Computing common intervals of K permutations, with applications to modular decomposition of graphs

Anne Bergeron, Cedric Chauve
LaCIM and Département d'Informatique
Université du Québec à Montréal

Fabien de Montgolfier
LIAFA, Université Denis Diderot (Paris VII)

Mathieu Raffinot
CNRS - Laboratoire Génome et Informatique

Algorithms – ESA 2005, 13th Annual European Symposium, volume 3669 of *Lecture Notes in Computer Science*, pages 779–790, 2005.

Computing common intervals of K permutations, with applications to modular decomposition of graphs

Anne Bergeron[1], Cedric Chauve[1],
Fabien de Montgolfier[2], and Mathieu Raffinot[3]

[1] Département d'informatique, Université du Québec à Montréal, Canada.
[bergeron.anne,chauve.cedric]@uqam.ca
[2] LIAFA, Université Denis Diderot - Case 7014, 2 place Jussieu,
F-75251 Paris Cedex 05, France. fm@liafa.jussieu.fr
[3] CNRS - Laboratoire Génome et Informatique, Tour Evry 2, 523, Place des Terrasses de l'Agora, 91034 Evry, France. raffinot@genopole.cnrs.fr

Abstract: We introduce a new way to compute common intervals of K permutations based on a very simple and general notion of generators of common intervals. This formalism leads to simple and efficient algorithms to compute the set of all common intervals of K permutations, that can contain a quadratic number of intervals, as well as a linear space basis of this set of common intervals. Finally, we show how our results on permutations can be used for computing the modular decomposition of graphs in linear time.

1 Introduction

The notion of *common interval* was introduced in [16] in order to model the fact that, when comparing genomes, a group of genes can be rearranged but still remain connected. In [16], Uno and Yagiura proposed a first algorithm that computes the set of common intervals of a permutation P with the identity permutation in time $O(n+N)$, where n is the length of P, and N is the number of common intervals. However, N can be of size $O(n^2)$, thus the algorithm of Uno and Yagiura has an $O(n^2)$ time complexity. Heber and Stoye [10] defined a subset of size $O(n)$ of the common intervals of K permutations, called *irreducible* intervals, that forms a basis of the set of all common intervals: every common interval is a chain overlapping irreducible intervals. They proposed an $O(Kn)$ time algorithm to compute the set of irreducible intervals of K permutations, based on Uno and Yagiura's algorithm.

One of the drawbacks of these algorithms is that properties of Uno and Yagiura's algorithm are difficult to understand [4]. Even the authors describe their $O(n+N)$ algorithm as *"quite complicated"*, and, in practice, simpler $O(n^2)$ algorithms run faster on randomly generated permutations [16]. On the other hand, Heber and Stoye's algorithms rely on a complex data structure that mimics what is known, in the theory of modular decomposition of graphs, as the *PQ*-trees of *strong intervals*. An incentive to revisit this problem is the central role that these *PQ*-trees seem to play in the field of comparative genomics. Strong intervals can be used to identify significant groups of genes that are conserved between genomes [11], or as guides to reconstruct evolution scenarios [1, 8].

In order to design alternative efficient algorithms to compute common intervals, we propose a theoretical framework for common intervals based on generating families of intervals. For two permutations, these families can be computed by straightforward $O(n)$ algorithms that use only tables and stacks as data structures, and that upgrade trivially to the case of K permutations. Using these families, we compute common intervals with simple $O(n+N)$ and $O(n)$ algorithms whose properties can be readily verified. We then link this work to previous studies on common interval, and we propose a new canonical representation of the family of common intervals that is simpler than the PQ-trees.

Finally, we extend our approach to the classical graph problem of *modular decomposition* that aims to efficiently compute a compact representation of the modules of a graph. The first linear time algorithms that were developed [6, 13] are rather complex and many efforts were have been made in the design of decomposition algorithms that are efficient in practice, even if they do not run in linear time but in quasi-linear time [7, 14].

