Algorithmica
DOI 10.1007/s00453-009-9332-1

Common Intervals of Multiple Permutations

Steffen Heber · Richard Mayr · Jens Stoye

Received: 24 January 2009 / Accepted: 11 June 2009

Abstract Given k permutations of n elements, a k-tuple of intervals of these permutations consisting of the same set of elements is called a *common interval*. We present an algorithm that finds in a family of k permutations of n elements all z common intervals in optimal $O(kn+z)$ time and $O(n)$ additional space. Additionally, we show how to adapt this algorithm to multichromosomal and circular permutations.

This extends a result by Uno and Yagiura (Algorithmica 26:290–309, 2000) who present an algorithm to find all z common intervals of $k=2$ (regular) permutations in optimal $O(n+z)$ time and $O(n)$ space. To achieve our result, we introduce the set of *irreducible intervals*, a generating subset of the set of all common intervals of k permutations.

Keywords Common intervals of permutations · Multichromosomal permutations · Circular permutations

1 Introduction

Let $\Pi = (\pi_1, \ldots, \pi_k)$ be a family of k permutations of $N := \{1, 2, \ldots, n\}$. For $x \in N$, we denote the x-th element of π_i by $\pi_i(x)$. Without loss of generality we suppose

S. Heber (✉)
Department of Computer Science, North Carolina State University, 1519 Partners II (Centennial Campus), Raleigh, NC 27695, USA
e-mail: sheber@ncsu.edu

R. Mayr
School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, UK

J. Stoye
Technische Fakultät, Universität Bielefeld, 33594 Bielefeld, Germany

that $\pi_1 = id_n$, the identity permutation of n elements. For $x, y \in N$, $x \leq y$, we define $[x, y] := \{x, x+1, \ldots, y\}$ and call $\pi([x, y]) := \{\pi(i) \mid i \in [x, y]\}$ an *interval* of π. A subset $c \subseteq N$ of cardinality $|c| \geq 2$ is called a *common interval* of Π if and only if there exist $1 \leq l_j < u_j \leq n$ for all $1 \leq j \leq k$ such that

$$c = id_n([l_1, u_1]) = \pi_2([l_2, u_2]) = \cdots = \pi_k([l_k, u_k]).$$

This definition excludes common intervals consisting of a single element, while the set N will always be a common interval. The set of all common intervals of Π is denoted by C_Π.

A common interval c is represented either by specifying its elements or by the shorter notation $\pi_j([l_j, u_j])$ for $j \in \{1, \ldots, k\}$. For $\pi_1 = id_n$ this notation further simplifies to $[l_1, u_1]$. We call $\pi_j(l_j)$ the *left end* of c in π_j, and $\pi_j(u_j)$ the *right end* of c in π_j.

Example 1 Let $n = 9$ and $\Pi = (\pi_1, \pi_2, \pi_3)$ with $\pi_1 = id_9$, $\pi_2 = (9, 8, 4, 5, 6, 7, 1, 2, 3)$, and $\pi_3 = (1, 2, 3, 8, 7, 4, 5, 6, 9)$. Set $c := [4, 7]$. Since $4 = \pi_1(4) = \pi_2(3) = \pi_3(6)$, $5 = \pi_1(5) = \pi_2(4) = \pi_3(7)$, $6 = \pi_1(6) = \pi_2(5) = \pi_3(8)$, and $7 = \pi_1(7) = \pi_2(6) = \pi_3(5)$, we have

$$c = \pi_1([4, 7]) = \pi_2([3, 6]) = \pi_3([5, 8]).$$

Hence, c is a common interval. It has left end 4 and right end 7 in $\pi_1 = id_9$. Note that the order of the elements of c, as well as its left and right end change in permutation π_3. The set of common intervals C_Π is given by

$$C_\Pi = \{[1, 2], [1, 3], [1, 8], [1, 9], [2, 3], [4, 5], [4, 6], [4, 7], [4, 8], [4, 9], [5, 6]\}.$$

Common intervals have applications in different fields. In a bioinformatical context, common intervals are used to detect possible functional associations between genes. It is assumed that neighboring genes occurring together in different genomes tend to encode functionally interacting proteins [3–5]. Other applications use common intervals to compute the reversal distance between genomes [6], and to define a similarity measure for gene order permutations [7].

In the context of combinatorial optimization, genetic algorithms using subtour exchange crossover based on common intervals have been proposed for sequencing problems such as the traveling salesman problem or the single machine scheduling problem [8–10]. In recent work, Bergeron et al. [11] developed an alternative way to compute common intervals based on generating families of intervals, and applied their approach to the classical graph problem of modular decomposition. Heber and Savage [12] generalized the concept of common intervals to labeled trees.

A related problem is the *consecutive arrangement problem*, defined as follows [13–15]: Given a finite set X and a collection $\mathcal{S}$ of subsets of X, find all permutations of X where the members of each subset $S \in \mathcal{S}$ occur consecutively. Finding all common intervals of a set of permutations reverses this problem.

Uno and Yagiura [16] presented three algorithms for finding all common intervals of $k = 2$ permutations π_1 and π_2 of N: two simple $O(n^2)$-time algorithms and one more complicated algorithm with optimal $O(n+z)$ running time where $z \leq \binom{n}{2}$ is the number of common intervals of π_1 and π_2.

The main result of this paper is a non-trivial generalization of Uno and Yagiura's algorithm to $k \geq 2$ permutations, yielding an optimal $O(kn+z)$-time and $O(n)$-space algorithm where z is the number of common intervals of the k permutations. Our approach relies on restricting the set of all common intervals C_Π to a smaller subset of *irreducible* intervals I_Π, from which C_Π can be reconstructed. While the number of common intervals can be as large as $\binom{n}{2}$, we show that $1 \leq |I_\Pi| \leq n-1$. Furthermore, if we assume that any permutation is selected as one of the k input permutations with probability $1/n!$, then the expected number of common intervals is $O(1)$.

Our algorithm to compute all common intervals consists of two steps. First we compute I_Π in optimal $O(kn)$ time, i.e., in time proportional to the input size. Then, we reconstruct C_Π from I_Π in $O(z)$ time, i.e., in time proportional to the output size. Both steps use $O(n)$ additional space. A simple modification of this procedure allows us also to handle multichromosomal and circular permutations.

The rest of this paper is organized as follows: In Section 2, we give an outline of Uno and Yagiura's approach for two permutations. Then, in Section 3, we further investigate the structure of common intervals and define irreducible intervals. Section 4 shows how to construct the set of all common intervals of $k \geq 2$ permutations, provided that the irreducible intervals are given, and Section 5 describes our algorithm for computing the irreducible intervals. Sections 6, 7, and 8 generalize this approach to multichromosomal permutations, circular permutations, and arbitrary mixtures of both types. In the Appendix, we investigate the expected number of common intervals of random permutations and show some computational running time experiments.

This paper is an extended version of two conference papers, [1] and [2].

2 Finding All Common Intervals of Two Permutations

In order to keep this paper self-contained, we briefly recall the algorithm RC (short for *Reduce Candidate*) of Uno and Yagiura [16], that finds all z common intervals of $k=2$ permutations $\pi_1 = id_n$ and π_2 of $N = \{1, 2, \ldots, n\}$ in $O(n+z)$ time and $O(n)$ space. For the correctness and analysis of the algorithm we refer to [16]. Like in Uno and Yagiura's original algorithm RC we supply our algorithm, here and in later sections of this paper, with $\pi_1 = id_n$ and π_2^{-1}. (As usual, π_2^{-1} denotes the inverse of permutation π_2, i.e. $\pi_2^{-1}(i) = j$ means that $i \in N$ is located in the j-th position of π_2.) This 'coordinate change' does not alter the set of common intervals, but allows us to compute all intervals directly with respect to the index set of $\pi_1 (= id_n)$.

An easy test if an interval $[x, y]$, $1 \leq x < y \leq n$, is a common interval of $\Pi = (\pi_1, \pi_2)$ is based on the following functions:

$$\begin{aligned} l(x,y) &:= \min \pi_2^{-1}([x,y]), \\ u(x,y) &:= \max \pi_2^{-1}([x,y]), \\ f(x,y) &:= u(x,y) - l(x,y) - (y-x). \end{aligned}$$

Since $f(x, y)$ counts the number of elements in $\pi_2([l(x, y), u(x, y)]) \setminus [x, y]$, an interval $[x, y]$ is a common interval of Π if and only if $f(x, y) = 0$. A simple algorithm to find C_Π is to test for each pair of indices (x, y) with $1 \le x < y \le n$ if $f(x, y) = 0$, yielding a naive $O(n^3)$ time or, using running minima and maxima, a slightly more involved $O(n^2)$-time algorithm.

The main idea of Algorithm RC is to save the time for testing $f(x, y) = 0$ for some pairs (x, y) by eliminating *wasteful* candidates for y.

Definition 1 For a fixed x, a right interval end $y > x$ is called *wasteful* if it satisfies $f(x', y) > 0$ for all $x' \le x$.

In Algorithm RC (Algorithm 1), the common intervals are found with the help of a data structure Y, which uses a doubly-linked list *ylist* to store right interval end candidates y for each given left interval end x. The list items are sorted in increasing order of their values. Additionally, two doubly-linked lists *llist* and *ulist* implement the functions l and u. They are used to compute $f(x, y) = u(x, y) - l(x, y) - (y - x)$ efficiently, and to update *ylist*. For any fixed x, the interval $[x + 1, n]$ is partitioned into subintervals $[y_0 = x + 1, y_1 - 1], [y_1, y_2 - 1], \dots, [y_{r-1}, y_r - 1 = n]$ where $u(x, y') = u(x, y'')$ if and only if both y' and y'' are in the same subinterval. The *ulist* reflects this partition. Each list item stores the corresponding subinterval and the value $u(x, y)$ for the y which the subinterval includes. The *llist* is defined similarly.

To be able to compute the values of $l(x, y)$ and $u(x, y)$ quickly, each item y_i of *ylist* has a pointer to the corresponding item of *llist* and *ulist* that includes y_i. Conversely, it will sometimes be necessary to access from an interval $[y, y']$ in *llist* or *ulist* the element of *ylist* that corresponds to its *end*, denoted $end([y, y']) := y'$. Therefore, any such interval $[y, y']$ has an *interval end pointer* that links to the element y' of *ylist*.

Pseudocode of algorithm RC is given in Algorithm 1. We use the list operations $L.head$ for the first element of list L, $L.succ(e)$ for the successor and $L.pred(e)$ for the predecessor of element e in L.

Algorithm 1 (Reduce Candidate, RC)

Input: A family $\Pi = (\pi_1 = id_n, \pi_2)$ of two permutations of $N = \{1, \dots, n\}$.
Output: The set of all common intervals C_Π.

```
 1: compute $\pi_2^{-1}$
 2: initialize $Y$, ulist, and llist using $\Pi' = (\pi_1 = id_n, \pi_2^{-1})$
 3: for $x = n - 1$ down to 1 do
 4:     $y \leftarrow ylist.head$
 5:     update ulist and llist using $\Pi'$     // (see Algorithm 2)
 6:     update $Y$ using $\Pi'$     // (see Algorithm 3)
 7:     while ($y \leftarrow ylist.succ(y)$) defined and $f(x, y) = 0$ do
 8:         output $[x, y]$
 9:     end while
10: end for
```

In the first step of Algorithm 1, we compute π_2^{-1} in order to use $\Pi' = (\pi_1 = id_n, \pi_2^{-1})$ instead of Π for all following initialization and update steps. We initialize *ylist* with the right interval end candidate n. The lists *llist* and *ulist* are both initialized with the interval $[n, n]$, and its values $u(n-1, n)$ and $l(n-1, n)$, respectively. We also add the corresponding pointers from *ylist* to *llist* and *ulist* and back.

Subsequently, a counter x (corresponding to the currently investigated left interval end) runs from $n-1$ down to 1. In each iteration, we update the lists *ulist*, *llist* (line 5), *ylist* (line 6), and we compute all common intervals with left end x (lines 7–9). Below, we show the algorithms involved for the case where $\pi_2^{-1}(x) > \pi_2^{-1}(x+1)$. The other case where $\pi_2^{-1}(x) < \pi_2^{-1}(x+1)$ is treated in a symmetric way: *ulist* is exchanged with *llist*, function u is exchanged with function l and the inequalities (except for the ones in Algorithm 3, line 4) are reversed.

The update of *ulist* and *llist* (Algorithm 2) is performed as follows. First, we prepend $[x, x]$ at the head of *llist*. Then, we find y^* which is maximum among y satisfying $u(x+1, y) < u(x, y)$ by traversing *ulist* and comparing the value of the *ulist* items with $\pi_2^{-1}(x)$. Subsequently, we delete all items from *ulist* which include some y satisfying $u(x+1, y) < u(x+1, y^*)$. Finally, the *ulist* item which includes y^* is changed to $[x, y^*]$, and its value is set to $u(x, y^*)$.

Algorithm 2 (Update of *ulist* and *llist* in line 5 of Algorithm 1)

```
1: prepend $[x, x]$ at the head of $llist$
2: $y^* \leftarrow \max\{y \mid u(x+1, y) < u(x, y)\}$
3: while $y \in ulist.head$ and $u(x+1, y) < u(x+1, y^*)$ do
4:     delete $ulist.head$
5: end while
6: $ulist.head \leftarrow [x, y^*]$
7: $ulist.head.value \leftarrow u(x, y^*)$
```

Afterwards, the data structure Y is updated (Algorithm 3). Using the above computed y^*, we first remove wasteful *ylist* items in lines 1–6. This is called TRIMMING_YLIST in [16]. Removing a wasteful *ylist* item just means adjusting the list pointers. The list item is not deleted, because the list item might still be the target of *llist* and *ulist* pointers. Then, in line 7, a list item containing the value x is prepended at the head of *ylist*. Note that after the update of *ulist.head* in Algorithm 2, lines 6–7, the references of *ylist* elements that originally pointed to this item need not be changed since the object they refer to is still the same, one of the key points in the complexity analysis of the algorithm (see [16]).

