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

Anne Bergeron* Cedric Chauve* Fabien de Montgolfier† Mathieu Raffinot‡

Abstract

We introduce a new approach 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 [18] in order to model the fact that, when comparing genomes, a group of genes can be rearranged but still remain connected. In [18], 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 [12] 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 [18]. 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 [13], or as guides to reconstruct evolution scenarios [1, 10].

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 also propose a new canonical representation of the family of common intervals that is simpler than the PQ-trees. We then link this work to previous studies on common intervals and show how our new representation can be transformed in linear

*LaCIM, Université du Québec à Montréal, Canada. E-mail: [anne,chauve]@lacim.uqam.ca

†LIAFA, Université Denis Diderot - Case 7014, 2 place Jussieu, F-75251 Paris Cedex 05, France. E-mail: fm@liafa.jussieu.fr

‡CNRS, Poncelet Laboratory, Independent University of Moscow, 11 street Bolchoï Vlassievski, 119 002 Moscow, Russia. E-mail: mathieu@raffinot.net

time into classical ones, namely PQ-trees, and irreducible intervals. Conservely, generating families can be linearly built from these representations.

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 [8, 15] 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 [9, 16].

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. Section 5 links this new representation to classical ones, namely strongs intervals, irreducible intervals and PQ-trees. Finally, in Section 6, we extend our results to the modular decomposition of graphs. An extended abstract of this article appeared 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 the indices of its left and right bounds $(i..j)$.

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 $\max(L_1[j], L_2[j]) \leq i \leq j \leq \min(R_1[i], R_2[i])$ □

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 $(i..j)$. □

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

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

The generators (Sup, Inf) for the common intervals of $\mathcal{P}$ and $\mathcal{Q}$ are shown on Figure 1. This figure also shows a generator for $\mathcal{P} \cup \mathcal{Q}$.

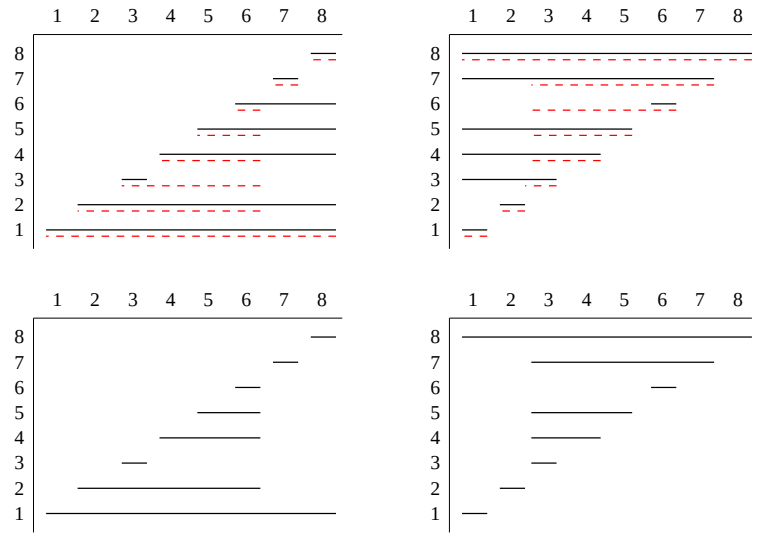

Figure 1: The top two diagrams show the generators (Sup, Inf) of the common intervals of the set $\mathcal{P}$ in solid lines, and of set $\mathcal{Q}$ in dashed lines. A line in row i of the left diagram extends from column i to column $Sup(i)$, and a line in row i of the right diagram extends from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

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

$Inf[1] \leftarrow 1, Sup[n] \leftarrow n$.
For k *from* 1 *to* n, $m[k] \leftarrow k, M[k] \leftarrow k$.
For k *from* 2 *to* n
    While $m[k]-1$ *is in* $IMin[k]$, $m[k] \leftarrow m[m[k]-1]$
    $Inf[k] \leftarrow m[k]$
For k *from* $n-1$ *to* 1
    While $M[k]+1$ *is in* $IMax[k]$, $M[k] \leftarrow M[M[k]+1]$
    $Sup[k] \leftarrow M[k]$