The article is structured as follows. In Section 2, we describe the notion of generators of common intervals and how to compute generators of K permutations of size n in $O(Kn)$ time. The third section explains how to generate the set of all N common intervals in $O(n+N)$ using a generator. Section 4 describes a new linear space basis of common intervals, called the canonical generator, and describes the relations between this new basis and the classical basis of strong intervals. Section 5 contains a simple linear-time algorithm to construct the strong intervals basis, given the canonical generator. Finally, in Section 6, we extend our results to the modular decomposition of graphs. Some proofs are omitted, however they can be found in [2].

2 Common intervals and generators

A *permutation* P on n elements is a complete linear order on the set of integers $\{1, 2, \ldots, n\}$. We denote Id_n the identity permutation $(1, 2, \ldots, n)$. An *interval* of a permutation $P = (p_1, p_2, \ldots, p_n)$ is a set of consecutive elements of permutation P. An interval of a permutation will be denoted by either giving its left and right bounds, such as $[i, j]$, or by giving the list of its elements, such as $(p_i, p_{i+1}, \ldots, p_j)$. An interval $[i, j] = (i, i+1, \ldots, j)$ of the identity permutation will be simply denoted by $(i..j)$.

Definition 1. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ be a set of K permutations on n elements. A *common interval* of $\mathcal{P}$ is a set of integers that is an interval in each permutation of $\mathcal{P}$.

The set $\{1, 2, \ldots, n\}$ and all singletons are always common intervals of any non empty set of permutations, they are called *trivial* intervals. In the sequel, we assume, without loss of generality, that the set $\mathcal{P}$ contains the identity permutation Id_n. A common interval of $\mathcal{P}$ can thus be denoted as an interval $(i..j)$ of the identity permutation.

Definition 2. Let $\mathcal{P} = \{Id_n, P_2, \ldots, P_K\}$ be a set of K permutations on n elements. A *generator* for the common intervals of $\mathcal{P}$ is a pair (R, L) of vectors of size n such that:
1. $R[i] \geq i$ and $L[j] \leq j$ for all $i, j \in \{1, 2, \ldots, n\}$,
2. $(i..j)$ is a common interval of $\mathcal{P}$ if and only if $(i..j) = (i..R[i]) \cap (L[j]..j)$.

The following proposition shows how to construct a generator for a union of sets of permutations, given generators for each set. If X and Y are two vectors, we denote by $\min(X, Y)$ the vector $\min(X[1], Y[1]), \ldots, \min(X[n], Y[n])$.

Proposition 1. *Let (R_1, L_1) and (R_2, L_2) be generators for the common intervals of two sets $\mathcal{P}_1$ and $\mathcal{P}_2$ of permutations, both containing the identity permutation. The pair $(\min(R_1, R_2), \max(L_1, L_2))$ is a generator for the common intervals of $\mathcal{P}_1 \cup \mathcal{P}_2$.*

Proof. First note that $(i..j) = (i..R[i]) \cap (L[j]..j)$ if and only if $L[j] \leq i \leq j \leq R[i]$. Interval $(i..j)$ is a common interval of $\mathcal{P}_1 \cup \mathcal{P}_2$ if and only if it is a common interval of both $\mathcal{P}_1$ and $\mathcal{P}_2$, which is equivalent to $L_1[j] \leq i \leq j \leq R_1[i]$ and $L_2[j] \leq i \leq j \leq R_2[i]$, and finally to $\max(L_1[j], L_2[j]) \leq i \leq j \leq \min(R_1[i], R_2[i])$ $\square$

Proposition 1 implies that, given an $O(n)$ algorithm for computing generators for the common intervals of two permutations, we can easily deduce an $O(Kn)$ algorithm for computing a generator for the common intervals of K permutations.

Generators are far from unique, but some are easier to compute than others. Identifying good generators is a crucial step in the design of efficient algorithms to compute common intervals. The remaining of this section focuses on particular classes of generators that turn out to have interesting properties with respect to computations.

Definition 3. Let $P = (p_1, \ldots, p_n)$ be a permutation on n elements. For each element p_i, we define two intervals containing p_i:
$IMax[p_i]$ is the largest interval of P whose elements are all $\geq p_i$,
$IMin[p_i]$ is the largest interval of P whose elements are all $\leq p_i$.
And the following two integers:
$Sup[p_i]$ is the largest integer such that $(p_i..Sup[p_i]) \subseteq IMax[p_i]$,
$Inf[p_i]$ is the smallest integer such that $(Inf[p_i]..p_i) \subseteq IMin[p_i]$.

