SIAM J. DISCRETE MATH.
Vol. 22, No. 3, pp. 1022–1039
© 2008 Society for Industrial and Applied Mathematics

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

COMPUTING COMMON INTERVALS OF K PERMUTATIONS, WITH APPLICATIONS TO MODULAR DECOMPOSITION OF GRAPHS*

ANNE BERGERON†, CEDRIC CHAUVE‡, FABIEN DE MONTGOLFIER§, AND MATHIEU RAFFINOT§

Abstract. We introduce a new approach to compute the 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.

Key words. common interval, permutation, PQ-tree, modular decomposition

AMS subject classifications. 05C05, 05C62, 68R99

DOI. 10.1137/060651331

1. Introduction.

The notion of *common interval* was introduced by Uno and Yagiura [19] in order to model the fact that, when comparing genomes, a group of genes can be rearranged but still remain connected. They 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 pioneering work.

One of the drawbacks of these algorithms is that properties of Uno and Yagiura's algorithm are not obvious [5]. 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 [19]. 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].

*Received by the editors February 1, 2006; accepted for publication (in revised form) February 17, 2008; published electronically June 11, 2008.
http://www.siam.org/journals/sidma/22-3/65133.html
†Université du Québec à Montréal, Montréal H3C 3P8, QC, Canada (bergeron.anne@uqam.ca).
‡Simon Fraser University, Burnaby V5A 1S6, BC, Canada (cedric.chauve@sfu.ca).
§LIAFA, Université Denis Diderot - Case 7014, 2 place Jussieu, F-75251 Paris Cedex 05, France (fm@liafa.jussieu.fr, raffinot@liafa.jussieu.fr). Most of the work was done when the fourth author was at Poncelet Laboratory (CNRS UMI-2615), Independent University of Moscow, 11 street Bolchoï Vlassievski, 119 002 Moscow, Russia.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

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 time into classical ones, namely PQ-trees and irreducible intervals. Conversely, 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 have been put into 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. Section 3 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 strong 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 the identity permutation will be denoted by giving the indices of its left and right bounds $(i..j)$.

DEFINITION 2.1. *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 nonempty set of permutations; they are called *trivial* intervals. In what follows, 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.

Algorithms that identify common intervals such as those in [12, 19] use a bottom-up approach that constructs the common intervals by extending small intervals until they form a common interval and then form longer common intervals by taking unions of common intervals that share elements. Here we adopt a dual approach of constructing common intervals as intersections of intervals that are not necessarily common. This had two major impacts on the rest of the theory. First, it allowed us to exploit the symmetric properties of minima and maxima of intervals, and it provided the natural notion of two families of intervals of size n. Moreover, by removing the need to maintain common intervals, the design of the algorithms was much easier, and the data structures became trivial.

DEFINITION 2.2. *Let $\mathcal{P} = \{\mathrm{Id}_n, P_2, \ldots, P_K\}$ be a set of K permutations on n*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

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* $(i..j) = (i..R[i]) \cap (L[j]..j)$.

Definition 2.2 has the following easy property that will turn out very useful in proving results on the structure of generators.

Proposition 2.3. *For any generator* (R, L), $(i..j) = (i..R[i]) \cap (L[j]..j)$ *if and only if* $L[i] \leq i \leq j \leq R[i]$.

Proof. If $(i..j) = (i..R[i]) \cap (L[j]..j)$, then $(i..j) \subseteq (i..R[i])$ and $(i..j) \subseteq (L[j]..j)$, which yields $L[i] \leq i \leq j \leq R[i]$. The converse is immediate. □

The following proposition shows how to construct a generator for the common intervals of a union of sets of permutations, given generators for the common intervals of 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 2.4. *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. 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, by Proposition 2.3, 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 2.4 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 remainder of this section focuses on particular classes of generators that turn out to have interesting properties with respect to computations.

Definition 2.5. *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*:*

$\text{IMax}[p_i]$ *is the largest interval of* P *containing* p_i *and whose elements are all* $\geq p_i$*;*

$\text{IMin}[p_i]$ *is the largest interval of* P *containing* p_i *and whose elements are all* $\leq p_i$*. And we define the following two integers:*

$\text{Sup}[p_i]$ *is the largest integer such that* $(p_i..\text{Sup}[p_i]) \subseteq \text{IMax}[p_i]$*;*

$\text{Inf}[p_i]$ *is the smallest integer such that* $(\text{Inf}[p_i]..p_i) \subseteq \text{IMin}[p_i]$*.*

Remark that $(p_i..\text{Sup}[p_i])$ and $(\text{Inf}[p_i]..p_i)$ 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 $\text{IMax}[5] = (7, 5, 9, 6)$ and $\text{Sup}[5] = 7$, and $\text{IMin}[8] = (6, 2, 3, 8)$ and $\text{Inf}[8] = 8$.