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.*

Proof. Suppose that $(k..Sup[k])$ contains k', we will show that it must also contain $Sup[k']$. If k' is in $(k..Sup[k])$, then k' is in $IMax[k]$, and $k' > k$, therefore, $IMax[k'] \subseteq IMax[k]$, and the interval $(k'..Sup[k'])$ is included in $IMax[k]$, thus in $(k..Sup[k'])$ also. Since $Sup[k]$ is maximal, we must have $Sup[k] \geq Sup[k']$. A similar argument holds for Inf. □

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

Proof. The time complexity of Algorithm 3 is immediate. At iteration i the stack contains the left bounds of all intervals of $(j..R[j])$ such that $R[j] \geq i$ and $j < i$, sorted in decreasing size order. It is then easy to see when equality holds. Note that $Support[1]$ is undefined and should not be used by subsequent algorithms. □

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 A new canonical representation 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 a permutation σ 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.*

Proof. Let $\mathcal{F}$ be a closed family. For $1 \leq i \leq n$, define $R[i]$ be the largest integer such that $(i..R[i]) \in \mathcal{F}$, and $L[i]$ be the smallest integer such that $(L[i]..i) \in \mathcal{F}$.

If an interval $(i..j) \in \mathcal{F}$, then $(i..j) \subseteq (i..R[i])$, and $(i..j) \subseteq (L[j]..j)$, thus (R, L) is a generator. It is canonical since we picked elements of $\mathcal{F}$. Suppose that there exists a second canonical generator (R', L'), with $R \neq R'$, then there exists $1 \leq i \leq n$ such that $R'[i] < R[i]$. Since $(i..R[i])$ is in $\mathcal{F}$, it should be generated by (R', L') but $(i..R'[i]) \cap (L'[R[i]], R[i])$ does not contain $R[i]$. A similar argument holds if $L \neq L'$. Finally, suppose that two intervals $(i..R[i])$ and $(j..R[j])$ overlap with $i < j < R[i] < R[j]$. Then $(i..R[j])$ is in $\mathcal{F}$ which contradicts the maximality of $(i..R[i])$. □

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.*

Proof. The time complexity of Algorithm 5 follows from the fact that testing if an interval belongs to $\mathcal{F}$ can be computed in $O(1)$ time with the generator (R', L'). Its correctness relies on the following observation: if $R[k] \neq k$, then there exists an integer $k' > k$ such that

$$Support[k'] = k, \text{ and } R[k'] = R[k] \tag{1}$$

where $Support[k']$ is defined (Definition 4) as the greatest integer smaller than k' such that $R'[Support[k']] \geq R'[k']$. Statement (1) implies that, since the values of $R[k]$ are computed in decreasing order, the value of $R[k]$ will be known at the start of iteration k.

In order to prove statement (1), let k' be the smallest integer such that $R[k'] = R[k]$. The hypothesis $R[k] \neq k$, and the fact that (R, L) is a commuting generator, imply that $k' > k$. We must show that $Support[k'] = k$. Since (R', L') is a commuting generator, and (R, L) is the canonical generator, we must have $R'[k] \geq R'[k']$. Thus, $Support[k'] \geq k$, implying that $R[Support[k']] \leq R[k]$. Since $R[k] = R[k']$, this would imply $R[Support[k']] = R[k]$, contradicting the definition of k'. □

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

The vector Support is obtained from R' using Algorithm 3
$R[1] \leftarrow n$
For k from 2 to n
    $R[k] \leftarrow k$
For k from n to 2
    If $(Support[k]..R[k]) \in \mathcal{F}$
        $R[Support[k]] \leftarrow \max(R[k], R[Support[k]])$
(Computation of L is similar, by defining the vector Support with respect to L' *)*

Example (continued) Let $\mathcal{P} = \{Id_8, P_2\}$ and $\mathcal{Q} = \{Id_8, P_3\}$ with

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

Two generators for the set $\mathcal{P} \cup \mathcal{Q}$ are shown on Figure 2. The second one is canonical.

