Finding all Common Intervals of k Permutations

Steffen Heber[1,2] * and Jens Stoye[1] **

[1] Theoretical Bioinformatics (H0300)
[2] Functional Genome Analysis (H0800)
German Cancer Research Center (DKFZ) Heidelberg, Germany
{s.heber,j.stoye}@dkfz.de

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 K common intervals in optimal $O(nk+K)$ time and $O(n)$ additional space.
This extends a result by Uno and Yagiura (*Algorithmica* 26, 290–309, 2000) who present an algorithm to find all K common intervals of $k = 2$ permutations in optimal $O(n+K)$ 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.

1 Introduction

Let $\Pi = (\pi_1, \ldots, \pi_k)$ be a family of k permutations of $N = \{1, 2, \ldots, n\}$. A k-tuple of intervals of these permutations consisting of the same set of elements is called a *common interval*.

Common intervals have applications in different fields. The *consecutive arrangement problem* is defined as follows [1, 3, 4]: 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. Some 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 [2, 5, 7]. In a bioinformatical context, common intervals can be used to detect possible functional associations between genes. It is supposed that genes occurring in different genomes in each other's neighborhood tend to encode

* Present address: Department of Computer Science & Engineering, APM 3132, University of California, San Diego, La Jolla, CA 92093-0114, USA. E-mail: sheber@ucsd.edu

** Present address: Max Planck Institute for Molecular Genetics, Ihnestr. 73, Berlin, Germany. E-mail: stoye@molgen.mpg.de

functionally interacting proteins [8, 6, 9]. If one models genomes as permutations of genes, the problem of finding co-occurring genes translates into the problem of finding common intervals.

Recently, Uno and Yagiura [10] presented three algorithms for finding all common intervals of $k = 2$ permutations π_1 and π_2: two simple $O(n^2)$ time algorithms and one more complicated $O(n + K)$ time algorithm where $K \leq \binom{n}{2}$ is the number of common intervals of π_1 and π_2. Since the latter algorithm runs in time proportional to the size of the input plus the size of the output, it is optimal in the sense of worst case complexity.

An obvious extension of this algorithm to find all common intervals of a family $\Pi = (\pi_1, \ldots, \pi_k)$ of $k \geq 2$ permutations would be to compare π_1 successively with π_i for $i = 2, \ldots, k$ and report those intervals that are common in all comparisons. This yields an $O(kn + \sum_{i=2}^{k} K_i)$ time algorithm where K_i is the number of common intervals of π_1 and π_i for $2 \leq i \leq k$. The main result of this paper is an improvement of this approach by a non-trivial extension of Uno and Yagiura's algorithm, yielding an optimal $O(kn+K)$ time and $O(n)$ space algorithm where K is the number of common intervals of Π. Note that this number can be considerably smaller than any of the K_i.

The approach relies on restricting the set of all common intervals C to a smaller subset of *irreducible* intervals I, from which C can be easily reconstructed. While the number of common intervals can be as large as $\binom{n}{2}$, we show that $1 \leq |I| \leq n - 1$ and present an algorithm to compute I in optimal $O(kn)$ time, i.e., in time proportional to the input size. Knowing I we can reconstruct C in $O(K)$ time, i.e., in time proportional to the output size. Both algorithms use $O(n)$ additional space and their combination yields our main result.

2 Permutations and Common Intervals

Given a permutation π of (the elements of) the set $N := \{1, 2, \ldots, n\}$, we denote by $\pi(i) = j$ that the ith element of π is j. For $x, y \in N$, $x \leq y$, $[x, y]$ denotes the set $\{x, x + 1, \ldots, y\} \subseteq N$ and $\pi([x, y]) := \{\pi(i) \mid i \in [x, y]\}$ is called an *interval* of π. Let $\Pi = (\pi_1, \ldots, \pi_k)$ be a family of k permutations of N. W.l.o.g. we assume in the following always that $\pi_1 = id_n := (1, \ldots, n)$. A k-tuple $c = ([l_1, u_1], \ldots, [l_k, u_k])$ with $1 \leq l_j < u_j \leq n$ for all $1 \leq j \leq k$ is called a *common interval* of Π if and only if

$$\pi_1([l_1, u_1]) \;=\; \pi_2([l_2, u_2]) \;=\; \ldots \;=\; \pi_k([l_k, u_k]).$$

This allows to identify a common interval c with the contained elements, i.e.

$$c \equiv \pi_j([l_j, u_j]) \quad \text{for} \quad 1 \leq j \leq k.$$

Since $\pi_1 = id_n$, the above set equals the index set $[l_1, u_1]$, and we will refer to this as the *standard notation* of c. The set of all common intervals of Π is denoted C_Π. Note that our definition excludes common intervals of size one.

