that a modular decomposition tree has no *Linear* nodes, and its *Degenerate* nodes are divided into *Serial* – adjacency guaranteed between all children – and *Parallel* – non-adjacency between all children [9, 25]. Then, by analogous remarks as in Section 2.4, nodes with the mark *Eaten* are *Degenerate*, others are *Prime*. Furthermore, thanks to *Adjacency* marks (see the Update-Detect routine in Appendix 3.3), we can differ *Serial* from *Parallel* nodes.

Result 4 *The modular decomposition is solved in $\mathcal{O}(n + m)$ time for any graph.*

There exists a $\mathcal{O}(n)$ time common interval decomposition algorithm of two permutations [23]. However, the algorithm therein is not that simple and relies on a rather sophisticated algorithm [12]. Moreover their approach is not extended to general modular decomposition. To this aim one could use the algorithm proposed in [8]. However, this latter produces a rather heavy sequence of trees. On the other hand, our approach uses a unique paradigm for both computations of common interval and modular decomposition tree, and not only we unify the two corresponding domains but also provide very efficient elementary algorithms.

3.3 The Update-Detect routine for Modular Decomposition

Let be given a graph $G = (V, E)$ along with a factorising permutation σ. Let N_X, resp. $\overline{N}_X$, refers to the set of adjacent, resp. non-adjacent, vertices to X. Then, we here implement the Update-Detect routine to be used for modular decomposition. Let us recall what has to be done in this routine.

Let $[p_1, p_2, \ldots, p_k]$ be the value of `Potential` at the beginning of iteration step i. After the computation of Update-Detect, from each p_j, one has to be able to compute the value of $s_i(p_j)$ or $\delta_i(p_j)$ in $\mathcal{O}(1)$ time. Besides, the routine is to outputs a list `Detected` of pointers to all members p_j

of `Potential` with $\delta_i(p_j) < 0$, plus some other useless boundaries w.r.t. i. In this case of modular decomposition, we will compute `Detected` such that the pointed boundaries are exactly those that have a strictly negative δ_i. Finally, `Detected` has to organise increasingly the pointed boundaries $p_{j_1} < \ldots < p_{j_h}$.

To obtain this, we need another data structure from the one used in the common interval decomposition. Our implementation aims at a $\mathcal{O}(n+m)$ decomposition time, and follows the rule: each step i only considers the neighbourhood of $\sigma(i)$. Some notions have to be introduced.

Let $N_{i,j}$ and $\overline{N}_{i,j}$ refer to $N_{\sigma([\![i,j]\!])}$ and $\overline{N}_{\sigma([\![i,j]\!])}$ for short. Let $[p'_1, \ldots, p'_l]$ be the value of `Potential` at the end of step i in the selection phase. Then, the fact that $p'_1 (= i) < \ldots < p'_l$ implies $N_{i,p'_1} \supseteq \ldots \supseteq N_{i,p'_l}$. Therefore, the neighbourhood of $\sigma(i)$ in G can be partitioned into l *neighbour wings* $N_{i,i} = (NW_{i,p'_1}, \ldots, NW_{i,p'_l})$, where each neighbour wing is defined as $NW_{i,p'_j} = N_{i,p'_j} \setminus N_{i,p'_{j+1}}$ for all $1 \leq j \leq l$ and $N_{i,p'_{(l+1)}} = \emptyset$. The definition of the *non-neighbour wings* such that $\overline{N}_{i,i} = (\overline{N}W_{i,p'_1}, \ldots, \overline{N}W_{i,p'_l})$ is analogous. Then, the *level* L_{i,p'_j} is defined as $L_{i,p'_j} = NW_{i,p'_j} \uplus \overline{N}W_{i,p'_j}$. We define $H_i = \{\sigma(i)\}$ and trivially deduce that $V = (L_{i,p'_1}, \ldots, L_{i,p'_l}, H_i)$. The *level threshold* does not depend on i and is defined as $\theta(p'_j) = p'_{j+1} - p'_j$ for all $1 \leq j \leq l$ and $p'_{l+1} = p'_l$. Notice that its obtaining does not require any data structure since it can be directly computed from p'_j and its successor in $\mathcal{O}(1)$ time.