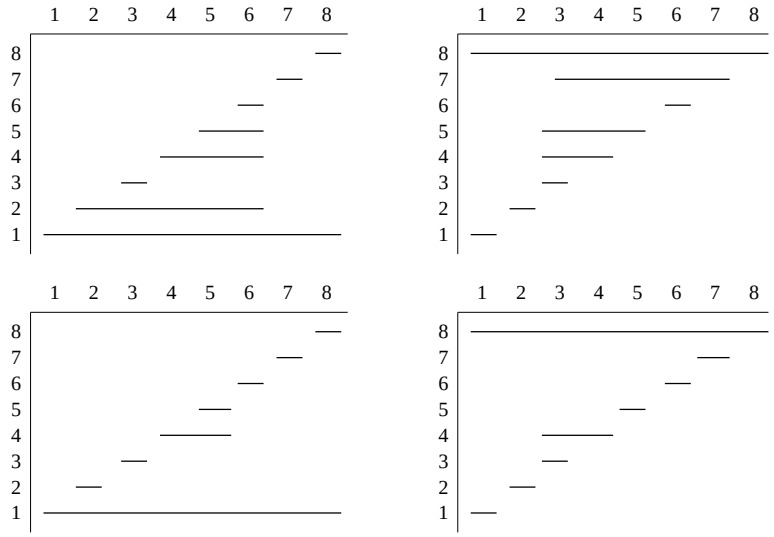

Figure 2: The top two diagrams show a generator for the common intervals of $\mathcal{P} \cup \mathcal{Q}$ constructed using Proposition 1 (see Figure 1). The bottom diagrams shows the canonical generator constructed by Algorithm 5

.

5 Computing a canonical representation from another one

Compared to PQ-trees, the canonical generator 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. Another canonical representation, the family of *irreducible intervals* was introduced by [12].

The links between PQ-trees and irreducible intervals have been studied in [13] that presents linear-time algorithms for the conversion.

In order to be complete, we next show how to transform the canonical generator into PQ-tree and vice versa.

5.1 From canonical generators to strong intervals and PQ-trees

The key notion we use is the strong common 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 [14].

Lemma 1 *Let (R, L) be the canonical generator. We have:*

(1) If $(i..R[i])$ overlaps $(L[j]..j)$ and $(L[j']..j')$ then $L[j] = L[j']$.

(2) If $(L[j]..j)$ overlaps $(i..R[i])$ and $(i'..R[i'])$ then $R[i] = R[i']$.

Proof.
(1) Let us suppose $L[j] \neq L[j']$ and w.l.o.g. $L[j] < L[j']$. Then $j > j'$ as L is commuting. As (R, L) is a generator $L[j] < i$ (because $L[j] > i$ would contradict the minimality of $L[j]$). Thus we have $L[j] < L[j'] < i \leq j' \leq j < R[i]$. Interval $(L[j]..i-1)$ is common since it is $(L[j]..j)$ minus $(i..R[i])$. Thus $(L[j]..j')$ is also common since it is the union of overlapping $(L[j]..i-1)$ and $(L[j']..j')$. This contradicts the minimality of $L[j']$.
(2) The proof is similar to Point 1 proof. □

Let us consider the canonical generator (R, L). The transitive closure of the overlap relation on $R \cup L$ is a equivalence and its equivalence classes are henceforth called *overlap classes*. A trivial overlap class only contains a single interval $(i..R[i])$ or $(L[j]..j)$. This interval is obviously strong. The following lemma 2 deals with non trivial classes.

Lemma 2 *Let $\mathcal{C}$ be a non trivial overlap class containing $(i_1..R[i_1]), ... (i_k..R[i_k])$ and $(L[j_1]..j_1), ... (L[j_l]..j_l)$. Let us suppose wlog $i_1 < ... < i_k$ and $j_1 < ... < j_l$. Then:*

(3) For all $a, b \in (1..k)$ $R[i_a] = R[i_b]$.

(4) For all $a, b \in (1..l)$ $L[j_a] = L[j_b]$. We will henceforth denote $L[\mathcal{C}]$ and $R[\mathcal{C}]$ these common bounds.