Example 1. Let $N = \{1, \ldots, 9\}$ 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 $\pi_3 = (1, 2, 3, 8, 7, 4, 5, 6, 9)$. We have

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

3 Finding All Common Intervals of Two Permutations

In order to keep this paper self-contained, here we briefly recall the algorithm RC (short for *Reduce Candidate*) of Uno and Yagiura [10] that finds all K common intervals of $k = 2$ permutations $\pi_1 = id_n$ and π_2 of N in $O(n + K)$ time and $O(n)$ space. For the correctness and analysis of the algorithm we refer to [10].

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

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

Since $f(x, y)$ counts the number of elements in $[l(x, y), u(x, y)] \setminus \pi_2([x, y])$, an interval $\pi_2([x, y])$ 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 $1 \leq x < y \leq n$ 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 to test $f(x, y) = 0$ 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 $x' \leq x$.*

In Algorithm RC (Algorithm 1), the common intervals are found using a data structure Y consisting of a doubly-linked list *ylist* for indices

of non-wasteful right interval end candidates and, storing intervals of *ylist*, two further doubly-linked lists *llist* and *ulist* that implement the functions l and u in order to compute f efficiently. They are also essential for an efficient update of *ylist*. In our pseudocode we use the standard 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 $N = \{1, \ldots, n\}$.
Output: C_Π in standard notation.

```
1: initialize Y
2: for x = n − 1, ..., 1 do
3:    update Y    // (see Algorithm 2)
4:    y ← x
5:    while (y ← ylist.succ(y)) defined and f(x, y) = 0 do
6:       output [l(x, y), u(x, y)]
7:    end while
8: end for
```

In the first step of Algorithm 1, *ylist* is initialized containing one element that stores the index n, and *llist* and *ulist* are both initialized with the one-element interval $[n, n]$ consisting of the last/only element of *ylist*.

Then Y is updated iteratively. A counter x (corresponding to the currently investigated left interval end) runs from $n-1$ down to 1. For any fixed x, the elements of *llist* are maximal intervals of *ylist* such that for an interval $[y, y']$ we have $l(x, y) = l(x, ylist.succ(y)) = \cdots = l(x, y')$; similar for *ulist*. For an interval $[y, y']$ in *llist* or *ulist*, we define its *value* by $val([y, y']) := \pi_2(y)$ and its *end* by $end([y, y']) := y'$. Algorithm 2 shows the update procedure for $\pi_2(x) > \pi_2(x+1)$. The case $\pi_2(x) < \pi_2(x+1)$ is treated in a symmetric way.

First, index x is prepended at the head of *ylist* and $[x, x]$ is prepended at the head of *llist*. Then *ylist* is trimmed by deleting all elements y ($> x$) that can be concluded to be wasteful (lines 3–12). This is called TRIMMING_YLIST in [10]. Simultaneously, *ulist* is trimmed in line 3. Finally, the interval ends of the new head of *ulist*, u^*, are updated.

Coming back to Algorithm 1, Uno and Yagiura show that in iteration step x, after the update of Y, the function $f(x, y)$ is monotonically increasing for the elements y remaining in *ylist*. This allows in lines 5–7 to find efficiently all common intervals with left end x by evaluating $f(x, y)$

Algorithm 2 (Update of data structure Y in line 3 of Algorithm 1)

```
 1: prepend x at the head of ylist
 2: prepend [x, x] at the head of llist
 3: while (u* ← ulist.head) has a successor u and val(u) < π2(x) do
 4:    delete u* from ulist and the corresponding elements from ylist
 5: end while
 6: y* ← end(u*)
 7: if (ỹ ← ylist.succ(y*)) is defined then
 8:    while f(x, y*) > f(x, ỹ) do
 9:       delete y* from ylist
10:       y* ← ylist.pred(ỹ)
11:    end while
12: end if
13: update the left and right end of u* ← [x, y*]
```

running left-to-right through *ylist* until an index y is encountered with $f(x, y) > 0$.