Property 5 *$s_i(p'_j) = n - (|[\![i, p'_j]\!]| + |N_{i,p'_j}| + |\overline{N}_{i,p'_j}|)$ for all $1 \leq j \leq l$.*

Proof. a splitter of a vertex subset is an exterior non-uniform vertex.

Corollary 4. *$\delta_i(p'_j) = s_i(p'_{j+1}) - s_i(p'_j) = |L_{i,p'_j}| - \theta(p'_j)$ for all $1 \leq j \leq l$.*

Proposition 1. *A data structure representing $V = (L_{i,p'_1}, \ldots, L_{i,p'_l}, H_i)$ with respect to partition refinements techniques [19] allows to implement the Update-Detect routine for graph modules to compute in $\mathcal{O}(|N_{i,i}|)$ time per step i.*

Indeed, from Corollary 4, the value of $\texttt{delta}(p'_j) = \delta_i(p'_j)$ can be obtained in $\mathcal{O}(1)$ from the one of $|L_{i,p'_j}|$. According to this, we use the partition refinement techniques [19] to maintain the partition of V into $V = (L_{i,p'_1}, \ldots, L_{i,p'_l}, H_i)$ at the end of each step i.

Let us assume a partition refinement function `Refine`, which takes as input a pivot set $\texttt{S} \subseteq V$ and a data structure `P` representing a partition of $(X_1, \ldots, X_p)$. Each X_i $(1 \leq i \leq q)$ has a pointer $\texttt{Size}(X_i)$ to its cardinal. Then, $\texttt{Refine}(\texttt{S}, \texttt{P})$ proceeds in $\mathcal{O}(|\texttt{S}|)$ and splits any X_i with $X_i \cap \texttt{S} \neq \emptyset$ to an *intersection subset* $I_i = X_i \cap \texttt{S}$ and a *different subset* $D_i = X_i \setminus \texttt{S}$. The two subsets has pointers to each other. The details of `Refine` are in [19].

We then define a data structure holding the following. At the beginning of each step i, besides the list $\texttt{Potential} = [p_1, \ldots, p_k]$, a data structure `Partition` is maintained with respect to the partition refinement techniques to represent $\texttt{Partition} = \big(N_{p_1}, \overline{N}_{p_1}, \ldots, N_{p_k}, \overline{N}_{p_k}, H\big)$. Each p_j points to both N_{p_j} and $\overline{N}_{p_j}$. The pointer $\texttt{delta}(p_j)$ does not exists: it is replaced by the addition of $\texttt{Size}(N_{p_j})$ and $\texttt{Size}(\overline{N}_{p_j})$, to which p_j can access in $\mathcal{O}(1)$ time.

Now, an inductive hypothesis provides $N_{p_j} = NW_{i+1,p_j}$, $\overline{N}_{p_j} = \overline{N}W_{i+1,p_j}$ and $H = H_{i+1}$. We will prove the inductive hypothesis for the next step by describing the Update-Detect routine.

To begin with, $\{\sigma(i)\}$ can be removed from `Partition` with a call to $\texttt{Refine}(\{\sigma(i)\}, \texttt{Partition})$. Indeed, there only is one single intersection subset $HTemp = \{\sigma(i)\}$, which is temporally stored apart. All the remaining is redefined `Partition`, where any member (wings or H) is the old one

excluded $\sigma(i)$. This takes $\mathcal{O}(1)$ time. Besides, an empty N'_i is created with a pointer $\texttt{Size}(N'_i)$ to 0, as well as an empty $\overline{N'}_i$ with $\texttt{Size}(\overline{N'}_i)$ to another 0. Furthermore, `Modified` is initialised to be an empty list (of pointers). This takes $\mathcal{O}(1)$ time.