(5) $k = l$

(6) $(L[\mathcal{C}]..j_a)$ overlaps $(i_b..R[i_b])$ iff $a \geq b$.

(7) For all $a \in (1..k-1)$ $i_{a+1} = j_a + 1$.

(8) $(L[\mathcal{C}]..R[\mathcal{C}])$ is a strong common interval and for all $a \in (1..k)$ $(i_a..j_a)$ is a strong common interval.

Proof.

(3) Direct consequence of Point 2 of Lemma 1.

(4) Direct consequence of Point 1 of Lemma 1.

(5) Let us suppose $k < l$ (there is a similar proof if we suppose $k > l$). Let f be the function $(1..l) \rightarrow (1..k)$ such that $f(a)$ is the largest b such that $(L[\mathcal{C}]..j_a)$ overlaps $(i_b..R[\mathcal{C}])$. As $k < l$ there must exist x, y and z such that $f(x) = f(y) = z$. Let us suppose wlog $j_x < j_y$. Interval $(j_x + 1..R[\mathcal{C}])$ is common, since it is the difference $(i_z..R[\mathcal{C}])$ minus $(L[\mathcal{C}]..j_x)$, and overlaps $(L[\mathcal{C}]..j_y)$. Furthermore $R[j_x + 1] = R[\mathcal{C}]$ since a larger $R[j_x + 1]$ would contradict maximality of $R[i_z] = R[\mathcal{C}]$. $(j_x + 1..R[j_x + 1])$ is hence in the overlap class $\mathcal{C}$ and that contradicts $f(y) = z$.

(6) We have indeed proved that f is a bijection. As $i_1 < ... < i_k$ and $j_1 < ... < j_l$, f is an increasing function and thus for all $a \in (1..k)$ $f(a) = a$. Point 6 directly derives.

(7) For all $a \in (i..k-1)$ $(L[\mathcal{C}]..j_a)$ overlaps $(i_a..R[\mathcal{C}])$ but not $(i_{a+1}..R[\mathcal{C}])$ which is overlapped by $(L[\mathcal{C}]..j_{a+1})$. $(j_a+1..R[\mathcal{C}])$ is a common interval (it is $(i_a..R[\mathcal{C}])$ minus $(L[\mathcal{C}]..j_a)$). We have $R[j_a+1] = R[\mathcal{C}]$ (a larger $R[j_a+1]$ would contradict maximality of $R[i_a] = R[\mathcal{C}]$) and $(j_a+1..R[\mathcal{C}]) \in \mathcal{C}$ (it overlaps $(L[\mathcal{C}..j_{a+1})$). As said in proof of Point 5 $f(i_{a+1}) = j_{a+1}$ hence $i_{a+1} = j_a + 1$.

(8) As $(L[\mathcal{C}]..R[\mathcal{C}])$ is the union of overlapping common intervals $(L[j_1]..j_1)$ and $(i_1..R[i_1])$ it is a common interval. As $(i_a..j_a)$ is the intersection of overlapping common intervals $(L[j_a]..j_a)$ and $(i_a..R[i_a])$ it is also a common interval.

Let us suppose an interval $(i..j)$ overlaps $(L[\mathcal{C}]..R[\mathcal{C}])$ and wlog $i < (L[\mathcal{C}] \leq j < R[\mathcal{C}]$. We have $L[j] \leq i < L[\mathcal{C}]$ and $(L[j]..j)$ overlaps $(i_1..R[i_1] = R[\mathcal{C}])$ in violation of Point 1 of Lemma 1, contradiction.

Let us suppose an interval $(i..j)$ overlaps $(i_a..j_a)$ wlog $i < i_a \leq j < j_a$. $(i..R[\mathcal{C}])$ is a common interval (union of $(i..j)$ and $(i_a..R[\mathcal{C}])$) of $\mathcal{C}$ (it overlaps $(L[\mathcal{C}]..j_a)$) not repertoried in $\mathcal{C}$ since $i \notin \{i_1..i_k\}$, a contradiction.