Remark that $(p_i..Sup[p_i])$ and $(Inf[p_i]..p_i)$ are intervals of the identity permutation, but not necessarily intervals of permutation P. For example, if $P = (1\ 4\ 7\ 5\ 9\ 6\ 2\ 3\ 8)$, we have: $IMax[5] = (7\ 5\ 9\ 6)$ and $Sup[5] = 7$, and $IMin[8] = (6\ 2\ 3\ 8)$ and $Inf[8] = 8$.

Proposition 2. *The pair of vectors (Sup, Inf) is a generator for the common intervals of P and Id_n.*

Proof. Suppose that $(i..j)$ is a common interval of P and Id_n, then $Sup[i] \geq j$ and $Inf[j] \leq i$ since all elements in the set $(i..j)$ are consecutive in permutation P. Thus $(i..j) = (i..Sup[i]) \cap (Inf[j]..j)$. On the other hand, suppose that $Sup[i] \geq j$ and $Inf[j] \leq i$, then $IMax[i]$ contains j and $IMin[j]$ contains i. Since both $IMax[i]$ and $IMin[j]$ are intervals of P, their intersection is an interval and is equal to $(i..j)$. □

Algorithm 1: Computing the generator (Sup, Inf).

```
Inf[1] ← 1, Sup[n] ← n.
For k from 1 to n, m[k] ← k, M[k] ← k.
For k from 2 to n
    While m[k] − 1 is in IMin[k], m[k] ← m[m[k] − 1]
    Inf[k] ← m[k]
For k from n − 1 to 1
    While M[k] + 1 is in IMax[k], M[k] ← M[M[k] + 1]
    Sup[k] ← M[k]
```

Proposition 3. *Let P be a permutation on n elements. If the bounds of intervals $IMax[k]$ and $IMin[k]$ are known for all k, then Algorithm 1 computes (Sup, Inf) in $O(n)$ time.*

Proof. We first show that Algorithm 1 is correct. Suppose that, at the beginning of the k-th iteration of the second *For* loop, $Inf[k'] = m[k']$ for all $k' < k$, and $m[k] \in IMin[k]$. This is the case at the beginning of iteration $k = 2$, since $Inf[1] = 1$. By definition, $Inf[k] \leq k$, thus before entering the *While* loop, we have $Inf[k] \leq m[k]$. If the test $m[k] - 1 \in IMin[k]$ of the *While* loop is true, then $Inf[k] \leq m[k] - 1$, implying $Inf[k] \leq Inf[m[k] - 1]$. Since $Inf[m[k] - 1] = m[m[k] - 1]$ by hypothesis, the instruction in the *While* loop preserves the invariant $Inf[k] \leq m[k]$. When the test of the *While* loop becomes false, then $Inf[k]$ is greater than $m[k] - 1$, thus $Inf[k] = m[k]$. The proof of correctness for Sup is similar.

Suppose that $IMin[k] = [i, j]$, then the tests in the *While* loops can be done in constant time using the inverse of permutation P. The total time complexity follows from the fact that the instruction within the *While* loop is executed exactly $n - 1$ times. Indeed, consider, at any point of the execution of the algorithm, the collection of intervals $(m[k]..k)$ of the identity permutation that are not contained in any other interval of this type. After the initialization loop, we have n such intervals, and at the completion of the algorithm, there is only one, namely $(1..n)$, since $Inf[n] = 1$. The instruction in the *While* loop merges two consecutive intervals into one and there can be at most $n - 1$ of these merges. □

The computation of the bounds of intervals $IMax[k]$ and $IMin[k]$, as well as the computation of the inverse of permutation P, are quite straightforward. As an example, Algorithm 2 shows how to compute the left bound of $IMax[p_i]$.