After this, $\texttt{Refine}(N_{i,i}, \texttt{Partition})$ is called with some extra rules. When it splits a neighbour wing $N_{p_j} = N_{i+1,p_j} \setminus N_{i+1,p_{j+1}} \setminus \{\sigma(i)\}$ in `Partition` into two subsets, it also perform the following. First, as the intersection subset holds $N_{p_j} \cap N_{i,i} = N_{i,p_j} \setminus N_{i,p_{j+1}}$, the old neighbour wing in `Partition` is replaced by this. Second, as the difference subset $N_{p_j} \setminus N_{i,i} = L^1_{p_j} \cap \overline{N}_{i,i}$ is included in $\overline{N}_{i,i} \setminus \overline{N}_{i,p_1}$, it is concatenated to $\overline{N'}_i$ and the involved pointers `Size` are updated. Last, a pointer to p_j is added to the end of the list `Modified`. It is analogous when a non-neighbour wing $\overline{N}_{p_j} = \overline{N}_{i+1,p_j} \setminus \overline{N}_{i+1,p_{j+1}} \setminus \{\sigma(i)\}$ is split since the difference subset holds $\overline{N}_{p_j} \setminus N_{i,i} = \overline{N}_{p_j} \cap \overline{N}_{i,i} = \overline{N}_{i,p_j} \setminus \overline{N}_{i,p_{j+1}}$ and the other holds $\overline{N}_{p_j} \cap N_{i,i} \subseteq N_{i,i} \setminus N_{i,p_1}$. When splitting H, we replaced it by $HTemp = \{\sigma(i)\}$, and concatenate $H \cap N_{i,i}$ to N'_i and $H \setminus N_{i,i}$ to $\overline{N'}_i$. All these operations are in $\mathcal{O}(1)$ time per splitting operation. Therefore, the refinement is in $\mathcal{O}(|N_{i,i}|)$. At this point, the value $\big(NN_{p_1}, \overline{NN}_{p_1} \ldots, NN_{p_k}, \overline{NN}_{p_k}, HH\big)$ of `Partition` holds $NN_{p_j} = NW_{i,p_j}$, $\overline{NN}_{p_j} = \overline{N}W_{i,p_j}$, and $HH = H_i$. Thus, acceding to $|L_{i,p_j}| - \theta(p_j) = \texttt{Size}(NN_{p_j}) + \texttt{Size}(\overline{NN}_{p_j}) - (p_{j+1} - p_j) = \delta_i(p_j)$ from each p_j takes $\mathcal{O}(1)$ time, which is one of the two main results of Update-Detect. Now, we deduce by elimination that $N'_i = N_{i,i} \setminus N_{i,p_1}$ and $\overline{N'}_i = \overline{N}_{i,i} \setminus \overline{N}_{i,p_1}$. Therefore, the routine outputs $\texttt{s} = \texttt{Size}(N'_i) + \texttt{Size}(\overline{N'}_i) - (p_1 - i) = s_i(i+1)$ for further use in the Add routine.. This takes $\mathcal{O}(1)$ time.

Finally, it is obvious to state that `Modified` contains the pointers to all p_j such that $\delta_{i+1}(p_j) = \delta_i(p_j)$. Therefore, `Detected` is a sublist of `Modified` since δ_i is positive elsewhere. Besides, the pointed boundaries by `Modified` are increasing since `Partition` has a specific order of increasing levels (the increasing monotonicity of boundaries pointed by `Modified` is not necessarily strict though: a boundary might be introduced twice when either N_{p_j} and $\overline{N}_{p_j}$ have been split). Hence, obtaining `Detected` by tracing `Modified` is straightforward in $\mathcal{O}(|\texttt{Modified}|) = \mathcal{O}(|N_{i,i}|)$. This is the second of the two main results of Update-Detect.