4 Irreducible Intervals

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

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

$$\begin{aligned} |c_1 \cap c_2| \geq 2 \quad &\Leftrightarrow \quad c_1 \cap c_2 \in C_\Pi, \\ c_1 \cap c_2 \neq \emptyset \quad &\Rightarrow \quad c_1 \cup c_2 \in C_\Pi. \end{aligned}$$

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

Two common intervals $c_1, c_2 \in C_\Pi$ have a *non-trivial overlap* if $c_1 \cap c_2 \neq \emptyset$ and they do not include each other. A list $p = (c_1, \ldots, c_{\ell(p)})$ of common intervals $c_1, \ldots, c_{\ell(p)} \in C_\Pi$ is a *chain* (of length $\ell(p)$) if every two successive intervals in p have a non-trivial overlap. A chain of length one is called a *trivial chain*, all other chains are called *non-trivial chains*. A chain that can not be extended to its left or right is a *maximal chain*. By Lemma 1, every chain p *generates* a common interval $c = \tau(p) := \bigcup_{c' \in p} c'$.

Definition 2. *A common interval c is called* reducible *if there is a non-trivial chain that generates c, otherwise it 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. Obviously, $1 \leq |I_\Pi| \leq |C_\Pi| \leq \binom{n}{2}$. 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 2. *Let Π be a family of permutations, $c \in C_\Pi$ a common interval, and $(b_1, \ldots, b_\ell)$ a chain of irreducible intervals generating c. The nesting levels of c and all the b_i for $i = 1, \ldots, \ell$ are equal.*

Proof (Sketch). Let n_c be the nesting level of c and n_i the nesting level of b_i for $i = 1, \ldots, \ell$. Since $b_i \subseteq c$ we have $n_i \geq n_c$ for $i = 1, \ldots, \ell$. If $n_i > n_c$, there exists an irreducible interval $c^* \not\supset c$ with $b_i \subset c^*$ and $\ell > 1$. Now we distinguish between internal and terminal intervals b_i in the chain. In both cases one can easily see that c^* can be generated by smaller common intervals, contradicting the assumption that c^* is irreducible. □

We can further partition I_Π into maximal chains. This partitioning is unique. For a maximal chain $p = (c_1, \ldots, c_{\ell(p)})$ and $1 \leq i \leq j \leq \ell(p)$, we call $p[i, j] := (c_i, \ldots, c_j)$ a *subchain* of p.

Lemma 3. *The set of common intervals that is generated from the subchains of the maximal chains of I_Π equals C_Π.*

Proof. This follows directly from the definition of the partition. □

Example 1 (cont'd). For $\Pi = (\pi_1, \pi_2, \pi_3)$ as above, the irreducible intervals are

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

The reducible intervals are generated as follows:

$$\begin{aligned} [1,3] &= [1,2] \cup [2,3], \\ [1,9] &= [1,8] \cup [4,9], \\ [4,6] &= [4,5] \cup [5,6]. \end{aligned}$$

A sketch of the structure of maximal chains of irreducible intervals and their nesting levels is shown in Figure 1.

Lemma 4. *Given two different maximal chains p_1 and p_2, exactly one of the following alternatives is true:*

- *$\tau(p_1)$ and $\tau(p_2)$ are disjoint,*

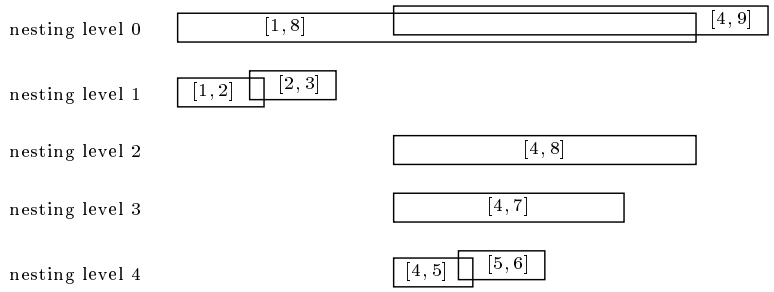

Fig. 1. Visualization of the irreducible intervals in I_Π and their nesting levels.