□

Note that if an overlap class $\mathcal{C}$ is not trivial then $(L[\mathcal{C}]..R[\mathcal{C}])$ is not an interval of (R, L). These intervals however form a commuting family since:

Lemma 3 *Let $\mathcal{C}$ and $\mathcal{C}'$ be two overlap classes of $\mathcal{G}(R, L)$ then:*

(9) $(L[\mathcal{C}]..R[\mathcal{C}])$ and $(L[\mathcal{C}']..R[\mathcal{C}'])$ commute.

Proof. If $\mathcal{C}$ is a trivial class let $I = J$ be the single member of this class. Else let I and J be two overlapping intervals. According to Point 3 and 4 of Lemma 2 $L[\mathcal{C}]$ is the left bound of one of them (wlog say I) and $R[\mathcal{C}]$ is the right bound of the other (J). Let I' be an interval of $\mathcal{C}'$. Either $I \subset I'$ and $J \subset I'$ and so $(L[\mathcal{C}]..R[\mathcal{C}]) \subset (L[\mathcal{C}']..R[\mathcal{C}'])$; or $I' \subset I$ and $I' \subset J$ and so $(L[\mathcal{C}']..R[\mathcal{C}']) \subset (L[\mathcal{C}]..R[\mathcal{C}])$; or $I \cap I' = \emptyset$ and $J \cap I' = \emptyset$ and so $(L[\mathcal{C}]..R[\mathcal{C}]) \cap (L[\mathcal{C}']..R[\mathcal{C}']) = \emptyset$. □

Lemma 4 *Let S be a strong interval of $\mathcal{F}$. There exists an overlap class $\mathcal{C}$ such that either $S = (L[\mathcal{C}]..R[\mathcal{C}])$ or $\mathcal{C}$ contains k intervals of R $(i_1..R[i_1]), ...\ (i_k..R[i_k])$ and k intervals of L $(L[j_1]..j_1), ...\ (L[j_k]..j_k)$ and there exists $a \in (1..k)$ such that $S = (i_a..j_a)$.*

Proof. Let $S = (i..j)$ be a strong interval. If $R[i] = j$ then $S \in R$ and S form a trivial overlap class. Same situation if $L[j] = i$. Otherwise $(i..R[i])$ and $(L[j]..j)$ overlap and thus belong to an overlap class $\mathcal{C}$. Using the same notations as above we have $i = i_a$ and $j = j_b$ for some a and b. If $a > b$ then according to Point 6 of Lemma 2 $(L[\mathcal{C}]..j_b)$ does not overlap $(i_a..R[i_a])$. If $a < b$, S is overlapped by $(i_b..R[\mathcal{C}])$ and thus is not strong. Therefore $a = b$. □

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 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 2 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.