Now, we have to prove the inductive hypothesis for the next step. To obtain this, the filtering routines (Pre-Filter, Customised Filter) have to perform some extra works: when a boundary p_j is removed, if it predecessor p_{j-1} in `Potential` exists, we concatenate the involved neighbour wings with respect to $N_{i,p_{j-1}} \setminus N_{i,p_{j+1}} = N_{i,p_{j-1}} \setminus N_{i,p_j} \uplus N_{i,p_j} \setminus N_{i,p_{j+1}}$. It is similar for the involved non-neighbour wings. If p_j is at the head of `Potential`, the wings are concatenated to N'_i and $\overline{N'}_i$. Besides, the values of `delta` have to be updated accordingly. If p_j is at the head of `Potential`, $\texttt{delta}(p_j)$ is added to `s` so that we always have $\texttt{s} = s_i(p_j)$ with p_j at the head of `Potential`. The cost of each removing still is $\mathcal{O}(1)$. Besides, for use in Section 3.2, when the Pre-Filter removes some $r < r_0 (= \max \texttt{Select}(i+1))$ and gives r_0 the mark *Eaten*, we mark r_0 as *Adjacency* if and only if $NW_{i,r}$ is not empty. Notice that either $NW_{i,r}$ is not empty and $\overline{N}W_{i,r}$ is empty or $NW_{i,r}$ is empty and $\overline{N}W_{i,r}$ is not empty. This helps distinguishing *Serial* from *Parallel* nodes. Finally, when the Add routine inserts i to the head of `Potential`, it also has to insert N'_i and $\overline{N'}_i$ to the head of `Partition`, and make i point to both of them. Let $p_1 = i$ and p_2 be the two first members of `Potential` at this state, the Add routine uses the value of $\texttt{s} = s_i(p_2) = \delta_i(p_1)$ to create the pointer $\texttt{delta}(p_1)$ for this boundary $p_1 = i$. The cost of the Add routine still is $\mathcal{O}(1)$. By doing so, it is straightforward to deduce the inductive hypothesis for the next step.

As for complexity issues, at each step i, the computing time of the Update-Detect routine is $\mathcal{O}(|N_{i,i}|)$ as explained in the above.

4 Conclusion and Perspectives

Our results are based on a gateway between both algorithms on permutations and those on graphs. We show the importance of graph layout approaches, e.g. with factorising permutations. Besides, whenever some analogous versions of Property 3 and Theorem 2 are provided, T. Uno and M. Yagiura's algorithm can easily be generalised for any weakly partitive family. Thus, the use of this algorithm would be an important crux for designing future algorithms. Indeed, it would be interesting to adopt the same philosophy conducted throughout our paper to other combinatorial problems such as decomposition into "inheritance-blocks" of an inheritance graph in $\mathcal{O}(n + m)$ time, which would yield an alternative to the algorithms proposed in [6, 7, 17]. Besides, it would also be fruitful to apply the same philosophy on modular decomposition in $\mathcal{O}(n)$ time of a bounded tolerance graph – trapezoid graph with solely parallelograms [5, 15] – when an intersection model is provided. Then, it would be very interesting to have a $\mathcal{O}(n)$ modular decomposition time for an interval or trapezoid graph on one of its intersection model, which would give interesting links to works on gene-teams [1].

5 Acknowledgements

Thanks to T. Uno and M. Yagiura for their algorithm and helpful discussions.