- $\tau(p_1)$ *is contained in a single element of* p_2*, or*
- $\tau(p_2)$ *is contained in a single element of* p_1*.*

Proof. $\tau(p_1)$ and $\tau(p_2)$ are either disjoint or have a non-empty intersection. In the latter case, $\tau(p_1)$ and $\tau(p_2)$ cannot overlap non-trivially, because of the maximality of p_1 and p_2. Therefore, w.l.o.g. suppose $\tau(p_2) \subseteq \tau(p_1)$. No element of p_2 can overlap non-trivially with any element of p_1, otherwise one could find an element of p_1 or p_2 that is generated by smaller intervals, contradicting its irreducibility. This yields the existence of exactly one irreducible interval c of p_1 that includes $\tau(p_2)$ completely, while no other element of p_1 overlaps with $\tau(p_2)$. □

Based on the above lemmas, we describe a linear 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 (Algorithm 3). The algorithm partitions I_Π into maximal chains (line 1). This can be done, for example, by the following three steps. First, I_Π is partitioned according to the nesting level. This is possible in $O(|I_\Pi|)$ time by applying a sweep line technique to all interval start and end points. Then the intervals in the resulting classes are sorted by their left end. Using radix sort, this can also be done in $O(|I_\Pi|)$ time. Finally, the classes are further refined at non-overlapping consecutive intervals, yielding the maximal chains of irreducible intervals. This again takes $O(|I_\Pi|)$ time. Using Lemma 3, we create C_Π by generating all subchains of the maximal chains (lines 2–4). This takes $O(|C_\Pi|)$ time. Since $|I_\Pi| \leq |C_\Pi|$, Algorithm 3 takes $O(|C_\Pi|)$ time in total.

The following theorem is the basis for the complexity analysis of our algorithm in the following section.

Algorithm 3 (Reconstruct C_Π from I_Π)

Input: I_Π in standard notation
Output: C_Π in standard notation
1: partition I_Π into maximal chains $p_1, p_2, \ldots$
2: **for each** $p_m = (b_1, \ldots, b_{\ell(p_m)})$ **do**
3: output $\tau(p_m[i,j])$ in standard notation **for all** $1 \le i \le j \le \ell(p_m)$
4: **end for**

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

Proof. For each interval $[j, j+1]$, $j = 1, \ldots, n-1$, of π_1 denote by $b_{[j,j+1]} \in I_\Pi$ the irreducible interval of smallest cardinality containing $[j, j+1]$. It is easy to see that $b_{[j,j+1]}$ is uniquely defined. For any $c = [x, y] \in C_\Pi$, a subset of $\{b_{[x,x+1]}, \ldots, b_{[y-1,y]}\}$ generates c. This yields $\{b_{[j,j+1]} \mid j = 1, \ldots, n-1\} = I_\Pi$. □

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

5 Finding All Irreducible Intervals of k Permutations

In this section we present our algorithm that finds all irreducible intervals of a family $\Pi = (\pi_1, \pi_2, \ldots, \pi_k)$ of $k \ge 2$ permutations of $N = \{1, \ldots, n\}$ in $O(kn)$ time. Together with Algorithm 3 this allows to find all K common intervals of Π in optimal $O(kn + K)$ time.

5.1 Outline of the Algorithm

For $1 \le i \le k$, set $\Pi_i := (\pi_1, \ldots, \pi_i)$. Starting with $I_{\Pi_1} = \{[j, j+1] \mid 1 \le j < n\}$, the algorithm successively computes I_{Π_i} from $I_{\Pi_{i-1}}$ for $i = 2, \ldots, k$ (see Algorithm 4). To construct I_{Π_i} from $I_{\Pi_{i-1}}$, we define the mapping