Proposition 2.6. *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 $\text{Sup}[i] \geq j$ and $\text{Inf}[j] \leq i$ since all elements in the set $(i..j)$ are consecutive in permutation P. Thus $(i..j) = (i..\text{Sup}[i]) \cap (\text{Inf}[j]..j)$. On the other hand, suppose that $\text{Sup}[i] \geq j$ and $\text{Inf}[j] \leq i$; then $\text{IMax}[i]$ contains j and $\text{IMin}[j]$ contains i. Since both $\text{IMax}[i]$ and $\text{IMin}[j]$ are intervals of P, their intersection is an interval and is equal to $(i..j)$. □

Example. Let $\mathcal{P} = \{\text{Id}_8, P_2\}$ and $\mathcal{Q} = \{\text{Id}_8, P_3\}$ with

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

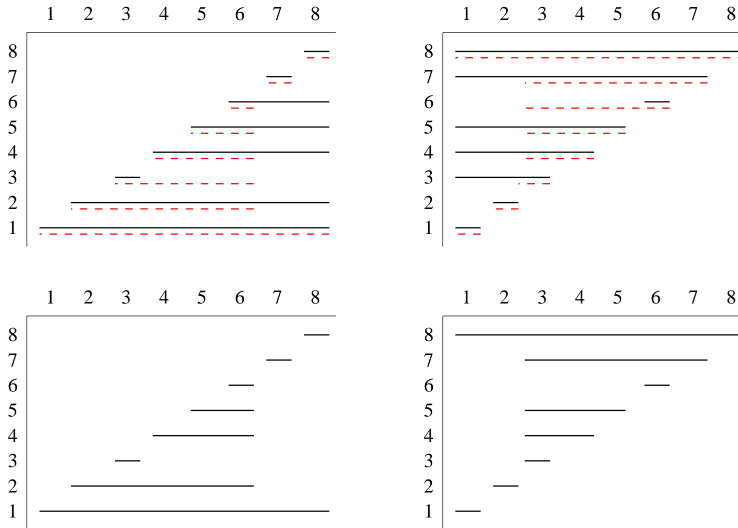

FIG. 1. *The top two diagrams show the generators* (Sup, Inf) *of the common intervals of the set $\mathcal{P}$ in solid lines and of set $\mathcal{Q}$ in dashed lines. A line in row i of the left diagram extends from column i to column* $\mathrm{Sup}(i)$*, and a line in row i of the right diagram extends from column* $\mathrm{Inf}(i)$ *to column i. The bottom diagrams show a generator for the common intervals of $\mathcal{P} \cup \mathcal{Q}$ constructed using Proposition* 2.4.

$$\mathrm{Id}_8 = (1,2,3,4,5,6,7,8), \quad P_2 = (1,3,2,4,5,7,6,8), \quad P_3 = (2,8,3,4,5,6,1,7).$$

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

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 2.7. *Let P be a permutation on n elements. If the bounds of intervals* $\mathrm{IMax}[k]$ *and* $\mathrm{IMin}[k]$ *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 kth iteration of the second *For* loop, $\mathrm{Inf}[k'] = m[k']$ for all $k' < k$, and $m[k] \in \mathrm{IMin}[k]$. This is the case at the beginning of iteration $k = 2$, since $\mathrm{Inf}[1] = 1$. By definition, $\mathrm{Inf}[k] \leq k$, thus before entering the *While* loop, we have $\mathrm{Inf}[k] \leq m[k]$. If the test $m[k] - 1 \in \mathrm{IMin}[k]$ of the *While* loop is true, then $\mathrm{Inf}[k] \leq m[k] - 1$, implying that $\mathrm{Inf}[k] \leq \mathrm{Inf}[m[k]-1]$. Since $\mathrm{Inf}[m[k]-1] = m[m[k]-1]$ by hypothesis, the instruction in the *While* loop preserves the invariant $\mathrm{Inf}[k] \leq m[k]$. When the test of the *While* loop becomes false, then $\mathrm{Inf}[k]$ is greater than $m[k]-1$, thus $\mathrm{Inf}[k] = m[k]$. The proof of correctness for Sup is similar.

Using the inverse of permutation P, the tests in the *While* loops can be done in

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

constant time. 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 I 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, that is, $I = \{(1..1), (2..2), \ldots, (n..n)\}$, and at the completion of the algorithm, there is only one, namely $(m[n]..n) = (1..n)$, since $\text{Inf}[n] = 1$. The instruction in the *While* loop merges two consecutive intervals $(m[m[k]-1]..m[k]-1)$ and $(m[k]..k)$ of I into one, since the current value of $m[k]$ becomes $m[m[k]-1]$. Since there were n intervals to start with, there can be at most $n-1$ of these merges. ∎

The computation of the bounds of intervals $\text{IMax}[k]$ and $\text{IMin}[k]$, 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 $\text{IMax}[p_i]$.

PROPOSITION 2.8. *Let $P = (p_1, \ldots, p_n)$ be a permutation on n elements. Algorithm 2 computes the left bound of all intervals $\text{IMax}[p_i]$ in $O(n)$ time.*

Proof. The time complexity of Algorithm 2 is immediate since each position is stacked once. Its correctness relies on the fact that, at the beginning of the ith 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 OF $\text{IMax}[p_i]$ 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
    left bound of $\text{IMax}[p_i] \leftarrow s+1$
    Push i on S

To summarize the results of this section, we have the following theorem.

THEOREM 2.9. *Let $\mathcal{P} = \{\text{Id}_n, P_2, \ldots, P_K\}$ 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, based on the notion of *commuting* sets. Note that commuting families are also known as *laminar* families [18] in the field of combinatorial optimization.

DEFINITION 3.1. *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 $\mathcal{P} = \{\text{Id}_n, P_2, \ldots, P_K\}$ 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 $R[i] \leq R[j]$. (Similar values can be defined for L, but we will not refer to them explicitly.)*

It turns out that generators defined in section 2 are commuting. Indeed, generators defined in Proposition 2.4 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

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

$a < b$ and $a' < b'$, then $\min(a, a') < \min(b, b')$ and $\max(b, b') > \max(a, a')$. For the generator (Sup, Inf), we have the following proposition.