Uno and Yagiura show that in each iteration step x, the function $f(x, y)$ is monotonically increasing for the elements y remaining in *ylist*. Based on this result, in lines 7–9 of Algorithm 1, all common intervals with left end x are efficiently found by evaluating $f(x, y)$, letting the right end y run left-to-right through *ylist*, until an index y is encountered with $f(x, y) > 0$.

Algorithm 3 (Update of data structure Y in line 3 of Algorithm 1)

```
1: while $y \leftarrow ylist.head$ and $u(x+1,y) < u(x+1,y^*)$ do
2:     remove $y$ from ylist
3: end while
4: while $y_i, y_{i+1}$ are adjacent items of ylist with $y_i \le y^* < y_{i+1}$, and $f(x,y_i) > f(x,y_{i+1})$ do
5:     remove $y_i$ from ylist
6: end while
7: prepend $x$ at the head of ylist
```

Example 1 (continued) To illustrate Algorithm 1 we apply it to $\Pi = (\pi_1, \pi_2)$, where $\pi_1 = id_9$ and $\pi_2 = (9, 8, 4, 5, 6, 7, 1, 2, 3)$ as above. We show the evolution of data structure Y during iterations $x = 3$ and $x = 2$.

We have $\pi_2^{-1} = (7, 8, 9, 3, 4, 5, 6, 2, 1)$. At iteration $x = 3$, the algorithm has already processed $x = 8, 7, \ldots, 4$, and it has reported the common intervals $[8, 9]$ in iteration $x = 8$, $[6, 7]$ in iteration $x = 6$, $[5, 6]$ and $[5, 7]$ in iteration $x = 5$, and $[4, 5]$, $[4, 6]$, $[4, 7]$, $[4, 8]$, and $[4, 9]$ in iteration $x = 4$. So far, no wasteful *ylist* element has been removed. The data structure Y after iteration $x = 4$ is shown in Fig. 1(a). The *ylist* is shown in the center, with *ulist* above and *llist* below.

In the update for $x = 3$, we have the case $\pi_2^{-1}(x) > \pi_2^{-1}(x + 1)$. Thus the interval $[3, 3]$ with value 9 is prepended to *llist*. We get that $y^* = 9$ is the maximal y where $u(x, y) > u(x + 1, y)$. Therefore, all the intervals in *ulist* are deleted and replaced by the interval $[3, 9]$ with value 9. In Algorithm 3 the values $y = 4$, $y = 5$ and $y = 6$ are detected as wasteful and removed from *ylist*. Since $f(3, 7) = 2$, no common intervals are reported in this phase. (We also have $f(3, 8) = 2$ and $f(3, 9) = 2$, but these are not even checked, since the function f is known to be nondecreasing for the remaining elements of *ylist*.) Finally, $x = 3$ is prepended to *ylist*. The resulting data structure is shown in Fig. 1(b).

In the update for $x = 2$ we have the case $\pi_2^{-1}(x) < \pi_2^{-1}(x + 1)$. Thus the interval $[2, 2]$ with value 8 is prepended at the head of *ulist*. We get that $y^* = 3$ is the maximal y where $l(x, y) < l(x + 1, y)$. Therefore the interval $[3, 3]$ at the head of *llist* is deleted and replaced by the interval $[2, 3]$ with value 8. In the update for $x = 2$, Algorithm 3 does not remove any items from *ylist*. We get $f(2, 3) = 0$. The elements 4, 5 and 6 have already been removed from *ylist* in step $x = 3$. For the next element 7 in *ylist* we obtain $f(2, 7) = 9 - 3 - (7 - 2) = 1$. Since function f is nondecreasing, we have $f(2, y) > 0$ for the remaining elements y of *ylist*. So we report the common interval $[2, 3]$. Finally, $x = 2$ is prepended to *ylist*. The resulting data structure after this phase is shown in Fig. 1(c).

3 Irreducible Intervals

In this section we define the set of *irreducible intervals*, prove some of their structural properties, and show how they can be used to reconstruct all common intervals. We start by characterizing the structure of the set of common intervals.

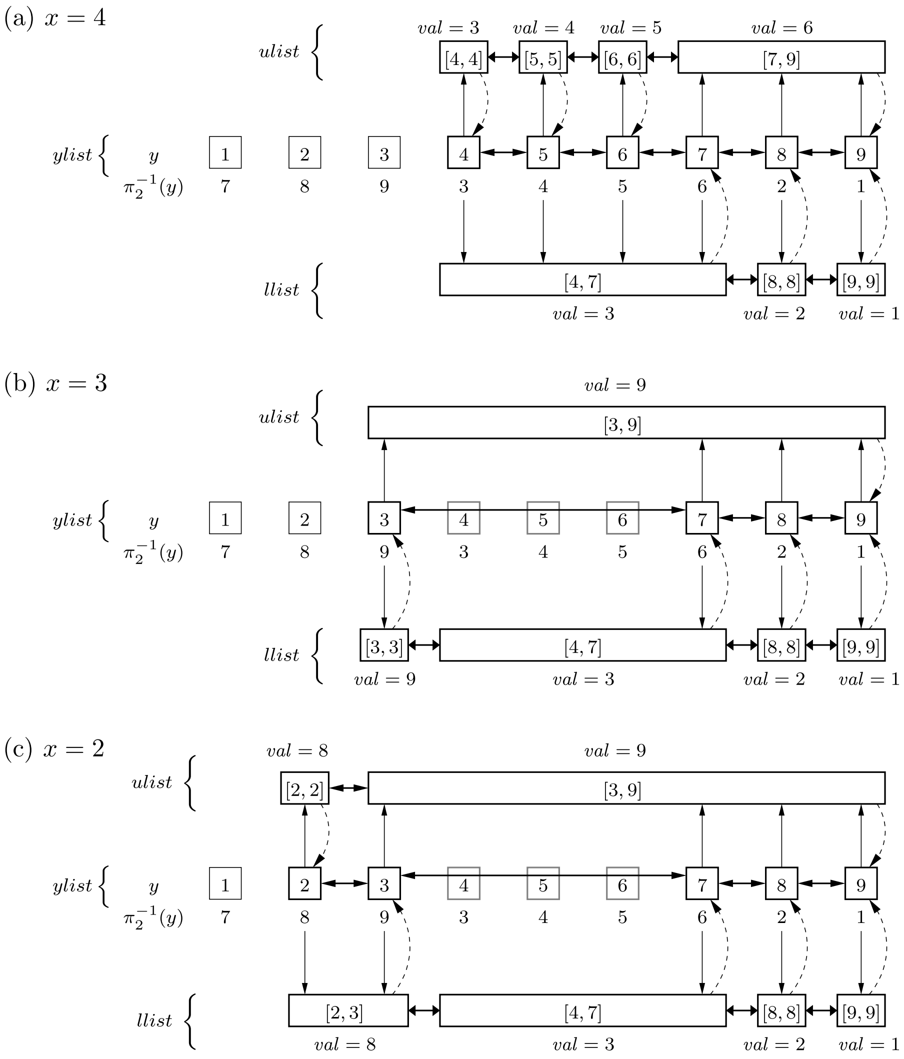

Fig. 1 Sketches of the data structure Y after processing (**a**) $x = 4$, (**b**) $x = 3$, and (**c**) $x = 2$ for permutations $\pi_1 = id_9$ and $\pi_2^{-1} = (7, 8, 9, 3, 4, 5, 6, 2, 1)$. The *ylist* is shown in the center, with *ulist* above and *llist* below. During the update in (**b**), the *ylist* items 4, 5, and 6 are removed. The unconnected elements in front of the head of *ylist* (e.g., 1, 2, 3 in (**a**)) are not part of the data structure. They are depicted only to illustrate the complete permutation π_2^{-1}

Lemma 1 *Let Π be a family of permutations. For $c_1, c_2 \in C_\Pi$ we have*

(a) $|c_1 \cap c_2| \geq 2 \Leftrightarrow c_1 \cap c_2 \in C_\Pi$,
(b) $c_1 \cap c_2 \neq \emptyset \Rightarrow c_1 \cup c_2 \in C_\Pi$.

Proof For (a) we note that no element of $(c_1 \cup c_2) \setminus (c_1 \cap c_2)$ can lie between two elements of $c_1 \cap c_2$ in any permutation, thus $c_1 \cap c_2$ is a common interval. For (b) we

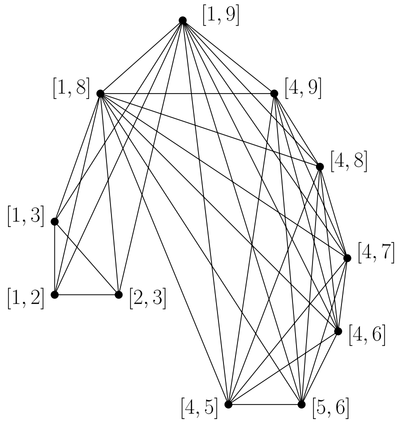

Fig. 2 The common interval graph of permutations Π from Example 1

remark that if $c_1 = \pi_i([l_i^1, u_i^1])$ and $c_2 = \pi_i([l_i^2, u_i^2])$ for $i = 1, \dots, k$, then $c_1 \cup c_2 = \pi_i([\min(l_i^1, l_i^2), \max(u_i^1, u_i^2)])$ for $i = 1, \dots, k$. □

Definition 2 Two common intervals $c_1, c_2 \in C_\Pi$ have a *non-trivial overlap* if $c_1 \cap c_2 \neq \emptyset$, and neither includes the other.

Lemma 2 *Let Π be a family of permutations. For $c_1, c_2 \in C_\Pi$ with non-trivial overlap we have*

$$|c_1 \setminus c_2| \geq 2 \quad \Rightarrow \quad c_1 \setminus c_2 \in C_\Pi.$$

Proof Since c_1 and c_2 overlap non-trivially, no two elements of $c_1 \setminus c_2$ can lie on opposite sides of $c_1 \cap c_2$, otherwise c_2 would not be a common interval. □

Given a family of permutations Π we define the *common interval graph* of Π to be the graph $G_\Pi = (C_\Pi, E_\Pi)$, whose vertex set is the set of common intervals of Π and whose edge set is defined by

$$E_\Pi = \{(c, d) \mid c, d \in C_\Pi, c \neq d, c \cap d \neq \emptyset\}.$$

For a set $V \subseteq C_\Pi$, $G_\Pi[V]$ is the subgraph of G_Π induced by V. Given a set $V \subseteq C_\Pi$ such that $G_\Pi[V]$ is connected, we denote as $\tau(V) = \{i \mid i \in v, v \in V\}$ the *support* of V, which is the union of all intervals of V, and we say that V *generates* $\tau(V)$. By Lemma 1(b), we conclude that $\tau(V) \in C_\Pi$.

Example 1 (continued) The common interval graph of the permutations Π from the Introduction is given in Fig. 2. For example, the common interval $[1, 3]$ is generated by $V = \{[1, 2], [2, 3]\}$.

Definition 3 A common interval c is called *reducible* if there is a set of common intervals $V \subseteq C_\Pi$ such that V generates c and each $d \in V$ is a proper subset of c. If there is no such V, c is called *irreducible*.

This definition partitions the set of common intervals C_Π into the set of reducible intervals and the set of irreducible intervals, denoted I_Π. Clearly, $1 \le |I_\Pi| \le |C_\Pi| \le \binom{n}{2}$.

The following definition and theorem and the subsequent corollary are the basis for the correctness and the complexity analysis of our algorithm in Section 5.

Definition 4 We define a total order on the set of intervals as follows. For intervals $c_1 = [x_1, y_1], c_2 = [x_2, y_2]$ let

$$c_1 < c_2 \quad \Leftrightarrow \quad x_1 > x_2, \quad \text{or} \quad x_1 = x_2, \quad \text{and} \quad y_1 < y_2. \tag{1}$$

Given a family $\Pi = (\pi_1, \dots, \pi_k)$ of permutations of $N = \{1, 2, \dots, n\}$ with $\pi_1 = id_n$ let $\Pi_i := (\pi_1, \dots, \pi_i)$ for $1 \le i \le k$.

For a subset $M \subseteq N$ with $|M| \ge 2$, we denote by $\varphi_{\Pi_i}(M)$ the smallest (w.r.t. the order on intervals defined above) common interval of Π_i which contains M. As an abbreviation, for $j = 1, \dots, n-1$, we write $\varphi_{\Pi_i}(j)$ for $\varphi_{\Pi_i}([j, j+1])$.

Theorem 1 *Given a family $\Pi = (\pi_1, \dots, \pi_k)$ of permutations of $N = \{1, 2, \dots, n\}$ with $\pi_1 = id_n$ we get the following.*

(a) *For a subset $M \subseteq N$ with $|M| \ge 2$ and $1 \le i \le k$, we have*
   (a.i) *$\varphi_{\Pi_i}(M)$ is well defined and unique.*
   (a.ii) *$\varphi_{\Pi_i}(M)$ is the uniquely smallest, w.r.t. cardinality, common interval of Π_i which contains M.*
   (a.iii) *If $c \in C_{\Pi_i}$ with $M \subseteq c$, then $\varphi_{\Pi_i}(M) \subseteq c$.*

(b) *For $1 \le i \le k$ and $1 \le j \le n-1$ we have that $\varphi_{\Pi_i}(j)$ is irreducible.*

(c) *For $1 \le i \le k$ we have $I_{\Pi_i} = \{\varphi_{\Pi_i}(j) \mid j = 1, \dots, n-1\}$.*

(d) *For $1 < i \le k$ and $1 \le j \le n-1$ we have $\varphi_{\Pi_i}(\varphi_{\Pi_{i-1}}(j)) = \varphi_{\Pi_i}(j)$.*

Proof For (a.i) we first remark that $[1, n]$ is a common interval and $M \subseteq [1, n]$, hence $\varphi_{\Pi_i}(M)$ is well defined. The order on the set of intervals, from Definition 4, (1), is a total order, and thus $\varphi_{\Pi_i}(M)$ is unique.