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
```

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.*

Proof. The $O(n)$ time complexity is obvious. Let us prove its correction.

1. We shall prove that every interval output by Algorithm 6 is strong. According to Lemma 3, the overlap component commute. At any step the top of the stack contains the left bounds of an overlap class $\mathcal{C}$ and the rest of the stack contains the left bounds of the overlap class containing $\mathcal{C}$, ordered by the inclusion.

According to Point 5 of Lemma 2 each overlap class contains as many intervals $(L[j_a]..j_a)$ as intervals $(i_b..R[i_b])$. So the algorithm outputs only intervals whose bounds belong to the same overlap class.

If the overlap class only contains a strong interval, clearly this interval and only it is output. Assume now that this class is non trivial. We show now that these intervals are exactly those described in Point 8 of Lemma 2. We just have to notice that there are k pushes of $L[\mathcal{C}]$, then one of i_1; then for each $a \in (1..k-1)$ we repeat a pop after j_a outputing $(i_a..j_a)$ followed by a push of i_{a+1}. Finally there a pop after j_k outputing $(i_k..j_k)$, then k pops after $R[\mathcal{C}]$ outputing $(L[\mathcal{C}]..R[\mathcal{C}])$ k times. The algorithm outputs therefore only strong intervals.

2. Conversely, according to Lemma 4 every strong interval is as described by Point 8 of Lemma 2, so is output. □

Note that, since $2n$ intervals are identified by Algorithm 6, and the number of strong intervals is between $n+1$ and $2n-1$, some of them may be output several times.

5.2 From strong intervals and generator to PQ-tree

Let T be a tree whose n leaves are labelled with n different labels. The *frontier* of a node is the set of labels of the leaves of the subtree rooted at this node.

A *proper commuting family* $\mathcal{F} \subset 2^V$ is a commuting family such that $V \in \mathcal{F}$, $\forall\, v \in V\ \{v\} \in \mathcal{F}$ and $\emptyset \notin \mathcal{F}$. The strong common intervals of K permutations obviously form a proper commuting family.

A proper commuting family $\mathcal{F}$ can be represented by its *inclusion tree* in which the frontiers of the nodes are in bijection with the members of the family. Clearly its root is V and its leaves are the singletons of $\mathcal{F}$. Conversely the frontiers of a tree with V as leaf labels define a proper commuting family.

Given a proper commuting family of $n \leq m < 2n$ intervals, the following folklore algorithm computes the inclusion tree in $O(n)$ time.

Algorithm 7: Building the inclusion tree of a proper commuting family $\mathcal{F}$

Bucket-sort in decreasing order the intervals of $\mathcal{F}$ according to their right bound
Bucket-sort in increasing order the intervals of $\mathcal{F}$ according to their first bound
Let $I_1..I_m$ be the list of sorted intervals
$F \leftarrow I_1$ (* $I_1 = V$ *is the root* *)
$k \leftarrow 2$
While $k \leq m$
    If $I_k \subset F$
        $Parent(I_k) \leftarrow F$
        $F \leftarrow I_k$
        $k \leftarrow k + 1$
    else
        $F \leftarrow Parent(F)$

Notice that the bucket sort is a stable linear-time sorting algorithm, thus intervals with the same left bound are sorted according to their right bounds. The overall time complexity is obviously $O(n)$.

The inclusion trees on V are in bijection with the proper commuting families. Similarly the closed families are in bijection with a family of tree, called *PQ-tree*, that can store them in $O(n)$ size.

$$\begin{array}{ccc} \text{Proper commuting family} & \Longleftrightarrow & \text{inclusion tree} \\ \downarrow & & \downarrow \\ \text{Closed family} & \Longleftrightarrow & PQ\text{-tree} \end{array}$$

Definition 6 *A PQ-*tree *is a tree whose leaves are labeled from 1 to n, whose internal nodes are labeled P-nodes or Q-nodes, such that a P-node has at least two children, such that a Q-node has at least three children, and such that the children of a Q-node are totally ordered.*

An *extended frontier* of a *PQ*-tree is either the frontier of a P node, or the union of frontiers of consecutive children of a Q node, or a singleton.