Proposition 4. *Let $P = (p_1, \ldots, p_n)$ be a permutation on n elements, Algorithm 2 computes the left bound of all intervals $IMax[p_i]$ in $O(n)$ time.*

Proof. The time complexity of Algorithm 2 is immediate since each position is stacked once. The correctness of $LMax$ relies on the fact that, at the beginning of the i-th iteration, the position j of the nearest left element such that $p_j < p_i$ must be in the stack. If it was not the case, then an element smaller than p_j was found between the positions j and i, contradicting the definition of position j □

Algorithm 2: Computing the left bound $LMax[p_i]$ of $IMax[p_i]$ for all p_i

S is a stack of positions; s denotes the top of S.
Push 0 on S
$p_0 \leftarrow 0$
For i from 1 to n
    While $p_i < p_s$ Pop the top of S
    $LMax[p_i] \leftarrow s + 1$
    Push i on S

To summarize the results of this section, we have:

Theorem 1. *Let $\mathcal{P} = \{Id_n, P_2, \ldots, P_K\}$ be a set of K permutations on n elements. A generator for the common intervals of $\mathcal{P}$ can be computed in $O(Kn)$ time.*

3 Common intervals of K permutations in optimal time

We now turn to the problem of generating all common intervals of K permutations in $O(N)$ time, where N is the number of such common intervals, given a generator satisfying the following property.

Definition 4. Two sets A and B *commute* if either $A \subseteq B$, or $B \subseteq A$, or A and B are disjoint, and otherwise they *overlap*. A collection $\mathcal{C}$ of sets is *commuting* if, for any pair of sets A and B in $\mathcal{C}$, A and B commute. A generator (R, L) for the common intervals of $\mathcal{P} = \{Id_n, P_2, \ldots, P_K\}$ is *commuting* if both the collections $\{(i..R[i])\}_{i\in(1..n)}$ and $\{(L[i]..i)\}_{i\in(1..n)}$ are commuting. If (R, L) is a commuting generator, we define $Support[i]$, for $i > 1$, to be the greatest integer $j < i$ such that $R[i] \leq R[j]$.

It turns out that generators defined in Section 2 are commuting. Indeed, generators defined in Proposition 1 are commuting if they are constructed with generators (R_1, L_1) and (R_2, L_2) that are commuting. This is a consequence of the fact that if $a < b$ and $a' < b'$ then $\min(a, a') < \min(b, b')$ and $\max(b, b') > \max(a, a')$. For the generator (Sup, Inf), we have:

Proposition 5. *The generator (Sup, Inf) for the common intervals of permutations P and Id_n is commuting.*

Proposition 6. *Given a commuting generator (R, L), Algorithm 3 computes the values $Support[i]$, for all $i > 1$, in linear time.*

Theorem 2. *Given a commuting generator (R, L), Algorithm 4 outputs all common intervals of a set $\mathcal{P}$ of K permutations on n elements, in $O(n+N)$ time, where N is the number of common intervals of the set $\mathcal{P}$.*

Proof. The time complexity of Algorithm 4 is immediate. Suppose that interval $(i..j)$ is identified by the algorithm. At the start of the j-th iteration of the *For* loop, $i = j$, thus $j \leq R[i]$. If the test of the *While* loop is true, then $i \geq L[j]$, and $(i..j)$ is a common interval. If $i' = Support[i]$, then $R[i'] \geq R[i]$, thus $j \leq R[i']$ at the end of the *While* loop.

On the other hand, if $(i..j)$ is a common interval of $\mathcal{P}$, with $i < j$, then $Support[j] \geq i$, since $R[i] \geq R[j]$. Let i' be the smallest integer such that $i < i'$ and $(i'..j)$ is identified by Algorithm 4 as a common interval. Such an interval exists, since $(j..j)$ is a common interval. Finally, $Support[i'] = i$ since $Support[i']$ must be greater than or equal to i. If it is greater, then $(Support[i']..j)$ is a common interval, contradicting the definition of i'. □

Algorithm 3: Computing $Support[i]$ for a commuting generator (R, L)

```
S is an empty stack; s denotes the top of S
Push 1 on S
For i from 2 to n
    While R[s] < i Pop the top of S
    Support[i] ← s
    Push i on S
```

Algorithm 4: Common intervals of a set $\mathcal{P}$ given a generator (R, L)

```
For j from n to 1
    i ← j
    While i ≥ L[j]
        Output (i..j) (* Interval (i..j) is a common interval of the set P *)
        i ← Support[i]
```

4 Canonical representations of closed families

The common intervals of a set of permutations is an example of a more general families of intervals, the *closed* families. In this section, we develop a new canonical representation for such families, based on the generators of the previous section.