To see (a.ii), note that if there existed a common interval c of Π_i which contains M and has a smaller cardinality than $\varphi_{\Pi_i}(M)$, then, by Lemma 1(a), $c \cap \varphi_{\Pi_i}(M)$ would also be a common interval of Π_i which contains M and $c \cap \varphi_{\Pi_i}(M) < \varphi_{\Pi_i}(M)$, a contradiction. Thus $\varphi_{\Pi_i}(M)$ is minimal w.r.t. both the order on intervals and the cardinality. Uniqueness follows from the fact that the above argument can be applied to any common interval c of Π_i containing M, yielding eventually the unique minimum $\varphi_{\Pi_i}(M)$.

The correctness of (a.iii) follows from (a.ii) and again Lemma 1(a). Since $\varphi_{\Pi_i}(M)$ is minimal w.r.t. cardinality we have $\varphi_{\Pi_i}(M) = \cap_{d \in C_{\Pi_i}, M \subseteq d} d$, therefore $\varphi_{\Pi_i}(M) \subseteq c$.

For (b) we can use (a.ii) to show that $\varphi_{\Pi_i}(j)$ is well defined, unique, and has minimal cardinality of all common intervals of Π_i which contain $[j, j+1]$. Now, assume that $\varphi_{\Pi_i}(j)$ is reducible, i.e. there is a subset $V \subseteq C_{\Pi_i}$ that generates $\varphi_{\Pi_i}(j)$, and each $d \in V$ is a proper subset of $\varphi_{\Pi_i}(j)$. Due to the minimality of $\varphi_{\Pi_i}(j)$, no $d \in V$ contains j and $j+1$ simultaneously. Define $VL := \{d \in V \mid d \subseteq [1, j]\}$, and

$VR := \{d \in V \mid d \subseteq [j+1, n]\}$. Since each $d \in V$ is an interval in $\pi_1 = id_n$, we have $V = VL \dot{\cup} VR$. On the other hand, we have $j, j+1 \in \varphi_{\Pi_i}(j)$, hence there are intervals $d', d'' \in V$ with $j \in d'$ and $j+1 \in d''$, ensuring $VL, VR \neq \emptyset$. Since $d_l \cap d_r = \emptyset$ for any $d_l \in VL, d_r \in VR$, the subgraph $G_{\Pi_i}[V]$ is not connected, and we get a contradiction to our assumption that V generates c.

For (c) we show that each $c \in C_{\Pi_i}$ is either an element of $\{\varphi_{\Pi_i}(j) \mid j = 1, \dots, n-1\}$ and therefore irreducible, or that it is generated by a subset $V \subseteq \{\varphi_{\Pi_i}(j) \mid j = 1, \dots, n-1\}$, where each $d \in V$ is a proper subset of c. In this case, c is reducible. Assume $c = [x, y] \in C_{\Pi_i}$, and set $V := \{\varphi_{\Pi_i}(j) \mid j = x, \dots, y-1\}$. For $j = x, \dots, y-1$ we have $c, \varphi_{\Pi_i}(j) \in C_{\Pi_i}$, and $j, j+1 \in c \cap \varphi_{\Pi_i}(j)$. Using Lemma 1(a) and the minimality of $\varphi_{\Pi_i}(j)$ we conclude $\varphi_{\Pi_i}(j) \subseteq c$, hence $\bigcup_{j=x,\dots,y-1} \varphi_{\Pi_i}(j) = c$. If there is a $\varphi_{\Pi_i}(j^*) \in V$ with $\varphi_{\Pi_i}(j^*) = c$, then we know from (b) that c is irreducible. Otherwise, we conclude that each interval $\varphi_{\Pi_i}(j)$ is a proper subset of c, and $\varphi_{\Pi_i}(j) \cap \varphi_{\Pi_i}(j+1) \neq \emptyset$ for $j = x, \dots, y-2$. Thus $G_{\Pi_i}[V]$ is connected, V generates c, and c is reducible.

For (d) we now show that $\varphi_{\Pi_i}(\varphi_{\Pi_{i-1}}(j)) = \varphi_{\Pi_i}(j)$. By (a.ii), $\varphi_{\Pi_i}(j)$ is the unique smallest (w.r.t. cardinality) common interval containing $j, j+1$ in C_{Π_i}. Since $C_{\Pi_i} \subseteq C_{\Pi_{i-1}}$, we have $\varphi_{\Pi_i}(j) \in C_{\Pi_{i-1}}$. Now we use (a.ii) and (a.iii), instantiated with $i-1$ for i, $M = \{j, j+1\}$, and $c = \varphi_{\Pi_i}(j)$. This yields $\varphi_{\Pi_{i-1}}(j) \subseteq \varphi_{\Pi_i}(j)$. Then we use (a.iii) again with the different instantiation i for i, $M = \varphi_{\Pi_{i-1}}(j)$, $c = \varphi_{\Pi_i}(j)$. (The preconditions are satisfied, since $c \in C_{\Pi_i}$ and $M \subseteq c$ as shown above.) This yields $\varphi_{\Pi_i}(\varphi_{\Pi_{i-1}}(j)) \subseteq \varphi_{\Pi_i}(j)$. Since $\varphi_{\Pi_i}(j)$ is the unique smallest common interval in C_{Π_i} that contains $j, j+1$ (see above), we get $\varphi_{\Pi_i}(\varphi_{\Pi_{i-1}}(j)) = \varphi_{\Pi_i}(j)$ as claimed. □

Corollary 1 *Given a family $\Pi = (\pi_1, \dots, \pi_k)$ of permutations of $N = \{1, 2, \dots, n\}$ we have $1 \leq |I_\Pi| \leq n-1$.*

Example 2 The limits given in Corollary 1 are actually achieved. For $\Pi = (id_n)$ we have $C_\Pi = \{[i, j] \mid 1 \leq i < j \leq n\}$ and $I_\Pi = \{[i, i+1] \mid 1 \leq i < n\}$. For $n = 2t$ and $\Pi = (id_n, (1, t+1, 2, t+2, \dots, t, 2t))$ we have $C_\Pi = I_\Pi = \{[1, n]\}$.

Lemma 3 *For any irreducible interval $c \in I_\Pi$ there are at most two irreducible intervals that have a non-trivial overlap with c. We call them the* neighbors *of c. If c has two neighbors $a, b \in I_\Pi$, $a \neq b$, then c contains exactly one left and one right end of them, and $a \cap b = \emptyset$.*

Proof Let $c = [l, r]$ be an irreducible interval that has a non-trivial overlap with the irreducible intervals $a = [l_a, r_a]$ and $b = [l_b, r_b]$. Without restriction we can assume that $l_a \leq l_b$ (otherwise exchange a and b). We now do an analysis of all cases of how the intervals a, b and c could overlap, and prove that only the case described in the lemma above is possible.

Case 1. First, suppose that c contains the left ends of both a and b. We investigate the possible interval end configurations (see Fig. 3; note that the non-trivial overlap of a and b with c implies $l < l_a, l_b \leq r < r_a, r_b$).

(i) Assume $l < l_a = l_b \leq r < r_a < r_b$. By Lemma 2, we conclude that $d := b \setminus c$ is a common interval. In contradiction to being irreducible, b is generated by

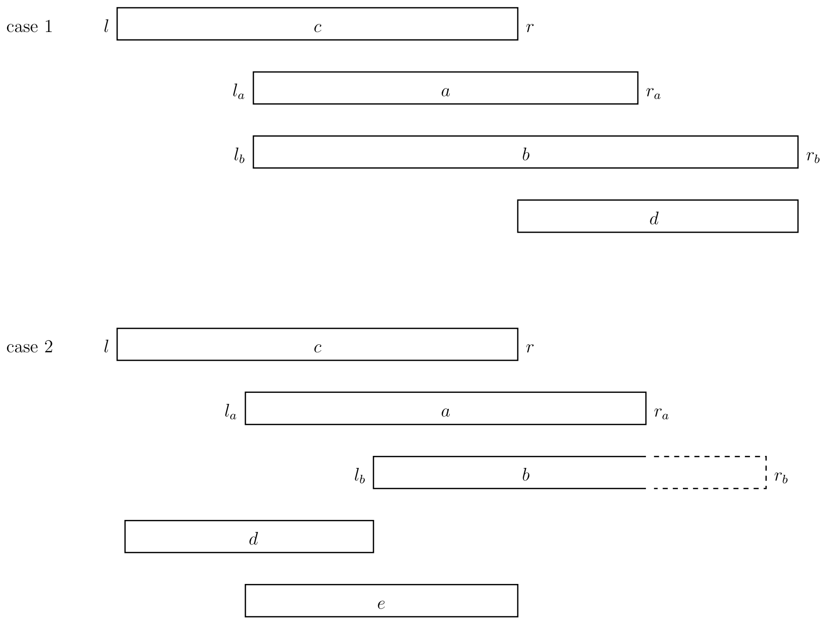

Fig. 3 Visualization of the different interval configurations

the common intervals a and d. The case $l < l_a = l_b \leq r < r_b < r_a$ is treated analogously.

(ii) Assume $l < l_a < l_b \leq r$. By Lemma 2 and Lemma 1(a) we conclude that $d := c \setminus b$ and $e := a \cap c$ are common intervals. In contradiction to being irreducible, c is generated by the common intervals d and e.

Case 2. The case where c contains the right ends of the intervals a and b is treated in a way that is symmetric with Case 1, with right and left ends exchanged.

Case 3. The remaining scenario is that c contains the right end of a and the left end of b (the reverse case is impossible, since we assumed $l_a \leq l_b$). It remains to consider whether a can overlap with b. If $a \cap b \neq \emptyset$, this yields a contradiction (by applying Case 1 to a), because a overlaps non-trivially with c and b, and a contains the left ends of both of these intervals.

We conclude that there cannot exist more than two intervals which have a non-trivial overlap with c, otherwise c would contain at least two left or two right interval ends, a contradiction to Cases 1 and 2. If there are two intervals a, b having a non-trivial overlap with c, the interval c contains exactly one left and one right end of a and b, and $a \cap b = \emptyset$ (Case 3). $\square$

A sequence $p = (c_1, \ldots, c_{\ell(p)})$ of irreducible intervals $c_1, \ldots, c_{\ell(p)} \in I_\Pi$ is a *chain* of length $\ell(p)$ if every two successive intervals c_j, c_{j+1}, $j = 1, \ldots \ell(p) - 1$ have a non-trivial overlap. For $1 \leq i \leq j \leq \ell(p)$, we call $p[i, j] := (c_i, \ldots, c_j)$ a *subchain of p* of length $j - i + 1$. A chain that cannot be extended to its left or right is a *maximal chain*.

For the following lemma, remember from Section 3 that for a set $V \subseteq C_\Pi$, $\tau(V) = \{i \mid i \in v, v \in V\}$ denotes the support of V.

Lemma 4 *Let Π be a family of permutations, C_Π its set of common intervals, and I_Π the corresponding set of irreducible intervals.*

(a) *I_Π can be partitioned into a uniquely defined set of maximal chains P.*
(b) *Given $p_1, p_2 \in P$ with $p_1 \neq p_2$, then either $\tau(p_1)$ and $\tau(p_2)$ are disjoint, or one chain is completely contained in exactly one element or in the intersection of two consecutive elements of the other chain.*
(c) *There is a bijection between the set of common intervals C_Π and the set of subchains of maximal chains in P. Each irreducible interval corresponds to a subchain of length one, and each reducible interval is generated by a subchain of length two or more.*

Proof To see (a), observe that Lemma 3 implies that for any irreducible interval c there is at most one irreducible interval a to the left and one irreducible interval b to the right that overlap c non-trivially. This guarantees that each irreducible interval is part of a uniquely defined maximal chain.

To see (b) we note that $\tau(p_1)$ and $\tau(p_2)$ are either disjoint or have a non-empty intersection. In the latter case, we conclude the existence of intervals $c_1 \in p_1, c_2 \in p_2$ with $c_1 \neq c_2$ and $c_1 \cap c_2 \neq \emptyset$. By Lemma 3 and the maximality of p_1 and p_2, no interval of p_1 can overlap non-trivially with an interval of p_2, and we conclude that c_1 and c_2 are nested. W.l.o.g. assume c_1 is a proper subset of c_2. Now consider a neighbor c^* of c_1 in p_1. Since c_1 and c^* have non-trivial overlap, we conclude that $c^* \cap c_2 \neq \emptyset$. Using again the fact that no interval of p_1 can overlap non-trivially with an interval of p_2, we conclude that either $c^* \subset c_2$, or $c_2 \subset c^*$. Since c_1 and c^* are neighbors, and $c_1 \subset c_2$, we cannot have $c_2 \subset c^*$. Hence, c^* is a proper subset of c_2. Iterating this argument we conclude that every interval of p_1 is a proper subset of c_2, and that $\tau(p_1)$ is contained in c_2. Now assume $\tau(p_1)$ is also contained in another interval $c_3 \neq c_2$ of p_2. Since c_2 and c_3 are both intervals of the same chain p_2, and both contain $\tau(p_1)$, they have a non-trivial overlap. By Lemma 3, c_3 is a neighbor of c_2. Also by Lemma 3 we know that there are at most two neighbors of c_2, and that these neighbors do not intersect. Therefore there cannot be a third interval in p_2 which contains $\tau(p_1)$. This yields the existence of one single, or two intersecting irreducible intervals of p_2 that include $\tau(p_1)$ completely, while all other elements of p_2 are disjoint from $\tau(p_1)$.