Proposition 9 *[3, 7] Given a closed family, there exists a PQ-tree such that the intervals of the family are exactly the extended frontiers. The strong intervals of the family are exactly the frontiers of this tree. Furthermore the PQ-tree is unique up to Q node reversals.*

Let $I_1 = (l_1..r_1), ...I_k = (l_k..r_k)$ the children of a Q node. For every i, $I_i \cup I_{i+1}$ is an interval of the family. I_i and I_{i+1} are disjoint, thus $r_i + 1 = l_{i+1}$ or $r_{i+1} = l_i + 1$. The *canonical PQ*-tree is the one where all Q nodes are sorted in increasing order, ie $r_i + 1 = l_{i+1}$.

Example. Let $\mathcal{P}_2 = \{Id_9, P_4, P_5, P_6\}$ with

$$\begin{array}{lcllcl} Id_9 & = & (1,2,3,4,5,6,7,8,9) & P_5 & = & (6,5,7,8,9,1,2,3,4) \\ P_4 & = & (9,8,7,5,6,4,3,2,1) & P_6 & = & (1,3,2,4,5,6,9,8,7) \end{array}$$

Figure 3 shows the canonical generator and the corresponding *PQ*-tree of the set $\mathcal{P}_2$. Algorithm 6 produces the stack $(1,1,1,1,\overline{1},2,2,2,\overline{2},3,\overline{3},\overline{3},\overline{3}\,4,\overline{4},\overline{4},5,5,5,\overline{5},6,\overline{6},\overline{6},\overline{6},7,7,7,\overline{7},8,\overline{8},9,\overline{9},\overline{9},\overline{9},\overline{9})$ and ouputs the set of strong intervals: (1..1), (2..2), (3..3), (2..3), (2..3), (4..4), (1..4), (5..5), (6..6), (5..6), (5..6), (7..7), (8..8), (9..9), (7..9), (7..9), (1..9). These strong intervals are the backbone of the *PQ*-tree.

Thus, given the inclusion tree of the strong intervals of a closed family $\mathcal{F}$, a *PQ*-tree of $\mathcal{F}$ can be built by labelling P or Q the internal nodes of the inclusion tree and by ordering the children of the Q nodes. Fortunately, Algorithm 7 directly orders the children of every node (including the upcoming Q nodes) by increasing values of the first bounds of their frontiers. The resulting *PQ*-tree is therefore the canonical one. We just have to label P or Q the resulting nodes.

No internal node of the inclusion tree has a single child, since the node and its child would have the same frontier. Every node with 2 children is labelled P. To test wether a node with at

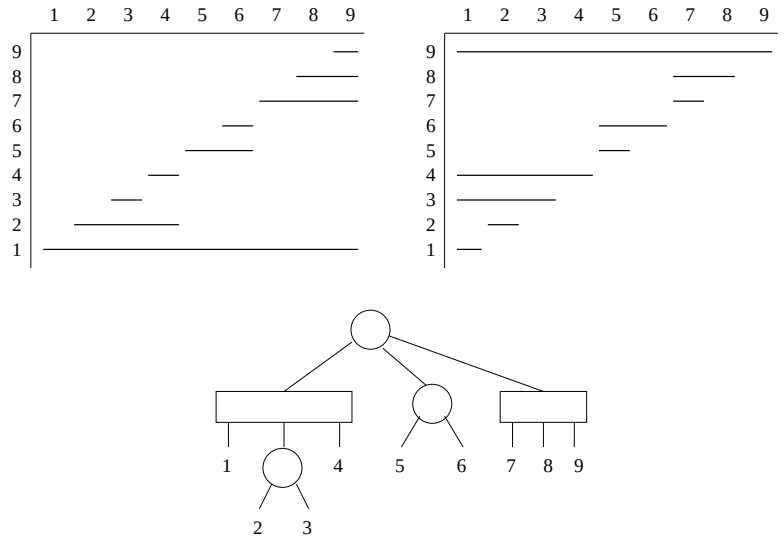

Figure 3: Example of the canonical generator and the associated PQ-tree for the set of permutations $P_2 = \{Id_9, P_4, P_5, P_6\}$.

least three children is P or Q, it suffice to probe its two first children: if their union is an interval of the family the node is labelled Q, otherwise it is labelled P. This can be done in $O(1)$ time per node, using the generator.

5.3 From PQ-tree to canonical generator

Given a PQ-tree T, the $\sigma(T)$ is a permutation of the leaves obtained by a left-to-right traversal of the tree. We can assume $\sigma(T) = Id_n$ by renaming the leaves of T. In this section we explain how to compute the canonical generator of the closed family represented by T.

Let N be a node with children $I_1..I_k$. Let l_i and r_i be the indices of the left and right bounds of I_i. Let $imin(N)$ and $imax(N)$ be the indice respectively of the minimum element of N and of the maximal element of N. Computing $imin$ and $imax$ can be done in $O(n)$ by a simple bottom-up traversal.

Algorithm 8: Computing the canonical generator from a PQ-tree.

For each internal node N taken bottom-up
  If N is a Q node
    For i from 1 to k
      $L[imax(I_i)] \leftarrow imin(N)$
      $R[imin(I_i)] \leftarrow imax(N)$
  Else
    $R[imin(I_1)] \leftarrow imax(N)$
    $L[imax(I_1)] \leftarrow imin(N)$
    $R[imin(I_k)] \leftarrow imax(N)$
    $L[imax(I_k)] \leftarrow imin(N)$
    For i from 2 to $k-1$
      $L[imax(I_i)] \leftarrow imin(I_i)$
      $R[imin(I_i)] \leftarrow imax(I_i)$

Proposition 10 *Algorithm 8 computes the canonical generator of a closed family in $O(n)$ time.*

Proof. Time complexity is obvious because the PQ-tree has $O(n)$ nodes. Let us prove that the algorithm computes the canonical generator. First, for all i $R[i]$ and $L[i]$ are assigned because for the node N that bears the leaf i as jth node, $i = imin(I_j)$ and $i = imax(I_j)$.