A *closed* family $\mathcal{F}$ of intervals of the identity permutation on n elements is a family that contains all singletons, the interval $(1..n)$ and that has the following property: if $(i..k)$ and $(j..l)$ are in $\mathcal{F}$, and $i \leq j \leq k \leq l$, then $(i..j)$, $(j..k)$, $(k..l)$ and $(i..l)$ belong to $\mathcal{F}$. It is easy to extend Definition 2 of generators to the more general case of closed families.

A classical result [3] establishes a bijection between the *PQ*-trees with n leaves and closed families of Id_n, thus allowing a representation of size $O(n)$ for any closed family. Among all possible generators, the following ones will also provide a representation of size $O(n)$ for any closed family:

Definition 5. *A generator* (R, L) *for a closed family* $\mathcal{F}$ *is* canonical *if, for all* $i \in (1..n)$*, intervals* $(i..R[i])$ *and* $(L[i]..i)$ *belong to* $\mathcal{F}$*.*

Proposition 7. *Let* $\mathcal{F}$ *be a closed family. The canonical generator of* $\mathcal{F}$ *always exists, and it is unique and commuting.*

Theorem 3. *Given a commuting generator* (R', L')*, Algorithm 5 computes the canonical generator* (R, L) *of a closed family* $\mathcal{F}$ *in* $O(n)$ *time.*

Algorithm 5: Canonical generator (R, L) *given a commuting generator* (R', L')

```
The vector Support is obtained from R' using Algorithm 3
R[1] ← n
For k from 2 to n
    R[k] ← k
For k from n to 2
    If (Support[k]..R[k]) ∈ F
        R[Support[k]] ← max(R[k], R[Support[k]])
(* Computation of L is similar, by defining the vector Support with respect
to L' *)
```

Example 1. Let $\mathcal{P} = \{Id_8, P_2\}$ and $\mathcal{Q} = \{Id_8, P_3\}$ with

$$Id_8 = (1, 2, 3, 4, 5, 6, 7, 8) \quad P_2 = (1, 3, 2, 4, 5, 7, 6, 8) \quad P_3 = (2, 8, 3, 4, 5, 6, 1, 7)$$

Figure 1 shows the generators (Sup, Inf) for the common intervals of $\mathcal{P}$ and the common intervals of $\mathcal{Q}$; a generator for the common intervals of the union $\mathcal{P} \cup \mathcal{Q}$ using the two generators (Sup, Inf); and the canonical generator for the common intervals of $\mathcal{P} \cup \mathcal{Q}$.

Compared to PQ-trees, this canonical representation of a closed family $\mathcal{F}$ is much simpler since it uses only two arrays. Moreover, some operations, for example testing whether an interval $(i..j)$ belongs to the family $\mathcal{F}$, are also simpler using this representation. However, PQ-trees have the advantage of being recursive structures. Thus, in order to be complete, we next show how to transform one representation into the other using the key notion of *strong intervals*.

5 From canonical generators to strong intervals

A *strong interval* of $\mathcal{F}$ is an interval that commutes with each interval of $\mathcal{F}$. In this section, we show how to compute the strong intervals, given a canonical generator. The structure of *PQ-tree*, the inclusion tree of the strong intervals, is another classical representation of an interval family [3]. This concept is investigated in [12], where it is explained how to compute the PQ-tree from the strong intervals.

Let (R, L) be the canonical generator. Consider the $4n$ bounds of intervals of the families $(i..R[i])$ and $(L[j]..j)$ for $i, j \in (1..n)$. Let $(a_1, \ldots, a_{4n})$ be the list of