PROPOSITION 3.2. *The generator* (Sup, Inf) *for the common intervals of permutations P and Id_n is commuting.*

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

PROPOSITION 3.3. *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 3.4. *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 jth 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 $i' = \mathit{Support}[i]$, 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 $\mathit{Support}[j] \geq i$, 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, $\mathit{Support}[i'] = i$ since $\mathit{Support}[i']$ must be greater than or equal to i. If it is greater, then $(\mathit{Support}[i']..j)$ 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 set of common intervals of a set of permutations is an example of more general families of intervals, the *closed* families. Closed families can have a quadratic number of elements, but a classical result establishes a bijection between PQ-trees with n leaves and closed families of Id_n, thus allowing a representation of size $O(n)$ ([3]—see also Proposition 5.3). In this section, we develop a new canonical representation for such families, based

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

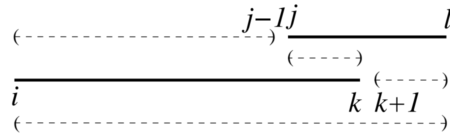

FIG. 2. *Two overlapping common intervals determine four more common intervals.*

on the generators of the previous section. We will discuss the relations between this representation and PQ-trees in section 5.

DEFINITION 4.1. *A* closed *family $\mathcal{F}$ of intervals of a permutation σ on n elements is a family that contains all singletons and the interval $(1..n)$ and that has the following property: if $(i..k)$ and $(j..l)$ are in $\mathcal{F}$, and $i < j \leq k < l$, then $(i..j-1)$, $(j..k)$, $(k+1..l)$, and $(i..l)$ belong to $\mathcal{F}$.*

In what follows, we will suppose, for simplicity, that the permutation σ is the identity permutation, thus every interval of the family $\mathcal{F}$ is an interval of the form $(i..j)$. The closure properties of Definition 4.1 are well-known properties of the set of common intervals of a set of permutations and are illustrated in Figure 2. Conversely, it is easy to extend the definition of generators (Definition 2.2) to the more general case of closed families, and we will refer to a *generator of a closed family* $\mathcal{F}$ as a pair (R, L) that satisfies the conditions of Definition 2.2 with respect to the members of $\mathcal{F}$. Among all possible generators, the following one provides a representation of size $O(n)$ for any closed family.

DEFINITION 4.2. *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 4.3. *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 $R[i]$ as the largest integer such that $(i..R[i]) \in \mathcal{F}$, and define $L[i]$ as 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 $(i..R[i])$. ∎

The following elementary property of canonical generators states that if two intervals of a generator overlap, they always do it correctly, in the sense that an $(L[j]..j)$ interval is always at the left of an $(i..R[i])$ interval.

PROPOSITION 4.4. *For a canonical generator (R, L), if $(i..R[i])$ and $(L[j]..j)$ overlap, then $L[j] \leq i \leq j \leq R[i]$.*

Proof. Suppose that $i \leq L[j] \leq R[i] \leq j$; then $(i..j)$ is in $\mathcal{F}$ since both $(i..R[i])$ and $(L[j]..j)$ are. Then, by definition, $(i..j) = (i..R[i]) \cap (L[j]..j)$, thus $(i..R[i]) \cup (L[j]..j) = (i..R[i]) \cap (L[j]..j)$, implying $(i..R[i]) = (L[j]..j)$, contrary to the assumption that these intervals overlap. ∎

THEOREM 4.5. *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

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

integer $k' > k$ such that

$$Support[k'] = k \quad \text{and} \quad R[k'] = R[k], \tag{4.1}$$

where $Support[k']$ is defined (Definition 3.1) as the greatest integer smaller than k' such that $R'[Support[k']] \geq R'[k']$. Statement (4.1) 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 (4.1), 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] ← 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 (continued). Let $\mathcal{R} = \{\text{Id}_8, P_2, P_3\}$ with

$$\text{Id}_8 = (1, 2, 3, 4, 5, 6, 7, 8), \quad P_2 = (1, 3, 2, 4, 5, 7, 6, 8), \quad P_3 = (2, 8, 3, 4, 5, 6, 1, 7).$$

Two generators for the common interval of $\mathcal{R}$ are shown in Figure 3. The second one is canonical.

5. Tranformations of canonical representations.

There exist many representations of closed families, and this variety is useful. Indeed, some are better suited for algorithmic purposes [12], and others yield nice graphical representations of the organization and nesting properties of a set of common intervals [3, 14]. In this section, we present algorithms that allow the conversion between the canonical generators of the preceding section and the classic representation using PQ-trees.