When $R[imin(I_i)]$ is assigned, $(imin(I_i)..R[imin(I_i)])$ is an interval of $\mathcal{F}$: if N is a P node then $(imin(I_i)..R[imin(I_i)]) = I_i$ and if N is a Q node then $(imin(I_i)..R[imin(I_i)])$ is the union of all consecutive children of Q from i to k. The other cases are similar.

Let $R_1[t]..R_t[i]$ be the values taken by variable $R[i]$ across time. As the nodes are visited bottom-up, the intervals $(i..R_a[i])$ are an increasing sequence of intervals. We shall now prove that this sequence ends on the largest interval starting at i (resp. ending at j). Let us suppose that $(i..R_t[i]) \neq (i..R[i])$. Then $R_t[i] < R[i]$ as $(i..R_t[i])$ belongs to $\mathcal{F}$. Let us suppose $(i..R_t[i])$ is not strong. Then it is the union of children $I_a..I_b$ of some Q node N. The algorithm sets $R_t[i]$ to the last bound of the last child of N. As $(i..R[i])$ is larger, it overlaps the strong interval N, a contradiction. Now let us suppose $(i..R_t[i])$ to be strong. It corresponds to a node I of the PQ-tree whose parent is N. Either $(i..R[i])$ contains N or N is a Q node and $(i..R[i])$ is a union of children of N. In the first case, $i = imin(N)$ and thus $R_t[i] \geq imax(N)$, a contradiction. In the second case, $R_t[i] = imax(N)$, which also contradicts $R_t[i] < R[i] = imax(N)$.

We proved that the sequence of $R_a[i]$ converges towards $R[i]$ for all i. The case for $L_b[j]$ is similar. □

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 [17]. The modular decomposition tree is indeed the PQ-tree of the family of modules.

Modular decomposition is the first step in many graph algorithms like graph recognition (eg. cographs, interval graphs, permutation graphs and other classes of perfect graphs, see [5] for a survey) and transitive orientation computation [15].

Linear-time decomposition algorithms have been discovered [8, 15] 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 [11]. In this paper, we simplify the second step.

A *factorizing permutation* of a graph [6] 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 [11]. 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 11 [17] *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 7 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 12 *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. □

In order to compute the maximal right-strong modules, we use a simplified version of an algorithm due to Capelle and Habib [6]. 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 9. 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 $(v-1, v)$. □

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 9: 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 [12, 18].

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 [11] 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 5.1 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. To appear in the 13th Annual European Symposium on Algorithms (ESA). 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] A. Brandstädt, V.B. Le, and J. Spinrad. *Graph Classes: a Survey.* SIAM Monographs on Discrete Mathematics and Applications. SIAM, 1999.

[6] 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.

[7] M. Chein, M. Habib and M. C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, 37:35–50. 1981.

[8] 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.

[9] E. Dahlhaus, J. Gustedt, and R. M. McConnell. Efficient and practical algorithms for sequential modular decomposition. *J. Algorithms*, 41(2):360–387. 2001.

[10] 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.

[11] 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.

[12] 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.

[13] G.M. Landau, L. Parida and O. Weimann. Gene Proximity Analysis Across Whole Genomes via PQ Trees. 6th Combinatorial Pattern Matching Conference (CPM), 2005.

[14] 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.

[15] 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.

[16] R. M. McConnell and J. Spinrad. Ordered vertex partitioning. *Discrete Mathematics & Theoretical Computer Science*, 4:45–60. 2000.

[17] 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.

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