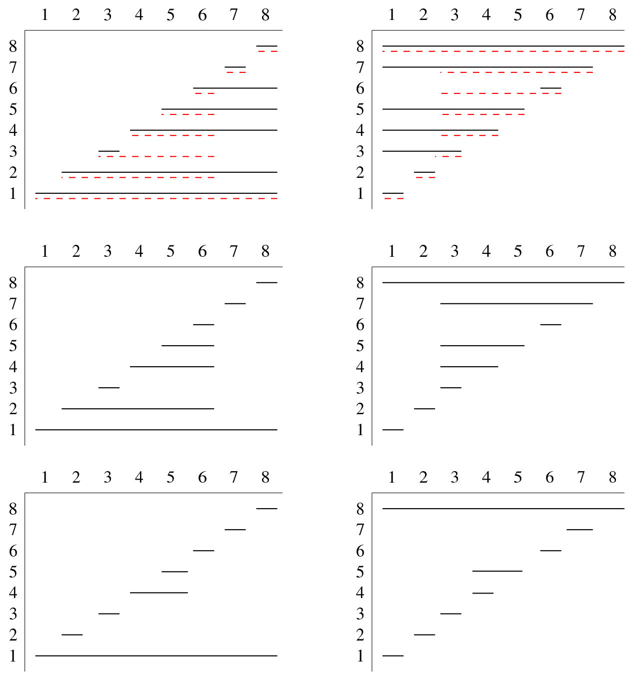

Fig. 1. The top two diagrams show the generators (Sup, Inf) of the common intervals of the set $\mathcal{P}$ in solid lines, and of set $\mathcal{Q}$ in dashed lines. A line in row i of the left diagram extends from column i to column $Sup(i)$, and a line in row i of the right diagram extends from column $Inf(i)$ to column i. The middle diagrams shows a generator for the common intervals of $\mathcal{P} \cup \mathcal{Q}$ constructed using Proposition 1. Finally, the bottom diagrams shows the canonical generator constructed by Algorithm 5.

these $4n$ bounds sorted in increasing order, with the left bounds placed before the right bounds when they are equal. For the example of Figure 1 this list is

$$(1, 1, 1, \overline{1}, 2, 2, \overline{2}, \overline{2}, 3, 3, \overline{3}, \overline{3}, 4, 4, 4, \overline{4}, 5, 5, \overline{5}, \overline{5}, 6, 6, \overline{6}, \overline{6}, 7, 7, \overline{7}, \overline{7}, 8, \overline{8}, \overline{8}, \overline{8})$$

where $\overline{i}$ denotes a right bound. Such a list can be constructed easily by scanning the two vectors R and L, and by noting that each $i \in (1..n)$ is a left bound at least once, and a right bound at least once.

Proposition 8. *Given the ordered list $(a_1, \ldots, a_{4n})$ of the $4n$ bounds of a canonical generator (R, L), Algorithm 6 outputs the strong intervals of a closed family in $O(n)$ time.*

Algorithm 6: Computation of the strong intervals

```
S is a stack of bounds, s denotes the top of S
For i from 1 to 4n
    If a_i is a left bound
        Push a_i on S
    Else
        Output (s..a_i) (* Interval (s..a_i) is strong *)
        Pop the top of S
```

6 Modular decomposition

Let $G = (V, E)$ be a directed, finite, loopless graph, with $|V| = n$ and $|E| = m$. Undirected graphs may be seen as symmetrical directed graphs in this context. A *module* is a subset M of V that behaves like a single vertex: for $x \notin M$ either there are $|M|$ arcs that join x to all vertices of M, or no arc joins x to M, and conversely either there are $|M|$ arcs that join all vertices of M to x, or no arc joins M to x. A *strong* module does not overlap any other module. There may be up to 2^n modules in a graph (in the complete graph for instance) but there are at most $O(n)$ strong modules, and the *modular decomposition tree* based on the strong modules inclusion tree is sufficient to represent all modules [15]. The modular decomposition tree is indeed the PQ-tree of the family of modules.

Linear-time decomposition algorithms have been discovered [6, 13] but remain rather complex. Simpler algorithms work in two steps: computing a factorizing permutation, and then building a tree representation on it. The first step was published in [9]. In this paper, we simplify the second step.

A *factorizing permutation* of a graph [5] is a permutation of the vertices of the graph in which every strong module of the graph is a *factor*, that is an interval of the permutation. Since the strong modules are a commuting family, every graph admits a factorizing permutation. A factorizing permutation of a graph can be computed in linear time [9]. In the following we assume, without loss of generality, that the vertex-set V is the set $\{1, ..n\}$ and that the identity permutation is a factorizing permutation of the graph.