$$\varphi_i : I_{\Pi_{i-1}} \to I_{\Pi_i}$$

where for $c \in I_{\Pi_{i-1}}$, $\varphi_i(c)$ is the smallest common interval $c' \in C_{\Pi_i}$ that contains c. Since $I_{\Pi_i} \subseteq C_{\Pi_i} \subseteq C_{\Pi_{i-1}}$ and, by Lemma 3, $I_{\Pi_{i-1}}$ generates the elements of $C_{\Pi_{i-1}}$, $I_{\Pi_{i-1}}$ also generates I_{Π_i}. One can easily see that $c' \in I_{\Pi_i}$ and that φ_i is surjective, i.e. $I_{\Pi_i} = \{\varphi_i(c) \mid c \in I_{\Pi_{i-1}}\}$. This implies the correctness of Algorithm 4. In Section 5.2 we will show how

$\varphi_i(I_{\Pi_{i-1}})$ can be computed in $O(n)$ time and space, yielding the $O(kn)$ time complexity to compute I_Π $(= I_{\Pi_k})$.

Algorithm 4 (Computation of I_{Π_k})

Input: A family $\Pi = (\pi_1 = id_n, \pi_2, \ldots, \pi_k)$ of k permutations of $N = \{1, \ldots, n\}$.
Output: I_Π in standard notation.

```
1: I_{Π_1} ← ([1,2],[2,3],...,[n−1,n])
2: for i = 2,...,k do
3:     I_{Π_i} ← {φ_i(c) | c ∈ I_{Π_{i−1}}}    // (see Algorithm 5)
4: end for
5: output I_{Π_k} in standard notation
```

5.2 Computing I_{Π_i} from $I_{\Pi_i\Pi_1}$

For the computation of $\varphi_i(I_{\Pi_{i-1}})$ we use a modified version of Algorithm RC where the data structure Y is supplemented by a data structure S that is derived from $I_{\Pi_{i-1}}$. S consists of several doubly-linked lists of intervals of *ylist*, one for each maximal chain of $I_{\Pi_{i-1}}$.

Using π_1 and π_i, as in Algorithm RC, the *ylist* of Y allows for a given x to access all non-wasteful right interval end candidates y of $C_{(\pi_1,\pi_i)}$. The aim of S is to further reduce these candidates to only those indices y for which simultaneously $[x, y] \in C_{\Pi_{i-1}}$ (ensuring $[x, y] \in C_{\Pi_i}$) and $[x, y]$ contains an interval $c \in I_{\Pi_{i-1}}$ that is not contained in any smaller interval from C_{Π_i}. Together this ensures that exactly the irreducible intervals $[x, y] \in I_{\Pi_i}$ are reported.

An outline of our modified version of Algorithm RC is shown in Algorithm 5. Since the first permutation handed to the algorithm has to be the identity and S (derived from $I_{\Pi_{i-1}}$) is compatible only with the index set of π_1, we supply the algorithm with $id_n = \pi_i^{-1} \circ \pi_i$ and $\pi_i^{-1} \circ \pi_1$ instead of $\pi_1 (= id_n)$ and π_i. (As usual, π_i^{-1} denotes the inverse of permutation π_i.) This does not change the index set of the computed irreducible intervals.

In line 1 of Algorithm 5, Y is initialized as in Algorithm RC. To initialize S, $I_{\Pi_{i-1}}$ is partitioned into maximal chains of non-trivially overlapping irreducible intervals as in line 1 of Algorithm 3. For each such chain, S contains a doubly-linked *clist* that initially holds the intervals of that chain in left-to-right order. Moreover, intervals from different *clist*s with the same left end are connected by *vertical pointers* yielding for each index $x \in N$ a doubly-linked *vertical list*. It is not difficult to add the vertical pointers during the construction of the *clist*s such that the intervals

Algorithm 5 (Extended Algorithm RC)

Input: Two permutations $\pi_1 = id_n$ and $\pi_2 = \pi_i^{-1}$ of $N = \{1, \ldots, n\}$; $I_{\Pi_{i-1}}$ in standard notation.
Output: I_{Π_i} in standard notation.

```
1: initialize Y and S
2: for x = n − 1, ..., 1 do
3:    update Y and S    // (see text)
4:    while ([x', y] ← S.first_active_interval(x)) defined and f(x, y) = 0 do
5:       output [l(x, y), u(x, y)]
6:       remove [x', y] from its active sublist    // (the interval is satisfied)
7:    end while
8: end for
```

in each vertical list are ordered by increasing length (decreasing nesting level).