5.1. Canonical representations of closed families.

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

Definition 5.1. *Let $\mathcal{F}$ be a closed family. A* strong interval *of $\mathcal{F}$ is an interval of $\mathcal{F}$ that commutes with each interval of $\mathcal{F}$.*

A *proper commuting family* $\mathcal{F}$ of subsets of a set V is a commuting family such that $V \in \mathcal{F}$, all the singletons are in $\mathcal{F}$, and $\emptyset \notin \mathcal{F}$. In particular, the strong common intervals of K permutations 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. 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

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

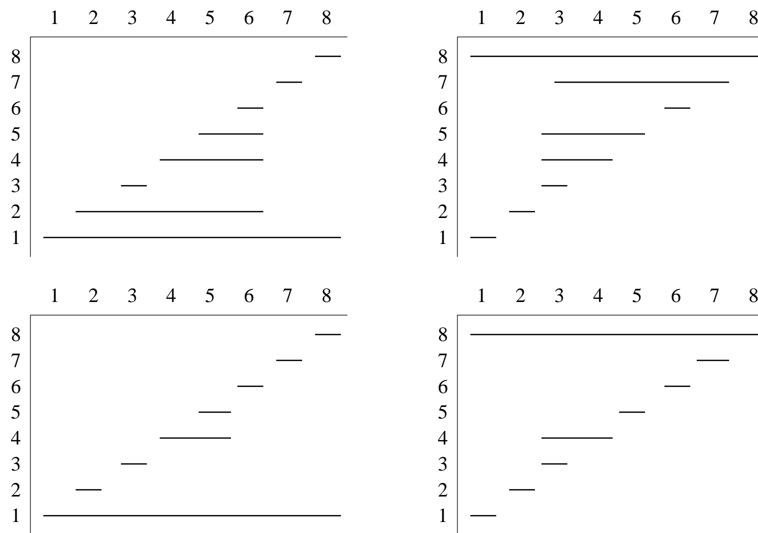

FIG. 3. *The top two diagrams show a generator for the common intervals of $\mathcal{R}$ constructed using Proposition* 2.4 *(see Figure* 1*). The bottom diagrams shows the canonical generator constructed by Algorithm* 5.

proper commuting family. The inclusion trees on V are thus in bijection with the proper commuting families on V.

For closed families, PQ-trees play the same role as the inclusion trees for proper commuting families.

DEFINITION 5.2. *A PQ-*tree *on finite set V is a tree whose leaves are labeled from* 1 *to $|V|$ and whose internal nodes are labeled P-nodes or Q-nodes. A P-node must have at least two children, and a Q-node must have at least three children. The children of a P-node are unordered, and the children of a Q-node are totally ordered.*

The *reversal* of a Q node consists in reversing the total order of its children. An *extended frontier* of a PQ-tree is either the frontier of a P node, the union of frontiers of consecutive children of a Q node, or a singleton.

PROPOSITION 5.3 (see [3, 7]). *Given a closed family $\mathcal{F}$, 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.*

Example. Let $\mathcal{P}_2 = \{\mathrm{Id}_9, P_4, P_5, P_6\}$ with

$$\mathrm{Id}_9 = (1,2,3,4,5,6,7,8,9), \qquad P_5 = (6,5,7,8,9,1,2,3,4),$$
$$P_4 = (9,8,7,5,6,4,3,2,1), \qquad P_6 = (1,3,2,4,5,6,9,8,7).$$

Figure 4 shows the canonical generator of the set $\mathcal{P}_2$ and the corresponding PQ-tree.

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 in [12]. The links between PQ-trees and irreducible intervals have been studied in [13] that present linear-time algorithms for the conversion.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

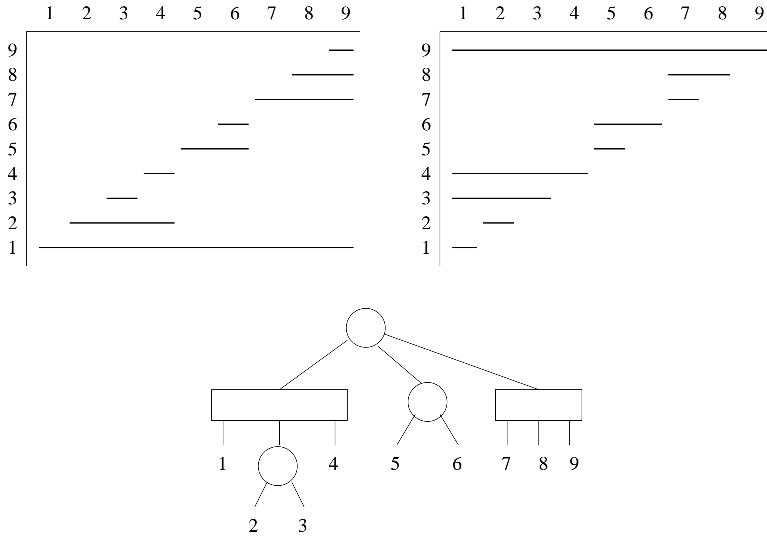

FIG. 4. *The canonical generator and the associated PQ-tree for the set of permutations* $\mathcal{P}_2 = \{\text{Id}_9, P_4, P_5, P_6\}$.

5.2. From canonical generators to strong intervals. Strong intervals of a closed family $\mathcal{F}$ are associated with *overlap classes*: let us consider the canonical generator (R, L) of $\mathcal{F}$. The transitive closure of the overlap relation on all intervals of $R \cup L$ is an equivalence relation, and its equivalence classes are henceforth called *overlap classes*. We will prove, in Theorem 5.8, that once the overlap classes are identified, the task of identifying the strong intervals is almost complete.

A *trivial* overlap class contains only a single interval $(i..R[i])$ or $(L[j]..j)$. We have the following lemma.

LEMMA 5.4. *A trivial overlap class is a strong interval of* $\mathcal{F}$.