For (c), we remark that the irreducible intervals of any subchain of a chain in P correspond to a connected subgraph of G_Π, therefore they generate a common interval. To see that each common interval $c = [x, y] \in C_\Pi$ can be generated in this way, we use the fact that $I_\Pi = \{\varphi_\Pi(j) \mid j = 1, \dots, n-1\}$ from Theorem 1. Set $V := \{\varphi_\Pi(j) \mid j = x, \dots, y-1\}$. We know that $G[V]$ is connected, and $\tau(V) = c$. Set $V' := V \setminus \{v \in V \mid \exists v' \in V, v \subset v'\}$. By construction, $G[V']$ remains connected, and $\tau(V') = c$. If $|V'| = 1$ then c is irreducible by Theorem 1(b). Otherwise, c is reducible, and we use Lemma 3 to show that V' corresponds to the chain of irreducible intervals generating c. To see that the above chain representation is unique, we assume that c is generated by two different subchains $p_1[i_1, j_1]$, and $p_2[i_2, j_2]$. Since different subchains of the same maximal chain generate different common intervals we conclude that $p_1 \neq p_2$, and obtain a contradiction from (b). □

For a common interval $c \in C_\Pi$, we count the number of irreducible intervals that properly contain c and call this number the *nesting level* of c.

Lemma 5 *Let Π be a family of permutations, $c \in C_\Pi$ a common interval, and $p = (c_1, \ldots, c_{\ell(p)})$ a chain of irreducible intervals generating c. The nesting levels of c and all the c_i for $i = 1, \ldots, \ell(p)$ are equal.*

Proof Let d be the nesting level of c and d_i the nesting level of c_i for $i = 1, \ldots, \ell(p)$. Since $c_i \subseteq c$ we have $d_i \geq d$ for $i = 1, \ldots, \ell(p)$. Now assume that $d_i > d$ for some $i \in \{1, \ldots, \ell(p)\}$. Then the chain must consist of at least two elements, i.e. $\ell(p) > 1$, and there must exist an irreducible interval $c^* \not\supset c$ with $c_i \subset c^*$. The latter implies that at least one interval $c_j \in \{c_1, \ldots, c_{\ell(p)}\} \setminus \{c_i\}$ has a non-trivial overlap with c^*. We distinguish the following cases:

1. $j < i$. Then c_j overlaps non-trivially with c_{j+1} and c^*, and contains the left ends of these intervals.
2. $j > i$. Then c_j overlaps non-trivially with c_{j-1} and c^*, and contains the right ends of these intervals.

In both cases, we obtain a contradiction to Lemma 3, showing that such a c^* cannot exist and hence c_i has the same nesting level as c. □

Example 1 (continued) For $\Pi = (\pi_1, \pi_2, \pi_3)$ as in Section 1, the irreducible intervals are

$$I_\Pi = \{[1,2], [1,8], [2,3], [4,5], [4,7], [4,8], [4,9], [5,6]\}.$$

The maximal chains are $([1,8], [4,9])$, $([1,2], [2,3])$, $([4,8])$, $([4,7])$, and $([4,5], [5,6])$. The reducible intervals are generated as follows:

$$[1,3] = [1,2] \cup [2,3],$$
$$[1,9] = [1,8] \cup [4,9],$$
$$[4,6] = [4,5] \cup [5,6].$$

A sketch of the structure of maximal chains of irreducible intervals and their nesting levels is shown in Fig. 4.

4 Finding All Common Intervals of k Permutations

In Algorithm 4 we describe an $O(n + |C_\Pi|)$ time algorithm to reconstruct the set C_Π of common intervals of a family of permutations Π from its set I_Π of irreducible intervals. In this section we assume that set I_Π of irreducible intervals is already given. We will show in Section 5 how to construct I_Π in $O(kn)$ time.

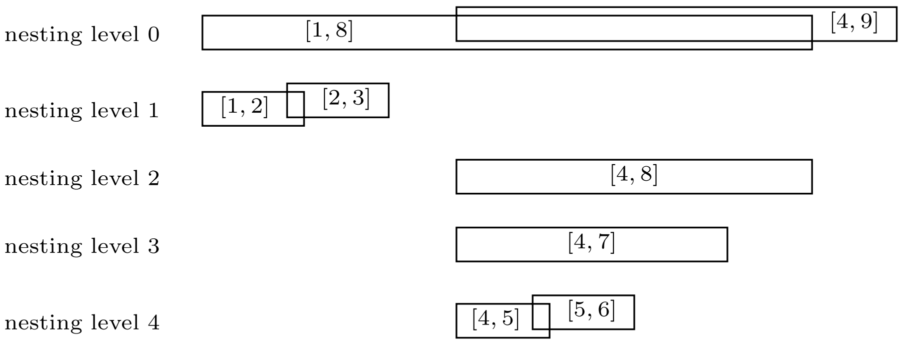

Fig. 4 Visualization of the irreducible intervals in I_Π and their nesting levels

Algorithm 4 (Reconstruct C_Π from I_Π)

Input: A set of irreducible intervals I_Π.

Output: The corresponding set of all common intervals C_Π.

```
1: partition $I_\Pi$ into maximal chains $p_1, p_2, \ldots$
2: for each $p_m = (c_1, \ldots, c_{\ell(p_m)})$ do
3:     output $\tau(p_m[i, j])$ for all $1 \le i \le j \le \ell(p_m)$
4: end for
```

First, Algorithm 4 partitions I_Π into maximal chains (line 1). This is done in two steps, each of which takes only $O(n)$ time (and space).

Step 1: We create a sorted linked list which contains every irreducible interval in I_Π exactly twice. The sorting criterion is as follows. For $k = 1, 2, \ldots, n$ the list first contains all irreducible intervals with *left end* k, in *decreasing* order of their length. Then it contains all irreducible intervals with *right end* k, in *increasing* order of their length. Then this is repeated for the next higher k. (In the special cases of $k = 1$ and $k = n$ there are no irreducible intervals with right end k, and left end k, respectively.) It follows that every irreducible interval $[k_1, k_2]$ appears exactly twice in this list; first due to its left end when k reaches k_1 and later due to its right end when k reaches k_2.

Step 2: We traverse this list, using a pushdown stack of intervals, and create all maximal chains from it (see below).

For Step 1, we sort the irreducible intervals in $O(n)$ time according to the order defined above. This is achieved as follows. We first sort all intervals by their length using bucket sort. Since there are at most $n-1$ irreducible intervals, and each interval length is in the range $[2, n]$, this can be done in $O(n)$ worst-case time using $n-1$ buckets. Then, we initialize for each possible interval border $k = 1, 2, \ldots, n$ a linked list. In order of decreasing interval length, we insert all interval end points into their corresponding list. Subsequently, we insert all interval startpoints, now in order of increasing interval length, into the lists. Every entry in the list is marked with information whether it represents a start point or an end point. Finally, we concatenate all linked lists in increasing order of $k = 1, 2, \ldots, n$. Since there are at most n lists, and at most $n-1$ intervals, the whole procedure can be performed in $O(n)$ worst-case time.

For Step 2, we traverse the resulting list of interval ends from left to right. Whenever we encounter a start point, we push the corresponding interval on a stack. Whenever we encounter an end point, we remove the corresponding interval from the stack. (Note that such an interval is not necessarily at the top of the stack as discussed below.) Using Lemma 4 and the fact that we process the sorted interval end points from left to right, we argue that at the time an interval c is removed, the stack only contains intervals which either include c completely, or overlap with its right end. By Lemma 3, there is at most one interval (c's right neighbor) which overlaps only with the right end of c. Since all intervals which include c have been pushed on the stack before c, the height of c in the stack corresponds to its nesting level. Interval c is either at the top of the stack, corresponding to a chain end, or it is directly below the top, and the top of the stack is c's right neighbor. This ensures that we never remove an element deep in the stack, hence this data structure can be realized by a normal stack. With each removal we report nesting level and chain structure. Since removing the top or the second highest element of a stack can be performed in constant time, I_Π is partitioned into maximal chains in overall $O(n)$ time. Since there are at most $n-1$ intervals, the stack height and memory used are bounded by $O(n)$.

By Lemma 4(c), the common intervals of Π correspond to exactly the subchains of the maximal chains of irreducible intervals in I_Π. Thus, we create C_Π by generating all subchains of the maximal chains (lines 2–4 of Algorithm 4). Due to the bijection between subchains and common intervals, this takes $O(|C_\Pi|)$ time. Altogether, Algorithm 4 takes $O(n+|C_\Pi|)$ time in total.

Example 1 (continued) In our running example $\Pi = (\pi_1, \pi_2, \pi_3)$ with $\pi_1 = id_9$, $\pi_2 = (9, 8, 4, 5, 6, 7, 1, 2, 3)$, and $\pi_3 = (1, 2, 3, 8, 7, 4, 5, 6, 9)$, the irreducible intervals are:

$$[5,6], [4,5], [4,7], [4,8], [4,9], [2,3], [1,2], [1,8].$$

Now we partition them into maximal chains, using the construction above.

First we apply Step 1 and create the list which contains every irreducible interval twice, in the desired order. The left-end entries are printed in bold face and the right-end entries in normal face. Thus, in this list, every intervals first appears in bold face and later in normal face.

$$[\mathbf{1},\mathbf{8}], [\mathbf{1},\mathbf{2}], [\mathbf{2},\mathbf{3}], [1,2], [2,3], [\mathbf{4},\mathbf{9}], [\mathbf{4},\mathbf{8}], [\mathbf{4},\mathbf{7}], [\mathbf{4},\mathbf{5}], [\mathbf{5},\mathbf{6}],$$
$$[4,5], [5,6], [4,7], [4,8], [1,8], [4,9].$$

Then we apply Step 2 to this list. We push the first three left-end entries of the list onto the stack, which then has the form $[1,8], [1,2], [2,3]$ (bottom-to-top). Then we encounter the right-end entry $[1,2]$, which is at the second stack position from the top. This indicates that the current stack-top element $[2,3]$ is the right neighbor of $[1,2]$. We then remove the element $[1,2]$ from the stack and obtain the new stack $[1,8], [2,3]$. We report the beginning of a chain $[1,2], [2,3]$ at nesting level 1 (the current height of the stack without $[2,3]$, i.e., only interval $[1,8]$ includes this chain). The next element in the list is the right-hand entry $[2,3]$, which is also the current head of the stack. This indicates that this chain ends here. So we remove $[2,3]$ from the stack and report the chain $[1,2]$, $[2,3]$. By processing the rest of the list in the same way, we obtain the chains $[4,5]$, $[5,6]$ (at level 4), $[4,7]$ (at level 3), $[4,8]$ (at level 2) and $[1,8]$, $[4,9]$ at level 0, as shown in Fig. 4.

5 Finding All Irreducible Intervals of k Permutations

In this section we present an algorithm that finds all irreducible intervals of a family $\Pi = (\pi_1, \pi_2, \dots, \pi_k)$ of $k \geq 1$ permutations of $N = \{1, \dots, n\}$ in $O(kn)$ time.

For $1 \leq i \leq k$, set $\Pi_i := (\pi_1, \dots, \pi_i)$. Starting with $I_{\Pi_1} = \{\varphi_{\Pi_1}(j) = [j, j+1] \mid 1 \leq j < n\}$, we successively compute $I_{\Pi_i} = \{\varphi_{\Pi_i}(c) \mid c \in I_{\Pi_{i-1}}\}$ for $i = 2, \dots, k$, using Theorem 1(b.ii) and 1(c). In the following we will show that $I_{\Pi_i} = \{\varphi_{\Pi_i}(c) \mid c \in I_{\Pi_{i-1}}\}$ can be computed in $O(n)$ time and space, yielding the $O(kn)$-time complexity to compute I_Π $(= I_{\Pi_k})$. We use an extended version of Algorithm RC (Algorithm 5) where we build the data structure Y for π_1 and π_i, complemented by a data structure S that is derived from $I_{\Pi_{i-1}}$.

First we give some auxiliary definitions and lemmas (Section 5.1), and describe the data structure S (Section 5.2). Then we describe the algorithm (Section 5.3), prove its correctness (Section 5.4) and establish its complexity (Section 5.5).

5.1 Auxiliary Definitions and Lemmas

In order to describe the algorithm and its data structures, and to reason about its correctness, we need some auxiliary results which are presented in this section.

Remember from Definition 1 that for an index $x \in N$, a right interval end $y > x$ is called wasteful with respect to two permutations $\pi_1 = id$ and π_2 and x if it satisfies $f(x', y) > 0$ for all $x' \leq x$, otherwise it is called non-wasteful. We will extend this latter notion now to common intervals.

Definition 5 Given a fixed $x \in N$, we call a common interval of $C_{\Pi_{i-1}}$ *non-wasteful* if its right border is non-wasteful with respect to $\{\pi_1, \pi_i\}$ and x. For $a \subseteq N$ with $|a| \geq 2$, denote by $SC(a, x)$ the set of non-wasteful (with respect to $\{\pi_1, \pi_i\}$, and x) common intervals of $C_{\Pi_{i-1}}$ which include a, and have a left end smaller than or equal to x. We define the *non-wasteful hull* $c(a, x)$ of a as $c(a, x) = \bigcap_{c \in SC(a,x)} c$. In particular, for $a = \varphi_{\Pi_{i-1}}(j)$, $j \in \{1, \dots, n-1\}$, we simplify the notation to $SC(j, x)$, and $c(j, x)$.

We note that if the right interval end of a is smaller than x this definition does not necessarily imply $x \in c(a, x)$.

Lemma 6 *The interval $c(a, x)$ is well defined, unique, non-wasteful, and contains a.*

(a) *For any non-wasteful common interval $b = [x', y] \in C_{\Pi_{i-1}}$ with $x' \leq x$ we have $c(a, x) \subseteq b \Leftrightarrow a \subseteq b$.*
(b) *For $a, b \subseteq N$ with $|a| \geq 2$ and $a \subseteq b$ we have $c(a, x) \subseteq c(b, x)$.*

(c) *Let* $\varphi_{\Pi_i}(j) = [x_{i,j}, y_{i,j}]$. *We have*
 (c.i) $c(j,x) \subseteq \varphi_{\Pi_i}(j)$ *for* $x \geq x_{i,j}$.
 (c.ii) $c(j, x_{i,j}) = \varphi_{\Pi_i}(j)$.