Given an interval $(u..v)$ of the factorizing permutation, a vertex $x \notin (u..v)$ is a *splitter* of the interval if there are between 1 and $v - u$ arcs going from x to $(u..v)$, or if there are between 1 and $v - u$ arcs going from $(u..v)$ to x. A *right-module* is an interval $(u..v)$ with no splitters greater than v. A *left-module* is an interval $(u..v)$ with no splitters smaller than u. An *interval-module* is an interval $(u..v)$ with no splitters. Clearly interval-modules are modules. However, some modules are not interval-modules, but, according to the definition of a factorizing permutation, the strong modules of the graph are interval-modules. It is well known that modules behave like intervals: unions, intersections or differences of two overlapping modules are modules. Thus:

Proposition 9. [15] *The interval-modules of a factorizing permutation of a graph G are a closed family. The strong intervals of this family are exactly the strong modules of the graph G.*

Definition 6. For a vertex v let $R[v]$ be the greatest integer such that $(v..R[v])$ is a left-module and $L[v]$ the smallest integer such that $(L[v]..v)$ is a right-module.

It can be proved that for every $w \in (L[v]..v)$, $(w..v)$ is a right-module, and for every $w < L[v]$, $(w..v)$ is not a right-module. For this reason $(L[v]..v)$ is called *the* maximal right-module ending at v. In a similar way, we can define the maximal left-module beginning at v. We have:

Proposition 10. *The pair (R, L) is a commuting generator of the interval-modules family.*

Proof. Interval $(u..v)$ is an interval-module if and only if $R[u] \geq v$ and $L[v] \leq u$, thus (R, L) is a generator. The family defined by R is commuting because if $(u..R[u])$ overlaps $(v..R[v])$, and if, without loss of generality, $u < v$, then $(u..R[v])$ is a left-module starting at u greater than the maximal left-module $(u..R[u])$, which is a contradiction. A similar argument shows that L also is commuting. $\square$

In order to compute the maximal right-strong modules, we use a simplified version of an algorithm due to Capelle and Habib [5]. The algorithm to compute the maximal left-modules is similar.

Let us consider the maximal right-module $(L[v]..v)$ ending at v. If $L[v] > 1$, then there exists an $x > v$ that splits $(L[v] - 1..v)$, otherwise this right-module would not be maximal, and x therefore splits $(L[v] - 1..L[v])$, but does not split $(y - 1, y)$ for all $L[v] < y \leq v$. Based on this observation, Capelle and Habib algorithm proceeds in two steps. First, for every vertex v the *rightmost splitter* $s[v]$ is computed. It is the greatest vertex, if any, that splits the pair $(v - 1..v)$. Then a loop for v from n to 2 computes all the maximal right-modules $(L[x]..x)$ such that $v = L[x]$. Computing $s[v]$ can be done by a simultaneous scan of the adjacency lists of v and $v-1$: the greatest element occurring in only one adjacency list is kept. This can be done in time proportional to the size of the adjacency lists. The computation of $s[v]$ for all v can therefore be done in $O(n + m)$ time, that is linear in the size of the graph. The second step is Algorithm 7. It clearly runs in $O(n)$ time, and its correctness relies on the following invariant:

Invariant *At step v, for all vertices x in the stack, $(v..x)$ is a right-module, and for all $x > v$ not in the stack, $L[v] > v$.*

Proof. The invariant is initially true. Every step maintains it: if $s[v]$ does not exist then for all x in the stack $(v-1..x)$ is a right-module, and $(v-1..v)$ also is a right-module. And if $s[v]$ exists, (v) is the maximal right-module ending at v. For all $x < s[v]$ $(v - 1..x)$ is not a right-module and $(v..x)$ is therefore the maximal right-module ending at x. For all $x \geq s[v]$ $(v - 1..x)$ is still a right-module, because $s[v]$ is the greatest of the splitters of $(v - 1, v)$. $\square$

We thus have:

Theorem 4. *Given a graph G, and a factorizing permutation of G, it is possible to compute the modular decomposition tree of G in time $O(n + m)$.*

Algorithm 7: Computing all maximal right-modules given $s[v]$

```
S is a stack of vertices; t denotes the top of S.
for v from n to 2
    if s[v] exists
        L[v] ← v
        While t < s[v]
            L[t] ← v
            Pop the top of S
    else
        Push v on S
```

7 Conclusion

In the present work, we formalized two concepts about common intervals, namely generators and canonical representation, that proved to have important algorithmic implications. Indeed, the combinatorial properties of these objects, and in particular the different links between them, are central in the design and the analysis of the simple optimal algorithms for computing common intervals of permutations we presented. It is important to highlight that our algorithms are really "optimal" since they are based on very elementary manipulations of stacks and arrays. This is, we believe, a significant improvement over the existing algorithms that are based on intricate data structures, both in terms of ease of implementation and time efficiency, and in terms of understanding the underlying concepts [10, 16].