Proof. Consider a trivial overlap class $\{(i..j)\}$. Either $j = R[i]$ or $L[j] = i$. Suppose that $j = R[i]$ and that $(i..j)$ is not strong. Then it is overlapped by an interval $(i'..j')$ belonging to $\mathcal{F}$ but not to the generator. If $i \leq i' \leq j \leq j'$, then the interval $(i..j')$ is in $\mathcal{F}$ but larger than $(i..R[i])$ in violation with the fact that $j' \leq R[i]$ (see Proposition 2.3). Let us then suppose $i' \leq i \leq j' \leq j$. Since $(L[j']..j')$ does not overlap $(i..R[i])$, we have $i \leq L[j'] \leq j'$. Since $i' \leq i \leq L[j']$, $(i'..j')$ is larger than $(L[j']..j')$ and thus does not belong to $\mathcal{F}$. Interval $(i..j)$ is therefore strong. The other case is similar. ∎

For nontrivial overlap classes, we need to explain the relations between the various intervals that belong to the classes. Note that any nontrivial overlap class must contain intervals of R and L since intervals of R (resp., L) do not overlap each other. Figure 5 shows a typical nontrivial class: the number of left and right intervals is the same, left and right bounds are nicely aligned, and the left and right intervals are nested. The following lemmas establish these properties formally and can be skipped at the first reading.

LEMMA 5.5. *Let* (R, L) *be the canonical generator of a closed family* $\mathcal{F}$. *We have the following:* (1) *if* $(i..R[i])$ *overlaps* $(L[j]..j)$ *and* $(L[j']..j')$, *then* $L[j] = L[j']$; *and* (2) *if* $(L[j]..j)$ *overlaps* $(i..R[i])$ *and* $(i'..R[i'])$, *then* $R[i] = R[i']$.

Proof. Note that if $(i..R[i])$ overlaps $(L[j]..j)$, then $(i..j)$ belongs to $\mathcal{F}$ and, by Proposition 4.4, $i \geq L[j]$. The hypothesis that $(i..R[i])$ overlaps $(L[j]..j)$ implies fur-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

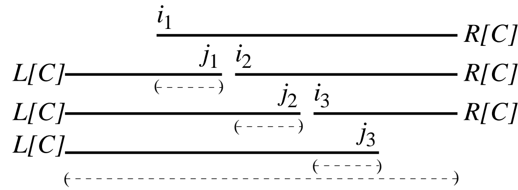

FIG. 5. *Structure of a nontrivial overlap class $\mathcal{C}$ with $k = 3$. Dashed lines represent the strong intervals corresponding to the class.*

ther that $i < L[j]$; otherwise we would have $(L[j]..j) \subseteq (i..R[i])$. A similar argument shows that $(i..R[i])$ overlapping $(L[j']..j')$ implies that $j' < R[i]$.

Now suppose $L[j] \neq L[j']$ and $j' > j$; then $L[j'] < L[j]$, since L is commuting. This implies that $L[j'] < L[j] < i \leq j < j' < R[i]$. Interval $(L[j']..i-1)$ belongs to $\mathcal{F}$ since it is $(L[j']..j') \setminus (i..R[i])$. Thus $(L[j']..j)$ is also in $\mathcal{F}$ since it is the union of overlapping $(L[j']..i-1)$ and $(L[j]..j)$. Applying Proposition 4.4 to interval $(L[j']..j)$ implies that $L[j'] \geq L[j]$, which yields a contradiction. Therefore $L[j] = L[j']$.

Statement (2) is proved in a similar way. ∎

Lemma 5.5 implies that, in a nontrivial class $\mathcal{C}$, all intervals of the form $(i..R[i])$ share the same right bound, and all intervals of the form $(L[j]..j)$ share the same left bound. We will henceforth denote $L[\mathcal{C}]$ and $R[\mathcal{C}]$ these common bounds. Moreover, the interval $(L[\mathcal{C}]..R[\mathcal{C}])$ is necessarily strong.

LEMMA 5.6. *Let $\mathcal{C}$ be a nontrivial overlap class of a closed family $\mathcal{F}$. Then $(L[\mathcal{C}]..R[\mathcal{C}])$ is a strong interval of $\mathcal{F}$.*

Proof. Since $\mathcal{C}$ is a nontrivial class, it contains at least two overlapping intervals $(L[\mathcal{C}]..j)$ and $(i..R[\mathcal{C}])$. The interval $(L[\mathcal{C}]..R[\mathcal{C}])$ is an interval of $\mathcal{F}$ since it is the union of those two overlapping intervals. Suppose that $(i..j) \in \mathcal{F}$ overlaps $(L[\mathcal{C}]..R[\mathcal{C}])$ and $i < L[\mathcal{C}] \leq j < R[\mathcal{C}]$; then $(L[j]..j)$ contains $(i..j)$ and overlaps $(L[\mathcal{C}]..R[\mathcal{C}])$, thus it overlaps at least one member of $\mathcal{C}$, and therefore it belongs to $\mathcal{C}$ and $L[j] = L[\mathcal{C}]$. On the other hand, if $L[\mathcal{C}] < i \leq R[\mathcal{C}] < j$, then $(i..R[i])$ overlaps $(L[\mathcal{C}]..R[\mathcal{C}])$, thus it overlaps at least one member of $\mathcal{C}$. Thus no interval overlaps $(L[\mathcal{C}]..R[\mathcal{C}])$, and therefore it is a strong interval. ∎

LEMMA 5.7. *Let (R, L) be the canonical generator of a closed family $\mathcal{F}$, and let $\mathcal{C}$ be a nontrivial overlap class containing $(i_1..R[\mathcal{C}]), \ldots, (i_k..R[\mathcal{C}])$ and $(L[\mathcal{C}]..j_1), \ldots, (L[\mathcal{C}]..j_l)$, with $i_1 < \cdots < i_k$ and $j_1 < \cdots < j_l$. Then $k = l$, and for all $a \in (1..k)$, $(i_a..j_a)$ is a strong interval of $\mathcal{F}$.*