Proof To show that $c(a,x)$ is well defined and unique we note that $SC(a,x)$ is not empty, for example $[1,n] \in SC(a,x)$. Since the intersection of any two non-wasteful intervals of $SC(a,x)$ with left end smaller than or equal to x is a non-wasteful common interval which includes a and has a left interval end smaller than or equal to x, the interval $c(a,x) = \bigcap_{c \in SC(a,x)} c$ is the uniquely defined smallest non-wasteful common interval of $C_{\Pi_{i-1}}$ with $a \subseteq c(a,x)$, and a left interval end smaller than or equal to x.

To show (a) we note that by definition of $c(a,x)$ we have $a \subseteq c(a,x)$. If $c(a,x) \subseteq b$ this implies $a \subseteq b$. To show the reverse, assume $a \subseteq b$. We have $b \in SC(a,x)$, and since $c(a,x) = \bigcap_{c \in SC(a,x)} c$ we get $c(a,x) \subseteq b$. For (b) we note that $a \subseteq b$ implies $SC(b,x) \subseteq SC(a,x)$, hence $\bigcap_{c \in SC(a,x)} c \subseteq \bigcap_{c \in SC(b,x)} c$. To show (c.i) we argue that by definition we have $\varphi_{\Pi_i}(j) \in C_{\Pi_{i-1}}$ and $\varphi_{\Pi_i}(j) \in C_{\Pi_i}$. For $x \geq x_{i,j}$ the interval $\varphi_{\Pi_i}(j)$ is non-wasteful (with respect to $\{\pi_1, \pi_i\}$, and x) and we have $\varphi_{\Pi_i}(j) \in SC(j,x)$, thus $c(j,x) \subseteq \varphi_{\Pi_i}(j)$.

To show (c.ii) we assume $c(j, x_{i,j}) = [x', y']$. Using $c(j, x_{i,j}) \subseteq \varphi_{\Pi_i}(j)$ from (c.i) we get $x' \geq x_{i,j}$, and $y' \leq y_{i,j}$. By definition, we have $c(j, x_{i,j}) = \bigcap_{c \in SC(j, x_{i,j})}$ and each $c \in SC(j, x_{i,j})$ has a left border $x \leq x_{i,j}$. Since $\bigcap_{c \in SC(j, x_{i,j})} \neq \emptyset$ and $[x_{i,j}, y_{i,j}] \in SC(j, x_{i,j})$ we conclude $x_{i,j} \in c(j,x)$. This proves $x' = x_{i,j}$. Now assume $y' < y_{i,j}$. Since, by construction, y' is non-wasteful (with respect to $\{\pi_1, \pi_i\}$, and $x_{i,j}$), and $f(x_{i,j}, y_{i,j}) = 0$ (because we have $[x_{i,j}, y_{i,j}] = \varphi_{\Pi_i}(j)$) we conclude that $f(x_{i,j}, y') = 0$, using Lemma 4.2 of Uno and Yagiura's paper [16]. This yields $[x_{i,j}, y'] \in C_{\Pi_i}$. Since $\varphi_{\Pi_{i-1}}(j) \subseteq c(j, x_{i,j})$ we get a contradiction to the fact that $\varphi_{\Pi_i}(j)$ is the smallest common interval including $\varphi_{\Pi_{i-1}}(j) \in C_{\Pi_i}$. Thus $y' = y_{i,j}$ and $c(j, x_{i,j}) = [x', y'] = [x_{i,j}, y_{i,j}] = \varphi_{\Pi_i}(j)$. □

Definition 6 Assume $\varphi_{\Pi_i}(j) = [x_{i,j}, y_{i,j}]$. For $x \geq x_{i,j}$, the interval $c(j,x)$ is in $C_{\Pi_{i-1}}$ and therefore has a subchain representation $p_{c(j,x)} = (c_{l(j,x)}, \ldots, c_{m(j,x)})$. We define $crb(j,x) := c_{m(j,x)} \in I_{\Pi_{i-1}}$, and call it the *right border* of $c(j,x)$.

We note that due to the nesting structure of maximal chains we have $\varphi_{\Pi_{i-1}}(j) \subseteq c_k$, with $k \in \{l(j,x), \ldots, m(j,x)\}$, and that due to the minimality of $c(j,x)$ the intervals $c_k, \ldots, c_{m(j,x)-1}$ are wasteful, while $c_{m(j,x)}$ is non-wasteful. Due to the nesting structure of chains, and our choice of x we get the following lemma.

Lemma 7 *Assume* $\varphi_{\Pi_{i-1}}(j) = [x_{i-1,j}, y_{i-1,j}]$.

(a) *For* $x \geq y_{i-1,j}$ *we have* $c(j,x) = crb(j,x)$.
(b) *Assume* $y_{i-1,j} > x \geq x_{i,j}$, *and denote* $c = [x,y]$ *a non-wasteful common interval of* $C_{\Pi_{i-1}}$. *We have*

$$crb(j,x) \subseteq c \quad \Leftrightarrow \quad \varphi_{\Pi_{i-1}}(j) \subseteq c.$$

Proof (a) By definition, a right interval end y can only be wasteful with respect to $\{\pi_1, \pi_i\}$, and x if $y > x$. Therefore, for $y_{i-1,j} \leq x$, we get $\varphi_{\Pi_{i-1}}(j) \in SC(j,x)$. We conclude $c(j,x) = \varphi_{\Pi_{i-1}}(j)$. The corresponding subchain representation $p_{c(j,x)} = (\varphi_{\Pi_{i-1}}(j))$ is trivial, and we have $c(j,x) = crb(j,x)$. (b) If $\varphi_{\Pi_{i-1}}(j) \subseteq c$ then we have $c(j,x) \subseteq c$ by Lemma 6, and hence $crb(j,x) \subseteq c$. The reverse follows from the nesting structure of maximal chains. Any common interval that contains $crb(j,x)$ either includes the maximal chain of $crb(j,x)$ completely, and therefore also $c(j,x)$, or is a subchain of $crb(j,x)$'s maximal chain. Let's assume c is a subchain of $crb(j,x)$'s maximal chain which includes $crb(j,x)$. Denote c_k the first element of this subchain with $y_{i-1,j} \in c_k$. Due to our assumption $x < y_{i-1,j}$, and the fact that the right end of $crb(j,x)$ is larger than or equal to $y_{i-1,j}$, the interval c_k is well-defined. Due to the nesting structure of maximal chains we conclude $\varphi_{\Pi_{i-1}}(j) \subseteq c_k \subseteq c$. □

5.2 The Extended Data Structure

We define a new data structure S, in addition to Y (which contains the *ylist*). The data structure S contains the intervals from $I_{\Pi_{i-1}}$ in a particular order.

By Lemma 4(a), $I_{\Pi_{i-1}}$ is partitioned into maximal chains. In the data structure S we have for each maximal chain a doubly-linked list containing as elements the intervals of the chain. These lists are referred to as *clists*. The intervals in each list are sorted in the increasing order of their right ends. By Lemma 4(b), any two chains are either disjoint or one chain is completely contained in exactly one element or the intersection of two consecutive elements of the other chain. Consider now all maximal chains in $I_{\Pi_{i-1}}$ which contain an interval with a particular left end x. Since each chain contains x, they are not disjoint and thus one is contained in one, or the intersection of two consecutive elements of the other. Therefore, these chains are nested and can be ordered hierarchically by nesting depth, starting with the highest nesting depth. In particular, intervals from different *clists* with the same left end x are nested. In S, they are connected by *vertical pointers*, yielding for each index $x \in N$ a doubly-linked *vertical left end list*. The intervals in each vertical list are ordered by increasing length (decreasing nesting level). Analogously, we add *vertical right end lists*. Finally we make a connection to the *ylist* Y, by adding to each *ylist* item a pointer to its corresponding vertical right end list.

In our algorithm (see Section 5.3) unsatisfied labels keep track for which $\varphi_{\Pi_{i-1}}(j) \in I_{\Pi_{i-1}}$ the corresponding $\varphi_{\Pi_i}(j) \in I_{\Pi_i}$ has not been found yet—such intervals are called *unsatisfied*. Originally, all intervals in $I_{\Pi_{i-1}}$ are unsatisfied, and we mark the *clist* item v_j with an unsatisfied label u_j that corresponds to $\varphi_{\Pi_{i-1}}(j)$. During the execution of our algorithm the unsatisfied labels might be shifted to other *clist* items. If we report $\varphi_{\Pi_i}(j) \in I_{\Pi_i}$ the corresponding unsatisfied label u_j is deleted and $\varphi_{\Pi_{i-1}}(j) \in I_{\Pi_{i-1}}$ becomes *satisfied*.

If a *clist* carries an unsatisfied label it is called *labeled*, otherwise *unlabeled*. Thus, originally, all *clist* items are labeled. The labeled elements of a *clist* are connected by a doubly-linked *labeled sublist*, a sublist of the original *clist* in which the left to right order is preserved.

Initially, all *ylist* elements are non-wasteful, but during the execution of our algorithm some values y might be discovered wasteful. In this case, each interval with

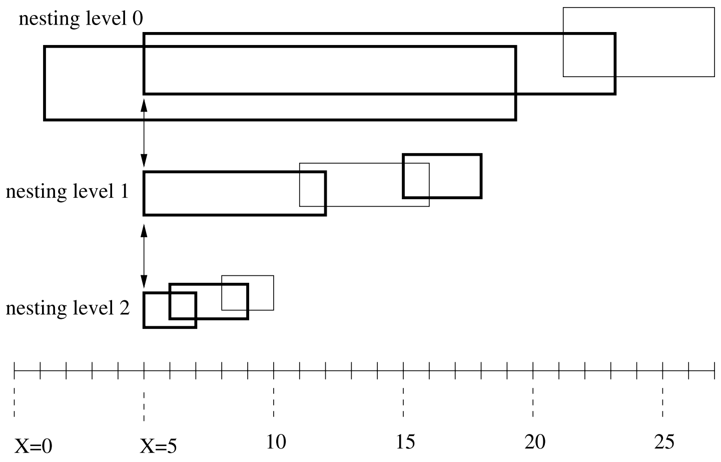

Fig. 5 Visualization of the data structure S. The labeled sublist is in bold face. The only vertical list depicted here is the vertical left end list for $x = 5$. Note that chains at nesting levels 1 and 2 are contained in the intersection of two intervals of nesting level 0

right interval end y becomes wasteful and is removed from its labeled sublist and its *clist*. Removing an interval from the labeled sublist and the *clist* just means adjusting the pointers. The list item is not deleted, because the interval might still be contained in a vertical list. If the interval is labeled, its labels are shifted to a non-wasteful interval (see below).

Figure 5 illustrates the structure of S.

5.3 Description of the Algorithm

In this section we will describe the algorithm which operates in the data structure S. The correctness of the algorithm will be shown in Section 5.4, and the time and space complexities of an efficient implementation of the algorithm are established in Section 5.5.

Algorithm 5 (Extended Algorithm RC)

```
Input: Two permutations $\pi_1 = id_n$ and $\pi_i$ of $N = \{1, \dots, n\}$;
    a set of irreducible intervals $I_{\Pi_{i-1}}$.
Output: The set of irreducible intervals $I_{\Pi_i}$.
1: initialize $Y$ and $S$
2: for $x = n - 1, \dots, 1$ do
3:    update $Y$ and $S$
4:    while $(([x, y], [x', y]) \leftarrow S.next\_candidate(x))$ exists and $f(x, y) = 0$ do
5:       report $[x, y]$
6:       remove $[x', y]$ from its labeled sublist
7:    end while
8: end for
```

Pseudocode is given in Algorithm 5. In addition to the notation used in Algorithms 1 and 2, in this section we will also denote by $v.start$ the left end and by $v.end$ the right end of an interval v.

The algorithm starts by initializing the data structures Y and S (line 1 of Algorithm 5). Y is initialized as in the original Algorithm RC. To initialize S, we partition $I_{\Pi_{i-1}}$ into maximal chains of non-trivially overlapping irreducible intervals as in line 1 of Algorithm 4. The details of this algorithm were explained in Section 4. We add for each chain $p = (c_1, \ldots, c_k)$ a corresponding $clist = (v_1, \ldots, v_k)$ to S, as described in the previous section.

Now, a counter x for the left end of the probed candidate intervals is decreased iteratively from $n - 1$ down to 1 (line 2). Each iteration starts with an update of data structures Y and S (line 3). First, we remove any *clist* with $clist.head.start = x + 1$ from S. Second, we update Y in the same way as in Algorithm RC, but each time we remove an element y from *ylist* we traverse the vertical right end list of y and mark any *clist* item v with $v.end = y$ wasteful, and remove it from its *clist* in S.

If during these updates some interval $v = [x_1, y_1]$, labeled with u_j, is removed, we also remove v from its labeled sublist, and shift the label u_j to the non-wasteful interval $crb(j, x)$. If $crb(j, x)$ was unlabeled at that point then this operation re-adds it to the labeled sublist. Section 5.4 shows that during the execution of our algorithm the label u_j is linked to $crb(j, x)$ until the irreducible interval $\varphi_{\Pi_i}(j) \in I_{\Pi_i}$ is found and reported, afterwards the label is deleted. The interval $crb(j, x)$ is defined as the right border of the non-wasteful interval $c(j, x)$, and it could be computed using this definition. However, to speed-up our algorithm, we describe in Section 5.5 how the algorithm can be modified to avoid this time consuming computation.