Moreover, we showed how, transposed in the more general context of modular decomposition of graphs, our results have a similar impact and lead to a significant simplification of some existing algorithms. Indeed, modular decomposition algorithms are quite complex algorithms, but using the simple factorizing permutation algorithm of [9] and the right-modules identification algorithm of Section 6, a generator of the interval-modules can easily be computed in linear time; tools from Section 5 can then be used to compute the strong interval-modules, that also are the strong modules, and the PQ-tree, called *modular decomposition tree* in this context.

References

1. S. Bérard, A. Bergeron, and C. Chauve. Conserved structures in evolution scenarios. In *Comparative Genomics, RECOMB 2004 International Workshop*, vol. 3388 of *Lecture Notes in Comput. Sci.*, p. 1–15. Springer-Verlag, 2005.
2. A. Bergeron, C. Chauve, F. de Montgolfier, and M. Raffinot. Computing common intervals of K permutations, with applications to modular decomposition of graphs. LIAFA technical report 2005-006 available at `http://www.liafa.jussieu.fr/web9/rapportrech/listrapport_fr.php?anscol=2005`
3. S. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-trees algorithms. *J. Comput. Syst. Sci.*, 13:335–379. 1976.

4. B. M. Bui Xuan, M. Habib, and C. Paul, From Permutations to Graph Algorithms, LIRMM technical report RR-05021, 2005.
5. C. Capelle and M. Habib. Graph decompositions and factorizing permutations. in *Fifth Israel Symposium on Theory of Computing and Systems, ISTCS 1997*, p. 132–143. IEEE Computer Society, 1997.
6. A. Cournier and M. Habib. A new linear algorithm for modular decomposition. In *Trees in algebra and programming – CAAP'94, 19th International Colloquium*, vol. 787 of *Lecture Notes in Comput. Sci.*, p. 68–84. Springer-Verlag, 1994.
7. E. Dahlhaus, J. Gustedt, and R. M. McConnell. Efficient and practical algorithms for sequential modular decomposition. *J. Algorithms*, 41(2):360–387. 2001.
8. M. Figeac and J.-S. Varré. Sorting by reversals with common intervals. In *Algorithms in Bioinformatics, 4th International Workshop, WABI 2004*, vol. 3240 of *Lecture Notes in Comput. Sci.*, p. 26–37. Springer-Verlag, 2004.
9. M. Habib, F. de Montgolfier and C. Paul. A Simple Linear-Time Modular Decomposition Algorithm for Graphs, Using Order Extension In *Algorithm Theory – SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory*, vol. 3111 of *Lecture Notes in Comput. Sci.*, p. 187–198. Springer-Verlag, 2004.
10. S. Heber and J. Stoye. Finding all common intervals of k permutations. In *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001*, vol. 2089 of *Lecture Notes in Comput. Sci.*, p. 207–218. Springer-Verlag, 2001.
11. G.M. Landau, L. Parida and O. Weimann. Gene Proximity Analysis Across Whole Genomes via PQ Trees. 6th Combinatorial Pattern Matching Conference (CPM), 2005.
12. R. M. McConnell and F. de Montgolfier. Algebraic Operations on PQ-trees and Modular Decomposition Trees. WG'05, 31st International Workshop on Graph-Theoretic Concepts in Computer Science, 2005.
13. R. M. McConnell and J. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, p. 536–545. ACM/SIAM, 1994.
14. R. M. McConnell and J. Spinrad. Ordered vertex partitioning. *Discrete Mathematics & Theoretical Computer Science*, 4:45–60. 2000.
15. 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.
16. T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309. 2000.