Proof. We first show the result for $k = l = 1$. In this case, there are only two intervals $(i_1..R[\mathcal{C}])$ and $(L[\mathcal{C}]..j_1)$, and they overlap to form interval $(i_1..j_1)$. To show that this interval is strong, suppose that it contains more than one element and that $(i..j)$ overlaps it with $i < i_1 \leq j < j_1$. Then $(L[j]..j)$ also overlaps $(i_1..R[\mathcal{C}])$, and the class must have more than two intervals. If $(i..j)$ overlaps $(i_1..j_1)$ with $i_1 < i \leq j_1 < j$, then we obtain a similar contradicion using interval $(i..R[i])$.

Now suppose that, up to index $a \geq 1$, $(i_a..j_a)$ is a strong interval of $\mathcal{F}$, $k > a$ if and only if $l > a$, and all intervals are consecutive; that is, $i_{b+1} = i_b + 1$ when $b < a$. We will show that these statements are true for $a + 1$.

First, note that the interval $(j_a + 1..R[\mathcal{C}])$ always belongs to $\mathcal{F}$, being the difference $(i_a..R[\mathcal{C}]) \setminus (L[\mathcal{C}]..j_a)$, and since the generator is commuting, this implies that $R[j_a + 1] = R[\mathcal{C}]$. If $k > a$, then $l > a$ since $(i_{a+1}..R[\mathcal{C}])$ does not overlap any interval already processed. If $l > a$, then $(L[\mathcal{C}]..j_{a+1})$ overlaps $(j_a + 1..R[\mathcal{C}])$ and both are in $\mathcal{C}$, implying that $k > a$ and $i_{a+1} = j_a + 1$, since the hypothesis that $(i_a..j_a)$ is strong

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

implies that $i_{a+1} > j_a$.

To finish the proof, we must show that $(i_{a+1}..j_{a+1})$ is strong. Suppose that $(i..j) \in \mathcal{F}$ overlaps $i_{a+1}..j_{a+1})$ and $i < i_{a+1} \leq j < j_{a+1}$; then $(L[j]..j)$ overlaps $(i_{a+1}..R[\mathcal{C}])$, thus it overlaps at least one member of $\mathcal{C}$, and therefore it belongs to $\mathcal{C}$. But $i_{a+1} \leq j < j_{a+1}$ and $i_{a+1} = j_a + 1$ implies that $j_a < j < j_{a+1}$, which contradicts the fact that the list of j indices is totally ordered.

On the other hand, if $i_{a+1} < i \leq j_{a+1} < j$, then $(i..R[i])$ belongs to $\mathcal{C}$, implying that $(L[\mathcal{C}]..i-1)$ is in $\mathcal{F}$, thus $(L[i-1]..i-1)$ is in $\mathcal{C}$. Since we have that $i_{a+1} - 1 < i - 1 < j_{a+1}$, thus $j_a < i - 1 < j_{a+1}$, $i - 1$ is again out of order in the list of j indices. ∎

Let (R, L) be the canonical generator of a closed family $\mathcal{F}$, and define the set of *traces* of overlap classes as the set $\mathcal{T}$ of intervals containing the following:

- For each nontrivial overlap class $\mathcal{C} = \{(i_1..R[\mathcal{C}]), \ldots, (i_k..R[\mathcal{C}]), (L[\mathcal{C}]..j_1), \ldots, (L[\mathcal{C}]..j_k)\}$
  - $(L[\mathcal{C}]..R[\mathcal{C}])$,
  - $(i_a..j_a)$ for each $1 \leq a \leq k$.
- $(i..j)$ if $\mathcal{C} = \{(i..j)\}$ is trivial.

THEOREM 5.8. *The set $\mathcal{T}$ of traces of overlap classes of the canonical generator of a closed family $\mathcal{F}$ is equal to the family of strong intervals of $\mathcal{F}$.*

Proof. The fact that all intervals of $\mathcal{T}$ are strong is proved in Lemmas 5.4, 5.6, and 5.7.

Conversely, let $S = (i..j)$ be a strong interval. If either $R[i] = j$ or $L[j] = i$, then S is in the generator and forms a trivial overlap class. Otherwise $(i..R[i])$ and $(L[j]..j)$ overlap and thus belong to an overlap class $\mathcal{C}$. Using the same notation as above we have $i = i_a$ and $j = j_b$ for some a and b. We showed in the proof of Lemma 5.7 that we must have $a \leq b$. If $a < b$, S is overlapped by $(i_b..R[\mathcal{C}])$ and thus is not strong. Therefore $a = b$. ∎

Theorem 5.8 implies that computing the strong intervals of $\mathcal{F}$ amounts to computing $\mathcal{T}$. This can be done by a classical parenthesis-matching algorithm using a stack since we proved that the left and and right bounds of the strong intervals defined by an overlap class form a balanced expression. Namely, for nontrivial classes, the corresponding expression is

$$(\ \ldots \ (\ldots)(\ldots)\ldots(\ldots) \ \ldots \ )$$

and for trivial classes is $(\ldots)$.