After removing interval $v = [x_1, y_1]$ we also update its left end vertical list. In order to implement the function $S.next\_candidate(x)$ which generates candidates for irreducible common intervals efficiently (see below), we need to treat the following cases separately:

- If $x_1 > x$ (i.e., $x \notin v$) then we do not have to update the corresponding vertical left end list at all. The interval $v = [x_1, y_1]$ can never be the first interval in a *clist* with left end x, because $x_1 > x$.
- Due to Definition 1 the wasteful interval $v = [x_1, y_1]$ always satisfies $y_1 > x$, but it is possible that $x_1 \leq x$ (i.e., it is possible that $x \in v$). In this case, the item v could be the first interval in a *clist* with left end x_1 which is accessed later when x has been decremented to x_1. Therefore, after shifting the label of v (if one exists), we search for the next labeled interval α in the *clist* of v. If α exists then we set a pointer from v to α and keep v in the vertical sublist for x_1, otherwise we remove v from the vertical list.

  Due to Lemma 3, there are at most two such intervals (e.g. v and v') which contain x, and have a wasteful right interval end in every *clist*. Therefore, in addition to its *clist* and labeled sublist pointers, any labeled interval α needs to maintain at most two additional 2-way pointers pointing to wasteful intervals like v and v' with left ends smaller than or equal to x. However, during the execution of the algorithm, x is decreased, and v and v' as well as the corresponding pointers might be replaced by new ones. The situation described above is illustrated in Fig. 6. The 2-way pointers are needed for subsequent updates where α might be discovered as

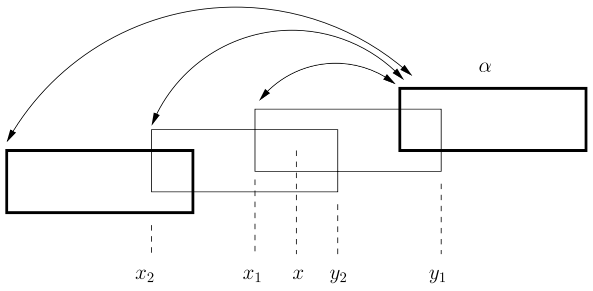

Fig. 6 Visualization of a clist in S which contains wasteful intervals $[x_1, y_1]$ and $[x_2, y_2]$. The labeled sublist is in bold face

wasteful and removed from its *clist*. We have to distinguish two cases. First, if the unsatisfied labels attached to α are shifted to another item α' in the same *clist*, then the 2-way pointers of α are also shifted to α'. If there are already 2-way pointers pointing to α' then the obsolete pointers and their corresponding items are deleted. Second, if the labels attached to α are moved to an item in a different *clist*, i.e. α is the last element in its *clist*, then we use the 2-way pointers to remove v and v' from their vertical left end lists, since the left ends of v and v' cannot be left ends of common intervals anymore.

After these updates, the function $S.next\_candidate(x)$ generates candidate intervals $[x, y]$ for irreducible common intervals in I_{Π_i} (this is checked in the following step). The function $S.next\_candidate(x)$ processes the *clists* which include an item v with $v.start = x$ in decreasing order of nesting level using the corresponding vertical left end list. Each time this function is called, it returns the smallest (w.r.t. the interval order; see Definition 4) remaining non-wasteful common interval $[x, y]$ of $C_{\Pi_{i-1}}$ which has a left end x, and contains some labeled interval $[x', y]$ (of $I_{\Pi_{i-1}}$) with $x' \geq x$. The interval $[x, y]$ is a candidate for a common interval (to be tested by checking if $f(x, y) = 0$; see line 4 in Algorithm 5). In addition, the function $S.next\_candidate(x)$ also returns the interval $[x', y]$ as a second part of its return value, because one might later need to remove this interval from the labeled sublist (see line 6 in Algorithm 5). In the following, we will focus only on the interval $[x, y]$, and call it the return value of $S.next\_candidate(x)$.

More formally, the function $S.next\_candidate(x)$ is defined on the labeled sublists of the data structure S as follows. $S.next\_candidate(x)$ returns an interval $[x, y]$ which is uniquely defined by the following criteria.

Primary criterion: $[x, y]$ contains a labeled interval $[x', y]$ from some chain in S which contains an element with left end x, and $x' \geq x$.
Secondary criterion: This labeled interval $[x', y]$ has maximal nesting depth.
Tertiary criterion: This labeled interval $[x', y]$ has minimal left end x'.

Note that during the execution of our algorithm for decreasing values x the sequence of candidate intervals $[x, y]$ obtained by successive calls to the function $S.next\_candidate(x)$ is strictly increasing w.r.t. the interval order from Definition 4. The function $S.next\_candidate(x)$ can be implemented in such a way that it uses only *constant time*. When the function $S.next\_candidate(x)$ is called for the first time with a particular parameter x, it accesses the vertical left end list for x which takes it to the *clist* with highest nesting depth which contained at initialization an interval with left end x, and which still contains a labeled interval with left end larger than or equal to x. The *clists* which do not contain any labeled interval with left end smaller than or equal to x are skipped in the vertical list (see above). Since all previous calls to the S*.next_candidate* function were with a parameter $x' > x$, the first interval v in the *clist* could not have become satisfied yet, because all intervals are initially unsatisfied and this interval with left end x has never been accessed before. However, v might have been labeled as wasteful earlier, and in this case we follow its pointer which takes us directly to the first labeled interval α in this *clist*. This takes only constant time.

Further calls of S*.next_candidate*(x) result in the traversal of the labeled sublist of the *clist* with the highest nesting depth and thus each call takes constant time. When the end of the labeled sublist of a *clist* is reached then the function accesses the labeled sublist of the *clist* with the next lower nesting depth which still contains labeled intervals by following the vertical left-end list for x. It then traverses the labeled elements in this *clist* as described above. Thus the function S*.next_candidate*(x) does not have to search for the first unsatisfied interval, but every call to S*.next_candidate*(x) takes constant time.

Our algorithm probes each candidate interval $c = [x, y]$ obtained from the function S*.next_candidate*(x). As in the original algorithm RC, we evaluate function $f(x, y)$ in order to decide if c is a common interval of permutations π_1 and π_i. If $f(x, y) > 0$ we conclude $c \notin I_{\Pi_i}$, and that no further irreducible interval with left interval end x exists. We decrement x and continue with the next iteration.

If $f(x, y) = 0$, and $v = [x', y]$ is labeled by u_j, then we conclude $c = \varphi_{\Pi_i}(j) \in I_{\Pi_i}$ and report c. The interval $\varphi_{\Pi_{i-1}}(j) \in I_{\Pi_{i-1}}$ becomes satisfied, we delete all unsatisfied labels from v, and the interval v is removed from its labeled sublist. This can only happen to intervals $v = [x'', y]$ with $x'' \geq x$ (see Section 5.4). Finally, the function S*.next_candidate*(x) continues its traversal.

The example in Fig. 5 shows a data structure S with some gaps in the labeled sublists of the *clists*. This is possible, because intervals with left end $\tilde{x} > x$ have been accessed earlier and some have become satisfied or wasteful.

In the example in Fig. 5, successive calls of the function S*.next_candidate*(5) would yield the increasing sequence of intervals $[5, 7]$, $[5, 9]$, $[5, 12]$, $[5, 18]$ and $[5, 23]$. If one considers the full combined return values of the form $([x, y], [x', y])$ containing also the interval $[x', y]$, then the results are $([5, 7], [5, 7])$, $([5, 9], [6, 9])$, $([5, 12], [5, 12])$, $([5, 18], [15, 18])$ and $([5, 23], [5, 23])$.

5.4 Correctness of the Algorithm

The correctness of our algorithm relies on the following Theorem.

Theorem 2 *Assume* $\varphi_{\Pi_i}(j) = [x_{i,j}, y_{i,j}]$. *For any* $x \geq x_{i,j}$ *the unsatisfied label* u_j *and the clist item* $crb(j, x)$ *exist, and label* u_j *is linked to* $crb(j, x)$ *at the start of line* 4 *in Algorithm* 5. *In iteration* $x = x_{i,j}$, *the function* S*.next_candidate*$(x_{i,j})$ *will report interval* $c(j, x_{i,j}) = \varphi_{\Pi_i}(j)$, *and the label* u_j *will be deleted. For* $x < x_{i,j}$ *the label* u_j *is deleted.*

Proof By Lemma 7(a), we have $crb(j,x) = \varphi_{\Pi_{i-1}}(j)$ for $x \geq y_{i-1,j}$. We use induction to prove the theorem. First we note that during the initialization of data structure S every *clist* item $\varphi_{\Pi_{i-1}}(j)$ is labeled with u_j, and that for $x = n-1$ no item is removed during the updates in line 3. Hence the theorem is true for $x = n-1$.

To prove correctness in the general case we assume that the theorem is true for $x = t+1, \dots, n-1$, and show its correctness for $x = t$. We distinguish three cases for t.

Case 1, $t \geq y_{i-1,j}$. By assumption, the label u_j exists and is linked to $\varphi_{\Pi_{i-1}}(j) = [x_{i-1,j}, y_{i-1,j}]$ in line 4 of iteration $x = t+1$. Since for any $x = t > y_{i-1,j} - 1$ our algorithm does not affect the *clist* item $[x_{i-1,j}, y_{i-1,j}]$, we conclude that u_j still exists and is still linked to $\varphi_{\Pi_{i-1}}(j)$ in iteration $x = t$.

Case 2, $x_{i,j} \leq t < y_{i-1,j}$. By assumption, the label u_j exists and is linked to $crb(j, t+1) = [x_*, y_*]$ at the start of line 4 in iteration $x = t+1$. First we show that at the start of line 4 in iteration $x = t$ the label u_j still exists, and then we will show that at this time the label u_j is linked to $crb(j,t)$.

To show that the label u_j still exists at the start of iteration $x = t$ we remark that our algorithm deletes labels only if common intervals of C_{Π_i} are reported in lines 4–7 of iteration $x = t+1$. Therefore, the label u_j will be deleted iff $[t+1, y_*]$ is a common interval of C_{Π_i}. In this case, by Lemma 6(a), we get $\varphi_{\Pi_{i-1}}(j) \subseteq [t+1, y_*]$. Using Theorem 1(a.ii) and (c) we conclude $[x_{i,j}, y_{i,j}] = \varphi_{\Pi_i}(j) \subseteq [t+1, y_*]$, and thus $x_{i,j} \geq t+1$. This contradicts our assumption $t \geq x_{i,j}$. Hence we may assume that at the beginning of iteration t the label u_j still exists, and that it is linked to $crb(j, t+1)$.

Now we argue that at the start of line 4 in iteration $x = t$ the label u_j is linked to $crb(j,t)$. Whenever item $crb(j, t+1)$ is deleted in the update of S in line 3 of iteration $x = t$, our algorithm will move u_j to $crb(j,t)$, where the interval $crb(j,t)$ can be computed using Definition 6.

Therefore, we will now focus on the case that $crb(j, t+1)$ is not deleted, and show that in this case $crb(j, t+1) = crb(j,t)$. We distinguish two sub-cases: 1. If $c(j, t+1) = c(j,t)$ then we also have $crb(j, t+1) = crb(j,t)$. 2. If $c(j, t+1) \neq c(j,t)$ we conclude $c(j, t+1) = [t+1, y_*]$. The interval $[t+1, y_*]$ corresponds to a subchain $p = (i_l, \dots, i_*)$. Due to the nesting structure of maximal chains we conclude that $c(j,t)$ corresponds to $p' = (i_{l-1}, i_l, \dots, i_*)$, hence $crb(j, t+1) = crb(j,t)$. Note that the existence of item i_{l-1} is guaranteed by the fact that the chain corresponding to p is not deleted in the transition from $x = t+1$ to $x = t$.

Case 3, $t < x_{i,j}$. We argue that in iteration $t = x_{i,j}$ the label u_j is deleted in lines 4–7 of Algorithm 5 and hence does not exist anymore for $t < x_{i,j}$. As seen above, at the start of line 4 in iteration $t = x_{i,j}$, the label u_j is linked to $crb(j, x_{i,j})$. Using Lemma 6(d) we get $c(j, x_{i,j}) = [x_{i,j}, y_{i,j}] = \varphi_{\Pi_i}(j)$. Since $\varphi_{\Pi_i}(j) \in I_{\Pi_i}$, and hence also $\varphi_{\Pi_i}(j) \in C_{\{\pi_1, \pi_i\}}$ we have $f(x_{i,j}, y_{i,j}) = 0$. Using the monotonicity of f we conclude $f(x_{i,j}, y) = 0$ for all non-wasteful $y \leq y_{i,j}$. Therefore, in iteration $t = x_{i,j}$, while traversing the *clist* which corresponds to $c(j, x_{i,j})$, the function $S.next\_candidate(t)$ will probe and report interval $c(j, x_{i,j})$, and the label u_j will be deleted. □

To prove the correctness of our algorithm, we show the following three conditions:

1. No interval is reported twice.
2. Only irreducible intervals are reported.
3. Every irreducible interval is reported.

No interval is reported twice. We note that $S.next\_candidate()$ probes candidate intervals in strictly increasing interval order, hence no interval will be tested or reported twice.

Only irreducible intervals are reported. The algorithm reports the interval $c = [x, y]$ only if c is a subchain $(c_l, \dots, c_m)$ of a maximal chain in $I_{\Pi_{i-1}}$, and c_m is labeled. Assume c_m is labeled by u_j. The algorithm will delete u_j after reporting c, hence, using Theorem 2 we conclude $c_m = crb(j, x)$, $x = x_{i,j}$, and $c = c(j, x) = \varphi_{\Pi_i}(j)$. This implies that c is irreducible.

Every irreducible interval is reported. This is a direct consequence of Theorem 1(b.ii) and Theorem 2.

5.5 Complexity Analysis

In the following we will describe an $O(kn)$ time and $O(n)$ space implementation of a slight modification of the above algorithm. The modification omits certain details of the original algorithm used only to simplify the correctness proof. This improves its efficiency without changing the output.

Our implementation differs from the original algorithm only in how unsatisfied intervals are handled. We keep track if a *clist* item is labeled using a binary flag, but we do not maintain individual labels for each unsatisfied interval of $I_{\Pi_{i-1}}$. Below we describe how these binary flags are updated during the execution of our algorithm. Since our original algorithm only uses the information that there is an unsatisfied interval of $I_{\Pi_{i-1}}$ contained in a candidate interval, this modification does not affect the correctness of the algorithm, but its running time is improved.