References

1. M.-P. Béal, A. Bergeron, S. Corteel, and M. Raffinot. An algorithmic view of gene teams. *Theoretical Computer Science*, 320(2-3):395–418, 2004.
2. S. Bérard, A. Bergeron, and C. Chauve. Conserved structures in evolution scenarios. *RECOMB04*, 2004. to appear in *Lecture Notes in Bioinformatics*.
3. A. Bergeron, S. Heber, and J. Stoye. Common intervals and sorting by reversals: A marriage of necessity. In *ECCB02*, pages S54–S63, 2002.
4. A. Bergeron and J. Stoye. On the similarity of sets of permutations and its applications to genome comparison. In *COCOON03*, volume 2697, pages 68–79, 2003.
5. K. P. Bogart, P. C. Fishburn, G. Isaak, and L. Langley. Proper and unit tolerance graphs. *Discrete Appl. Math.*, 60:99–117, 1995.
6. C. Capelle. Block decomposition of inheritance hierarchies. *WG97*, pages 118–131, 1997.
7. C. Capelle. *Décomposition de Graphes et Permutations Factorisantes*. PhD thesis, Université Montpellier II, 1997.
8. C. Capelle, M. Habib, and F. de Montgolfier. Graph decomposition and factorizing permutations. *Discrete Mathematics and Theoretical Computer Sciences*, 5(1):55–70, 2002.
9. M. Chein, M. Habib, and M.C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, 37:35–50, 1981.
10. A. Cournier and M. Habib. A new linear algorithm for modular decomposition. In S. Tison, editor, *Trees in algebra and programming–CAAP 94*, volume 787 of LNCS, pages 68–84, 1994.
11. D.D. Cowan, L.O. James, and R.G. Stanton. Graph decomposition for undirected graphs. In R.B. Levow eds. F. Hoffman, editor, *3rd S-E Conf. Combinatorics, Graph Theory and Computing, Utilitas Math*, pages 281–290, Winnipeg, 1972.
12. E. Dahlhaus. Parallel algorithms for hierarchical clustering, and applications to split decomposition and parity graph recognition. *Journal of Algorithms*, 36(2):205–240, 2000.
13. E. Dahlhaus, J. Gustedt, and R.M. McConnell. Efficient and practical algorithms for sequential modular decomposition. *Journal of Algorithms*, 41(2):360–387, 2001.
14. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry*. Springer-Verlag, 1991.
15. S. Felsner. Tolerance graphs and orders. *J. Graph Theory*, 28:129–140, 1998.
16. M. Figeac and J.-S. Varré. Sorting by reversals with common intervals. *Lecture Notes in Bioinformatics*, 3240:26–37, 2004.

17. M. Habib, M. Huchard, and J.P. Spinrad. A linear algorithm to decompose inheritance graphs into modules. *Algorithmica*, 13:573–591, 1995.
18. M. Habib, F. de Montgolfier, and C. Paul. A simple linear-time modular decomposition algorithm. *SWAT04*, LNCS(3111):187–198, 2004.
19. M. Habib, C. Paul, and L. Viennot. Partition refinement : an interesting algorithmic tool kit. *IJFCS*, 10(2):147–170, 1999.
20. S. Heber and J. Stoye. Finding all common intervals of k permutations. *CPM04*, LNCS(2089):207–218, 2001.
21. W.-L. Hsu and T.-M. Ma. Substitution decomposition on chordal graphs and applications. In *Proceedings of the 2nd ACM-SIGSAM International Symposium on Symbolic and Algebraic Computation*, number 557 in LNCS, pages 52–60, 1991.
22. G. M. Landau, L. Parida, and O. Weimann. Gene proximity analysis across whole genomes via PQ Trees, 2004. Submitted.
23. R. M. McConnell and F. de Montgolfier. Algebraic Operations on PQ Trees and Modular Decomposition Trees, 2005. To appear in Proceedings of WG05.
24. R.M. McConnell and J.P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics–SODA94*, 201:189–241, 1999.
25. R.H. Möhring and F.J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *Annals of Discrete Mathematics*, 19:257–356, 1984.
26. F. de Montgolfier. *Décomposition modulaire des graphes. Théorie, extensions et algorithmes*. PhD thesis, Université Montpellier II, 2003.
27. T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.
28. D.J.A. Welsh. Matroids: Fundamental concepts. In R.L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics*, volume 1, pages 481–526. North-Holland, 1995.