More formally, 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 3 this list is

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

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. The following algorithm takes this sorted list as input and outputs the strong intervals sorted by increasing left bounds.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

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 5.9. *Given the ordered list* $(a_1, \dots, a_{4n})$ *of the* $4n$ *bounds of a canonical generator* (R, L)*, Algorithm* 6 *computes* $\mathcal{T}$ *in* $O(n)$ *time.*

Proof. Algorithm 6 will correctly match left and right bounds of a balanced expression. Since the left and right bounds of the strong intervals defined by each overlap class are balanced, and two overlap classes are either nested or disjoint, the list $(a_1, \dots, a_{4n})$ is balanced, and a correct matching of corresponding bounds will generate all the strong intervals. 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. A suitable sorting algorithm, bucket-sort for example, will allow the identification of duplicates in $O(n)$ time. □

5.3. From strong intervals to PQ-trees. By Proposition 5.3, the PQ-tree of a family $\mathcal{F}$ can be generated from its canonical generator by first computing the set of strong intervals of $\mathcal{F}$, ordering them into a tree structure, and finally labeling them as P or Q nodes.

Consider the example of Figure 4; Algorithm 6 produces the stack

$$(1,1,1,1,1,\overline{1},2,2,\overline{2},3,\overline{3},\overline{3},4,\overline{4},\overline{4},\overline{4},5,5,5,5,\overline{5},6,\overline{6},\overline{6},\overline{6},7,7,7,\overline{7},8,\overline{8},9,\overline{9},\overline{9},\overline{9},\overline{9},\overline{9})$$

and outputs the set of strong intervals

$$(1..1),(2..2),(3..3),(2..3),(4..4),(1..4),(1..4),(5..5),(6..6),$$
$$(5..6),(5..6),(7..7),(8..8),(9..9),(7..9),(7..9),(1..9),(1..9).$$

Each of these intervals corresponds to a node of the PQ-tree.

Given a proper commuting family of $n \le m < 2n$ intervals, the following well-known algorithm computes its inclusion tree. To upgrade an inclusion tree to a PQ-tree, it is necessary to label the nodes as P or Q and to order the children of Q-nodes. Theorem 5.10 explains how to do these last tasks.

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

```
Bucket-sort in decreasing order the intervals of F according to their right bound
Bucket-sort in increasing order the intervals of F according to their left bound
Let I_1..I_m be the list of sorted intervals
F ← I_1 (* I_1 = V is the root *)
k ← 2
While k ≤ m
    If I_k ⊂ F
        Parent(I_k) ← F
        F ← I_k
        k ← k + 1
    else
        F ← Parent(F)
```

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Let $I_1 = (l_1..r_1), \ldots, I_k = (l_k..r_k)$ be the children of a Q node. For every i, $I_i \cup I_{i+1}$ is an interval of the family $\mathcal{F}$. Intervals 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, i.e., $r_i + 1 = l_{i+1}$.

THEOREM 5.10. *Let (R, L) be the canonical generator of a closed family $\mathcal{F}$. The canonical PQ-tree of $\mathcal{F}$ can be computed from (R, L) in $O(n)$ time.*

Proof. Algorithm 7 computes the inclusion tree in $O(n)$ time. The bucket sort is a stable linear-time sorting algorithm, thus intervals with the same left bound are sorted by their right bounds. The overall time complexity is obviously $O(n)$.

Given the inclusion tree of the strong intervals of a closed family $\mathcal{F}$, a PQ-tree of $\mathcal{F}$ can be built by labeling as 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. Thus, it is necessary only to label the nodes.

Each internal node of the inclusion tree has at least two children, since a node and its only child would have the same frontier. Every node with two children is labeled P. To test whether a node with at least three children is a P or Q node, it suffices to probe its two first children: if their union is an interval of the family $\mathcal{F}$, the node is labeled Q; otherwise it is labeled P. This can be done in $O(1)$ time per node, using the generator. ∎

5.4. From *PQ*-trees to canonical generators. Given a PQ-tree T, $\sigma(T)$ is the permutation of the leaves of T obtained by a left-to-right traversal of the tree. We may always assume that $\sigma(T) = \mathrm{Id}_n$ 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 $I_1, \ldots, I_k$. Let l_i and r_i be the indices of the left and right bounds of I_i. Let $\mathrm{imin}(N)$ and $\mathrm{imax}(N)$ be the indices, 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 of T.

ALGORITHM 8. COMPUTING THE CANONICAL GENERATOR FROM A PQ-TREE.

```
For each internal node N of T taken bottom-up
    If N is a Q node
        For i from 1 to k
            L[imax(I_i)] ← imin(N)
            R[imin(I_i)] ← imax(N)
    Else
        R[imin(I_1)] ← imax(N)
        L[imax(I_1)] ← imin(N)
        R[imin(I_k)] ← imax(N)
        L[imax(I_k)] ← imin(N)
        For i from 2 to k − 1
            L[imax(I_i)] ← imin(I_i)
            R[imin(I_i)] ← imax(I_i)
```

PROPOSITION 5.11. *Algorithm 8 computes the canonical generator of a closed family in $O(n)$ time.*

Proof. The time complexity of the algorithm is obvious because the PQ-tree T has $O(n)$ nodes. Let us prove that the algorithm computes the canonical generator.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

First, for all i, the value of $R[i]$ and $L[i]$ will be determined because, for the node N that bears the leaf i as jth node, $i = \text{imin}(I_j)$ and $i = \text{imax}(I_j)$.