The initialization of Y in line 1 is done as in the original Algorithm RC and takes overall $O(n)$ time, see [16]. To initialize S, we first partition $I_{\Pi_{i-1}}$ in $O(n)$ time into maximal chains, see Section 4. During initialization, all intervals in S are *unsatisfied*. Thus we mark the corresponding *clist* items unsatisfied, and add *labeled sublist* pointers between all consecutive *clist* elements. In contrast to the above description, the unsatisfied label is now only a flag, and does not include the information of the corresponding underlying interval of $I_{\Pi_{i-1}}$. We also keep track of interval ends, and add the corresponding *vertical interval end lists*. Since there are at most $O(n)$ intervals and unsatisfied labels, and each interval appears in exactly one labeled sublist, one vertical left end list, and one vertical right end list, the entire initialization takes $O(n)$ time.

As in the original Algorithm RC [16], in the extended Algorithm RC the time spent in updating Y in line 3 is proportional to the number of deleted items in *ulist*, *llist*, and *ylist*. Hence, it takes overall $O(n)$ time.

The update of S in line 3 is performed as follows. For each index y that is deleted from the *ylist*, all corresponding *clist* items with right end y in S are removed. If a removed item v is labeled, and if its successor in the same *clist* exists, then we label the successor. Subsequently, the labeled sublists are updated correspondingly. However, in contrast to the description of the update of S in Section 5.4, if the *clist* successor does not exist, we do not perform additional updates. Although such cases might occur, e.g. after deleting the tail item of a *clist*, or after deleting an entire *clist*, we argue in the following that an update is not necessary because the corresponding interval is already labeled. This saves us the time for searching the corresponding non-wasteful hull in another *clist*, and guarantees that the label manipulations can be performed in constant time. Assume a *clist* item v marked with the label u_j is deleted. As noted in Section 5.1, the corresponding non-wasteful hull $c(j,x)$ exists and has a subchain representation $p_{c(j,x)} = (c_l, \ldots, c_m)$, where $v \subseteq c_k$, $k \in \{l, \ldots, m\}$ and $c_k, \ldots, c_{m-1}$ are wasteful. Assume c_k corresponds to $\varphi_{\Pi_{i-1}}(k')$. Since $v \subseteq c_k$, label $u_{k'}$ still exists, and is linked to c_m. Using Lemma 6(b) we conclude $c(j,x) \subseteq c(k',x)$. We also have $\varphi_{\Pi_{i-1}}(k') \subseteq c(j,x)$, hence $c(k',x) \subseteq c(j,x)$ by Lemma 6(a). Together this yields $c(j,x) = c(k',x)$. Since $c(j,x-t) = c(c(j,x), x-t)$ we conclude that $c(j,x-t) = c(k',x-t)$ for $t \in \{1, \ldots, x-1\}$. This implies $crb(j,x-t) = crb(k',x-t)$ for $t \in \{1, \ldots, x-1\}$. We conclude that during the execution of our algorithm we only have to make sure that $crb(k',x)$ is updated correctly, i.e. marked with an unsatisfied label. If for $t \in \{1, \ldots, x-1\}$ the chain that represents $c(k',x)$ has to be extended in order to represent $c(k',x-t)$, the update will be correctly performed by our algorithm. If however $c(k',x-t)$ is represented by another chain, we use the same argument as above for a different k'_2. This concludes our argument.

Since each of the $O(n)$ *clist* items is discovered as wasteful at most once during the updates, and since each such event causes one removal from the *clist*, at most one removal and one insertion in the labeled sublist, a constant number of removals from vertical left end lists, and a constant number of pointer updates, and since each of these operations can be performed in constant time, this part takes overall $O(n)$ time.

For every x, the function $S.next\_candidate(x)$ generates the first candidate interval $[x,y]$ and its potential successors in constant time, as described in Section 5.3.

For every candidate interval, the evaluation of $f(x,y)$, as well as the corresponding update operations for $f(x,y) = 0$, are constant time operations. Therefore, as in Algorithm RC, the time required for reporting the output is proportional to the size of the output, here $|I_{\Pi_i}| < n$.

Putting things together, Algorithm 5 takes $O(n)$ time and space. Since at any point of Algorithm 5 we need to store only two permutations π_1 and π_i, the data structures Y and S, and the current I_{Π_i}, we have:

Theorem 3 *The irreducible intervals of k permutations of n elements can be found in optimal $O(kn)$ time and $O(n)$ additional space.*

Combining these results with the result of Section 4 we get:

Theorem 4 *The z common intervals of k permutations of n elements can be found in optimal $O(kn+z)$ time and $O(n)$ additional space.*

6 Common Intervals of Multichromosomal Permutations

We define a *chromosome* γ of $N := \{1, 2, \ldots, n\}$ as a linearly ordered subset of N, and represent it as a linear list. A *multichromosomal permutation* π of N is defined as a set of chromosomes, containing each element of N exactly once, i.e.

$$\pi = \{\gamma_1, \ldots, \gamma_l\} \quad \text{with } N = \dot{\bigcup}_{1 \le i \le l} \gamma_i .$$

Given a family $\Pi = (\pi_1, \ldots, \pi_k)$ of k multichromosomal permutations of N, a subset $c \subseteq N$ of cardinality $|c| \ge 2$ is called a *common interval* of Π if and only if for each multichromosomal permutation π_i, $i = 1, \ldots, k$, there exists a chromosome with c as an interval. *Reducible* and *irreducible intervals* are defined as in Definition 3.

Example 2 Let $N = \{1, \ldots, 6\}$ and $\Pi = (\pi_1, \pi_2, \pi_3)$ with $\pi_1 = \{(1, 2, 3), (4, 5, 6)\}$, $\pi_2 = \{(1, 5, 6, 4), (3, 2)\}$, and $\pi_3 = \{(1, 6, 4, 5), (3), (2)\}$. Here chromosome ends are indicated by parentheses. The only common interval is $\{4, 5, 6\}$. □

A modification of the algorithms from the previous section can be used for finding all common intervals of k multichromosomal permutations. We start by concatenating the chromosomes of each multichromosomal permutation in arbitrary order. This way we obtain a family $\Pi' = (\pi'_1, \pi'_2, \ldots, \pi'_k)$ of k (standard) permutations π'_i, $i = 1, \ldots, k$. Without loss of generality we assume that $\pi'_1 = id_n$. Now, as above, set $\Pi'_i := (\pi'_1, \pi'_2, \ldots, \pi'_i)$. Starting with

$$I_{\Pi'_1} := \{[j, j+1] \mid 1 \le j < n \text{ and } j, j+1 \text{ on the same chromosome in } \pi_1\},$$

we successively compute $I_{\Pi'_i}$ from $I_{\Pi'_{i-1}}$ for $i = 2, \ldots, k$ using a modification of the algorithm described in Section 5, where we suppress reporting irreducible intervals if the elements at indices x and y belong to different chromosomes of π_i (line 5 of Algorithm 5), but continue the execution of the algorithm without a change otherwise.

By the definition of $I_{\Pi'_1}$, this algorithm will never place two elements from different chromosomes in π_1 together in an irreducible interval. Moreover, by the modification of Algorithm 5, no irreducible interval containing two elements from different chromosomes of the other permutations $\pi_2, \ldots, \pi_k$ will be reported. Nevertheless, the location of irreducible intervals that lie on the same chromosome in all permutations is not affected by the modification of the algorithm. Since the additional test if x and y belong to the same chromosome is a constant-time operation, and the output can not be larger than that of the original Algorithm 5, the new algorithm also takes $O(n)$ time to generate $I_{\Pi'_i}$ from $I_{\Pi'_{i-1}}$. The final generation of the common intervals from the irreducible intervals (Algorithm 4) is unchanged, so that we have the following:

Theorem 5 *Given k multichromosomal permutations of $N = \{1, \ldots, n\}$, all z common intervals can be found in optimal $O(kn + z)$ time using $O(n)$ additional space.* □

7 Common Intervals of Circular Permutations

In this section we consider arrangements of $N = \{1, 2, \dots, n\}$ along a circle and call this a *circular permutation*. Common intervals of circular permutations are of special interest in biological applications, such as genome comparisons.

Given a family $\Pi = (\pi_1, \dots, \pi_k)$ of k circular permutations of N, a subset $c \subseteq N$ is called a *common interval* if and only if the elements of c occur uninterruptedly in each circular permutation. Note that here we do not exclude trivial intervals consisting of only a single element, or the empty set, in order to simplify the notation of the following Lemma 8.

Example 3 Let $N = \{1, \dots, 6\}$ and $\Pi = (\pi_1, \pi_2, \pi_3)$ be a family of circular permutations with $\pi_1 = (1, 2, 3, 4, 5, 6)$, $\pi_2 = (2, 4, 3, 5, 1, 6)$, and $\pi_3 = (6, 1, 5, 4, 3, 2)$. The common intervals of Π are $\{\}$, $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$, $\{5\}$, $\{6\}$, $\{1, 6\}$, $\{3, 4\}$, $\{2, 3, 4\}$, $\{3, 4, 5\}$, $\{5, 6, 1\}$, $\{6, 1, 2\}$, $\{2, 3, 4, 5\}$, $\{5, 6, 1, 2\}$, $\{1, 2, 3, 4, 5\}$, $\{2, 3, 4, 5, 6\}$, $\{3, 4, 5, 6, 1\}$, $\{4, 5, 6, 1, 2\}$, $\{5, 6, 1, 2, 3\}$, $\{6, 1, 2, 3, 4\}$, and $\{1, 2, 3, 4, 5, 6\}$. □

In the following we will show how to find all z common intervals in a family of circular permutations in optimal $O(kn + z)$ time. This can be done by a modification of the original algorithm from Section 5, in combination with the following observation.

Lemma 8 *Let c be a common interval of a family Π of circular permutations of N. Then its complement $\bar{c} := N \setminus c$ is also a common interval of Π.*

Proof This follows immediately from the definition of common intervals of circular permutations. □

Note that Lemma 8 does not hold for irreducible intervals. For instance, in Example 3, $\{3, 4\}$ is an irreducible interval, while its complement $\{5, 6, 1, 2\}$ is not, as it is generated by the shorter intervals $\{5, 6, 1\}$ and $\{6, 1, 2\}$.

The general idea is now to find the common intervals of size $\le \lfloor \frac{n}{2} \rfloor$, and then to form their complements. The procedure is outlined in Algorithm 6. The main difference from the algorithm described in Section 5 is that function φ_i is replaced by a variant, denoted φ_i^*, that works on circular permutations and only generates irreducible intervals of size $\le \lfloor \frac{n}{2} \rfloor$. This function is implemented by multiple calls to the original function φ_i. The two circular permutations π_1 and π_i are linearized in two different ways each, namely by once cutting them between positions n and 1, and once cutting between positions $\lfloor \frac{n}{2} \rfloor$ and $\lfloor \frac{n}{2} \rfloor + 1$. Then φ_i is applied to each of the four resulting pairs of linearized permutations. For convenience, the output of irreducible intervals of length $> \lfloor \frac{n}{2} \rfloor$ is suppressed. Note that no irreducible interval will become reducible due to the above linearization, and that every irreducible interval of size $\le \lfloor \frac{n}{2} \rfloor$ will be reported at least once. The resulting sets of intervals of the four runs of φ_i are merged, sorted according to their start and end positions using bucket sort, and duplicates are removed. This procedure generates $I^*_{\Pi_i}$, the set of all irreducible intervals of Π_i of size $\le \lfloor \frac{n}{2} \rfloor$, in $O(n)$ time.

We order the resulting irreducible intervals with respect to the two linearizations of π_1. Subsequently, we apply Algorithm 4, where the output of common intervals of

Algorithm 6 (Finding all Common Intervals of k Circular Permutations)

```
Input: A family $\Pi = (\pi_1 = id_n, \pi_2, \dots, \pi_k)$ of $k$ circular permutations of
    $N = \{1, \dots, n\}$.
Output: The set of all common intervals $C_\Pi$.
1: $I^*_{\Pi_1} \leftarrow \{\{1,2\},\{2,3\},\dots,\{n-1,n\},\{n,1\}\}$
2: for $i = 2, \dots, k$ do
3:     $I^*_{\Pi_i} \leftarrow \{\varphi^*_i(c) \mid c \in I^*_{\Pi_{i-1}}\}$
4: end for
5: generate $C^*_\Pi$ from $I^*_\Pi = I^*_{\Pi_k}$ using Algorithm 4    // (suppress intervals of
   size $> \lfloor \frac{n}{2} \rfloor$)
6: $\overline{C}^*_\Pi \leftarrow \{\bar{c} \mid c \in C^*_\Pi\}$
7: output $C^*_\Pi \cup \overline{C}^*_\Pi$
```

size $> \lfloor \frac{n}{2} \rfloor$ is suppressed. Merging the resulting intervals, removing duplicates, and adding the trivial intervals gives the set $C^*_{\Pi_i}$ of all common intervals of Π_i of size $\leq \lfloor \frac{n}{2} \rfloor$. Finally, the set of interval complements $\overline{C}^*_\Pi \leftarrow \{\bar{c} \mid c \in C^*_\Pi\}$ is added. This procedure generates all z common intervals in $O(n+z)$ time. Hence, we have the following:

Theorem 6 *Given k circular permutations of $N = \{1, \dots, n\}$, all z common intervals can be found in optimal $O(kn + z)$ time using $O(n)$ additional space.* □

8 Common Intervals of Mixed Permutations

A *mixed permutation* of $N = \{1, \dots, n\}$ is a multichromosomal permutation, where individual chromosomes might be linear or circular. The definitions of common and irreducible intervals are carried over from the previous sections. In the following, we adapt our algorithms to mixed permutations without losing the optimal running time. Difficulties arise because circular chromosomes of different permutations might not contain the same set of elements, and Lemma 8 no longer holds as the following example shows.