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

Let $R_t[1], \ldots, R_t[i]$ be the values taken by the variable $R[i]$ during the execution of the algorithm. Since the nodes are visited bottom-up, the intervals $(i..R_t[i])$ are a nested sequence of intervals. We shall now prove that this sequence ends with the largest interval starting at i (resp., ending at j). Let us suppose that $(i..R_t[i]) \neq (i..R[i])$. Then $R_t[i] < R[i]$ since $(i..R_t[i])$ belongs to $\mathcal{F}$. Let us suppose $(i..R_t[i])$ is not strong. Then it is the union of children $I_a, \ldots, I_b$ of some Q-node N. The algorithm sets $R_t[i]$ to the left bound of the last child of N. Since $(i..R[i])$ is larger, it overlaps the strong interval N, which is impossible. Now let us suppose that $(i..R_t[i])$ is 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, $i = \text{imin}(N)$ and thus $R_t[i] \geq \text{imax}(N)$, a contradiction. In the second case, $R_t[i] = \text{imax}(N)$, which also contradicts $R_t[i] < R[i] = \text{imax}(N)$.

We proved that the sequence of $R_t[i]$ converges toward $R[i]$ for all i. The case for $L_t[j]$ 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 such as graph recognition (e.g., cographs, interval graphs, permutation graphs, and other classes of perfect graphs; see [4] 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 $\{1, \ldots, n\}$ 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 $(u..v)$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

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 we have the following proposition.

PROPOSITION 6.1 (see [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 6.2. *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 the following proposition.

PROPOSITION 6.3. *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's 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 $(v - 1..v)$ is also 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 the following theorem.

THEOREM 6.4. *Given a graph G and a factorizing permutation of G, it is possible to compute the modular decomposition tree of G in time $O(n + m)$ and in a simple way.*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

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 the 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, 19].

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 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 5.2 can then be used to compute the strong interval-modules, that are also 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, Lect. Notes in Bioinform. 3388, Springer-Verlag, Berlin, 2004, pp. 1–15.

[2] A. Bergeron, C. Chauve, F. de Montgolfier, and M. Raffinot, *Computing common intervals of K permutations, with applications to modular decomposition of graphs*, in Proceedings of the 13th Annual European Symposium on Algorithms (ESA), Lecture Notes in Comput. Sci. 3669, Springer-Verlag, Berlin, 2005, pp. 779–790.

[3] S. Booth and G. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-trees algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[4] A. Brandstädt, V. B. Le, and J. P. Spinrad, *Graph Classes: A Survey*, SIAM Monogr. Discrete Math. Appl. 3, SIAM, Philadelphia, 1999.

[5] B. M. Bui Xuan, M. Habib, and C. Paul, *Revisiting T. Uno and M. Yagiura's algorithm*, in Proceedings of the 16th International Symposium on Algorithms and Computation (ISAAC), Lecture Notes in Comput. Sci. 3827, Springer-Verlag, Berlin, 2005, pp. 146–155.

[6] C. Capelle and M. Habib, *Graph decompositions and factorizing permutations*, in Proceedings of the Fifth Israel Symposium on Theory of Computing and Systems (ISTCS), IEEE Computer Society, New York, 1997, pp. 132–143.

[7] M. Chein, M. Habib, and M. C. Maurer, *Partitive hypergraphs*, Discrete Math., 37 (1981), pp. 35–50.

[8] A. Cournier and M. Habib, *A new linear algorithm for modular decomposition*, in Proceedings of the 19th International Colloquium of Trees in Algebra and Programming (CAAP), Lecture Notes in Comput. Sci. 787, Springer-Verlag, Berlin, 1994, pp. 68–84.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

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

[10] M. Figeac and J.-S. Varré, *Sorting by reversals with common intervals*, in Proceedings of the 4th International Workshop of Algorithms in Bioinformatics (WABI), Lecture Notes in Comput. Sci. 3240, Springer-Verlag, Berlin, 2004, pp. 26–37.

[11] M. Habib, F. de Montgolfier, and C. Paul, *A simple linear-time modular decomposition algorithm for graphs, using order extension*, in Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT), Lecture Notes in Comput. Sci. 3111, Springer-Verlag, Berlin, 2004, pp. 187–198.

[12] S. Heber and J. Stoye, *Finding all common intervals of k permutations*, in Proceedings of the 12th Annual Symposium of Combinatorial Pattern Matching (CPM), Lecture Notes in Comput. Sci. 2089, Springer-Verlag, Berlin, 2001, pp. 207–218.

[13] G. M. Landau, L. Parida, and O. Weimann, *Gene proximity analysis across whole genomes via PQ trees*, J. Comput. Biol., 12 (2005), pp. 1289–1306.

[14] R. M. McConnell and F. de Montgolfier, *Algebraic operations on PQ-trees and modular decomposition trees*, in Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG), Lecture Notes in Comput. Sci. 3787, Springer-Verlag, Berlin, 2005, pp. 421–432.

[15] R. M. McConnell and J. Spinrad, *Linear-time modular decomposition and efficient transitive orientation of comparability graphs*, in Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1994, pp. 536–545.

[16] R. M. McConnell and J. Spinrad, *Ordered vertex partitioning*, Discrete Math. Theor. Comput. Sci., 4 (2000), pp. 45–60.

[17] R. H. Möhring and F. J. Radermacher, *Substitution decomposition for discrete structures and connections with combinatorial optimization*, Ann. Discrete Math., 19 (1984), pp. 257–356.

[18] A. Schrijver, *Combinatorial Optimization*, Springer-Verlag, Berlin, 2003.

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

Downloaded 05/13/16 to 128.179.252.139. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.