Example 4 Let $N = \{1, \dots, 8\}$ and $\Pi = (\pi_1, \pi_2)$ with $\pi_1 = \{(1,2,3,4), (5,6,7,8)\}$ and $\pi_2 = \{(1,3,5,6,7), (2,4,8)\}$ where all chromosomes are circular. While $c = \{5,6\}$ is a common interval, its complement $N \setminus c = \{1,2,3,4,7,8\}$ is not. □

We overcome this problem by a preprocessing step where we include artificial *breakpoints* into the permutations. The breakpoints do not affect common intervals but refine the permutations so that they can be handled by our algorithms. The first time a breakpoint is inserted in a circular chromosome, the chromosome is linearized by cutting it at the breakpoint and replacing it in the permutation by the appropriately circularly shifted linear chromosome. Breakpoints in a linear chromosome dissect the chromosome. The preprocessing is performed as follows.

For a given set $\Pi = (\pi_1, \ldots, \pi_k)$ of mixed permutations we compare permutation π_1 successively to each of the other permutations π_i, $2 \leq i \leq k$, and test for each pair of neighboring elements in π_1 (i.e. for each chromosome $\gamma = (\pi_1(l), \pi_1(l+1), \ldots, \pi_1(r))$ the pairs $\{\pi_1(j), \pi_1(j+1)\}$ for $l \leq j \leq r-1$, plus the pair $\{\pi_1(l), \pi_1(r)\}$ if γ is circular) whether they lie on the same chromosome in π_i or not. If not, they cannot be elements of the same common interval and we introduce a new artificial breakpoint between the two elements in π_1. Then we reverse the comparison, i.e., we introduce breakpoints between neighboring elements of π_i, $2 \leq i \leq k$, whenever they do not lie on the same chromosome of π_1. This preprocessing can be performed in $O(kn)$ time.

After the preprocessing, the elements of each remaining circular chromosome correspond in the other permutations either to exactly one circular chromosome or to one or more linear chromosomes. This allows us to partition N into a set N_l of elements that only occur in linear chromosomes, and into sets $N_{c_1}, \ldots, N_{c_t}$ of elements that occur in at least one circular chromosome.

The elements of N_l can be handled by the algorithm for multichromosomal permutations (Section 6) in a straightforward way. The sets $N_{c_1}, \ldots, N_{c_t}$ are treated separately. We start by restricting all permutations to the selected element set. If each of the restricted permutations is circular, we can apply the algorithm for circular permutations (Section 7) directly. Otherwise, we choose one of the restricted permutations that consists of one or more linear chromosomes, as a start permutation, and arrange these chromosomes in an arbitrary order. Denote by l (r) the first (last) element in this order. We proceed as in the multichromosomal case (Section 6), except if we encounter a circular permutation π_c. If l and r are neighboring elements in π_c, we linearize π_c by cutting between them and proceed as for a linear permutation. Otherwise, similar to the case of circular permutations (Section 7), we copy π_c four times and linearize the copies by cutting one copy on the left of l, one copy on the right of l, one copy on the left of r, and one copy on the right of r. For each of these permutations we compute the irreducible common intervals with the previously processed permutations $\pi_1, \ldots, \pi_{c-1}$. The resulting intervals are merged, sorted according to their start and end positions using bucket sort, and duplicates are removed. This procedure guarantees that we determine all irreducible intervals except for those that contain l and r simultaneously. Due to our choice of l and r there is at most one such interval—the trivial one—which contains all elements. We test this interval separately. Then we continue with permutation π_{c+1} in the same way. Finally, we order the resulting irreducible intervals with respect to the start permutation π_1 and apply Algorithm 4 to recover all common intervals.

Since the above described preprocessing, and the modifications of the algorithms for multichromosomal and circular permutations do not affect the optimal asymptotic running time, we have

Theorem 7 *Given k mixed permutations of $N = \{1, \ldots, n\}$, all z common intervals can be found in optimal $O(kn + z)$ time using $O(n)$ additional space.*

9 Conclusion

In this paper we have presented a time and space optimal algorithm for finding common intervals in a family of permutations, and we have shown how this algorithm can be extended to multichromosomal, circular, and mixed permutations. The algorithm is based on the idea of restricting the computations to the smaller set of irreducible intervals and later recovering the whole solution. The efficiency of the algorithm is derived from the fact that there are always less than n irreducible intervals, which are sufficient to generate the whole set of common intervals with up to $\binom{n}{2}$ elements. Additionally, this compact representation might be of importance for applications in pattern matching and association detection.

In the Appendix we demonstrate the potential advantages of Algorithm 5 over naive approaches. We show that, for fixed k, the expected number of non-trivial common intervals of k permutations generated uniformly at random is $O(1)$, thus the expected runtime in this case is $O(kn)$; in essence the time necessary for reading the input data. In a simulation experiment we verify this linear run-time behavior, and compare it with an approach that checks all intervals of one permutation and, by consequence, shows $O(n^2)$ running time behavior.

Acknowledgements The authors would like to thank the anonymous referees for their extremely helpful suggestions, which contributed greatly to this paper.

Appendix

A.1 Random Inputs

Let $\Pi = (\pi_1, \dots, \pi_k)$ be a family of $k \geq 2$ linear permutations, generated uniformly at random; i.e., every permutation appears with probability $\frac{1}{n!}$. For $l = 2, \dots, n$ and $i = 1, \dots, n-l+1$ we define indicator variables

$$X^k_{l,i} := \begin{cases} 1 & \text{if } \pi_1([i, i+l-1]) \text{ is a common interval of } \Pi, \\ 0 & \text{otherwise.} \end{cases}$$

Let $X^k_l := \sum_{i=1}^{n-l+1} X^k_{l,i}$ be the number of common intervals of size l, and $X^k := \sum_{l=2}^{n} X^k_l$ the total number of common intervals. In the following we will show that for $k \geq 2$ we get the expected value $E(X^k) = O(1)$. More precisely,

$$E(X^k) = \begin{cases} 3 + O(n^{-1}) & \text{if } k = 2, \\ 1 + O(n^{-1}) & \text{if } k > 2. \end{cases}$$

This extends the result of Uno and Yagiura [16] for $k = 2$ permutations. Figure 7 shows the corresponding number of common intervals for different values of n and k, averaged over 1000 randomly generated permutations.

Proof We have

$$E(X^k_{n-1}) = \frac{2^k}{n^{k-1}}, \qquad E(X^k_n) = 1.$$

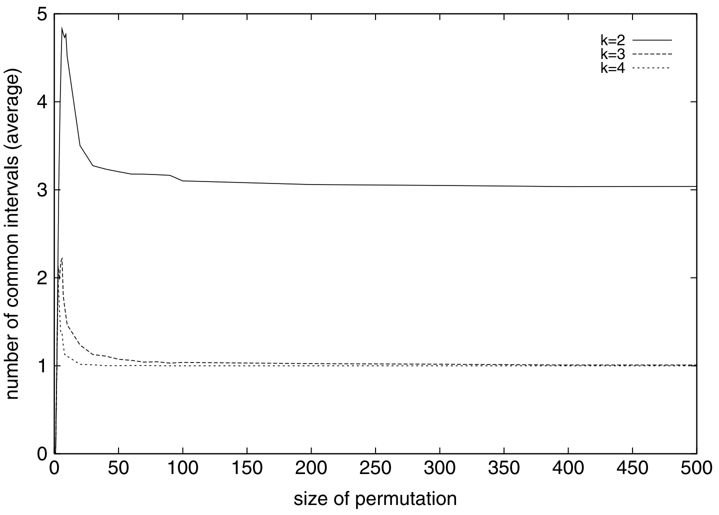

Fig. 7 Empirical measurement of the number of common intervals. After an initial increase, the averaged number of common intervals approaches the asymptotic value

Uno and Yagiura [16] showed that $E(X_2^2) = 2 - \frac{2}{n}$ and $E(\sum_{l=3}^{n-2} X_l^2) = O(n^{-1})$. This yields the above result for $k = 2$.

Now suppose that $k > 2$. Since $E(X_l^k) = [\frac{(n-l)!l!}{n!}(n-l+1)]^{k-2} E(X_l^2)$ and $[\frac{(n-l)!l!}{n!}(n-l+1)]^{k-2} < \frac{C}{n^{k-2}}$ for $l = 2, \dots, n-1$, we get:

$$\begin{aligned} E(X^k) &= E(X_2^k) + E\left(\sum_{l=3}^{n-2} X_l^k\right) + E(X_{n-1}^k) + E(X_n^k) \\ &< \frac{C}{n^{k-2}} E(X_2^2) + \frac{C}{n^{k-2}} E\left(\sum_{l=3}^{n-2} X_l^2\right) + \frac{2^k}{n^{k-1}} + 1 \\ &= 1 + O(n^{-1}). \end{aligned}$$

□

A.2 Empirical Running Times

Here we compare an implementation of our new algorithm with a naive approach that tests for each interval of the first input permutation if it also exists in the remaining $k-1$ permutations. All implementations were done in ANSI C. Time measurements were performed on a Linux laptop with a 750 MHz Intel Pentium III Mobile CPU and 256 MB of main memory. We generated instances of $k = 2, \dots, 10$ permutations with up to $n = 10,000$ elements uniformly at random. Figure 8, left, shows the average running time of the naive algorithm, showing a distinct quadratic increase in running time. In contrast to that, our algorithm (Fig. 8, right) shows a linear running time increase. Each value is an average over 100 independent measurements.

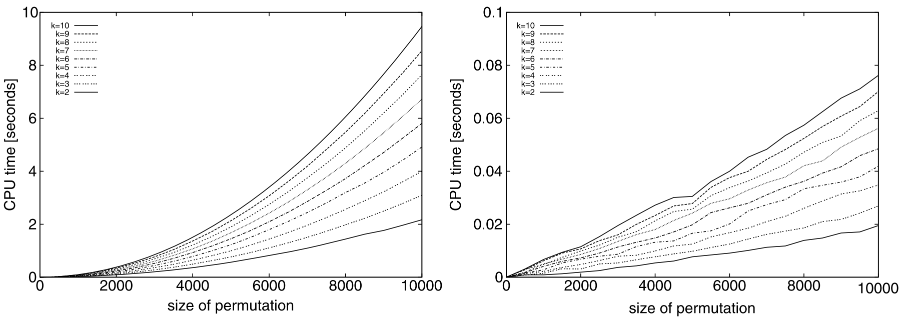

Fig. 8 Average running times of the algorithms. *Left*: naive $O(n^2)$ time algorithm. *Right*: linear time algorithm (Algorithms 5–4). Note the different scalings on the time axis

References

1. Heber, S., Stoye, J.: Algorithms for finding gene clusters. In: Proceedings of the First International Workshop on Algorithms in Bioinformatics (WABI 2001). Lecture Notes in Computer Science, vol. 2149, pp. 252–263. Springer, Berlin (2001)
2. Heber, S., Stoye, J.: Finding all common intervals of k permutations. In: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001). Lecture Notes in Computer Science, vol. 2089, pp. 207–218. Springer, Berlin (2001)
3. Marcotte, E.M., Pellegrini, M., Ng, H.L., Rice, D.W., Yeates, T.O., Eisenberg, D.: Detecting protein function and protein-protein interactions from genome sequences. Science **285**, 751–753 (1999)
4. Overbeek, R., Fonstein, M., D'Souza, M., Pusch, G.D., Maltsev, N.: The use of gene clusters to infer functional coupling. Proc. Natl. Acad. Sci. USA **96**, 2896–2901 (1999)
5. Snel, B., Lehmann, G., Bork, P., Huynen, M.A.: STRING: A web-server to retrieve and display the repeatedly occurring neigbourhood of a gene. Nucleic Acids Res. **28**, 3443–3444 (2000)
6. Bergeron, A., Heber, S., Stoye, J.: Common intervals and sorting by reversals: A marriage of necessity. In: Proceedings of the European Conference on Computational Biology (ECCB 2002) (Supplement of Bioinformatics), vol. 18, pp. 54–63. University Press, Oxford (2002) (Suppl. 2)
7. Bergeron, A., Stoye, J.: On the similarity of sets of permutations and its applications to genome comparison. In: Proceedings of the 9th International Computing and Combinatorics Conference, COCOON 2003. Lecture Notes in Computer Science, vol. 2697, pp. 68–79. Springer, Berlin (2003)
8. Brady, R.M.: Optimization strategies gleaned from biological evolution. Nature **317**, 804–806 (1985)
9. Kobayashi, S., Ono, I., Yamamura, M.: An efficient genetic algorithm for job shop scheduling problems. In: Proceedings of the 6th International Conference on Genetic Algorithms, pp. 506–511. Morgan Kaufmann, San Francisco (1995)
10. Mühlenbein, H., Gorges-Schleuter, M., Krämer, O.: Evolution algorithms in combinatorial optimization. Parallel Comput. **7**, 65–85 (1988)
11. Bergeron, A., Chauve, C., de Montgolfier, F., Raffinot, M.: Computing common intervals of K permutations, with applications to modular decomposition of graphs. In: Proceedings of the 13th Annual European Symposium on Algorithms, ESA 2005. Lecture Notes in Computer Science, vol. 3669, pp. 779–790. Springer, Berlin (2005)
12. Heber, S., Savage, C.: Common intervals of trees. Inf. Process. Lett. **93**, 69–74 (2005)
13. Booth, K.S., Lueker, G.S.: Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. J. Comput. Syst. Sci. **13**, 335–379 (1976)
14. Fulkerson, D., Gross, O.: Incidence matrices with the consecutive 1s property. Bull. Am. Math. Soc. **70**, 681–684 (1964)
15. Golumbic, C.: Algorithmic Graph Theory and Perfect Graphs. Academic Press, New York (1990)
16. Uno, T., Yagiura, M.: Fast algorithms to enumerate all common intervals of two permutations. Algorithmica **26**, 290–309 (2000)