To describe the update of S in line 3 of the algorithm, we introduce the notion of *sleeping*, *active*, and *satisfied* intervals. Initially all intervals of the *clist*s are sleeping. In iteration step x, all intervals with left end x become active and are included at the head of an (initially empty) *active sublist* of their *clist*. An interval remains active until it is satisfied or deleted. A *clist* L and the contained intervals are deleted whenever x becomes smaller than the left interval end of $L.head$. It might be that the right end y of an interval $[x, y]$ at the time of activation is already deleted from the *ylist*. In this case, the interval is merged with the successing interval $[x', y']$ in its *clist*, i.e. the corresponding two elements of *clist* are replaced by a new one, containing the interval $[x, y']$. If no successor exists, the interval $[x, y]$ is deleted.

Concerning the function φ_i, sleeping or active intervals correspond to irreducible intervals from $I_{\Pi_{i-1}}$ whose images have not yet been determined. The status changes to satisfied when the image is known.

The update of Y in line 3 is the same as in Algorithm RC, the only difference being that whenever an element y is removed from *ylist* and y is the right end of some active or satisfied interval, this interval is merged with its successor in its *clist* if such a successor exists, otherwise it is deleted. The resulting interval inherits the active status if one of the merged intervals was active, otherwise it is satisfied. (If both merged intervals are active, this reflects the case that φ_i maps both intervals of $I_{\Pi_{i-1}}$ to the same (larger) interval of I_{Π_i}.) Note that even though y can be the right end of many irreducible intervals, at any point of the algorithm y can be the right end of at most one active or satisfied interval. This is due to the fact that no two intervals of a maximal chain can have the

same right end, and whenever two intervals from different chains have the same right end, the chain of the shorter interval is deleted before the longer interval is made active (cf. Lemma 4). Hence it suffices to keep for each index $y > x$ a pointer to the (only) active or satisfied interval with right end y. This *right end pointer* is set when the interval is made active and is deleted when the interval's *clist* is deleted.

In contrast to the simple traversal of the *ylist* in Algorithm RC, here the generation of right interval end candidates in lines 4–7 is slightly more complicated. Function $S.first\_active\_interval(x)$ returns the first active interval $[x', y]$ in the *clist* of the interval at the head of the vertical list at index x. If the right end y of this interval gives rise to a common interval $[l(x, y), u(x, y)]$, i.e., if $f(x, y) = 0$, a common interval of smallest size containing an active interval is encountered. Hence we have found an element of $\varphi_i(I_{\Pi_{i-1}})$ which then is reported in line 5. Therefore $[x', y]$ becomes satisfied and is removed from its active sublist in line 6. In case this interval was the last active interval of its *clist*, the pointer to the head of the vertical list at index x is redirected to the successor of the current head, such that in the next iteration $S.first\_active\_interval(x)$ returns the leftmost active interval from the *clist* with the next lower nesting level (if such a list containing an interval with left end x exists).

This way we only look at elements of *ylist* that are candidates for right ends of minimal common intervals with left end x and that contain an active interval. $S.first\_active\_interval(x)$ generates these candidates in left-to-right order such that, since $f(x, y)$ is monotonically increasing for the elements y of *ylist* and hence also for the elements of any sublist of *ylist*, by evaluating $f(x, y)$ until an index y is encountered with $f(x, y) > 0$, all irreducible intervals from I_{Π_i} with left end x are found. This implies the correctness of our implementation of φ_i.

The complete data structure S for $\Pi = (\pi_1, \pi_2, \pi_3)$ as in Example 1 while processing index $x = 4$ of permutation π_3 is shown in Figure 2.

5.3 Analysis of Algorithm 5

Since all operations modifying Y are the same as in Algorithm RC, this part of the analysis carries over from [10], and we can restrict our analysis to the initialization and update of S.

The initialization of S in line 1, including the creation of the vertical lists, can easily be implemented in linear time in a way similar to the first step of Algorithm 3.

In line 3, the intervals with left end x are easily found using the vertical lists, marked active, and prepended to the active sublists in constant time

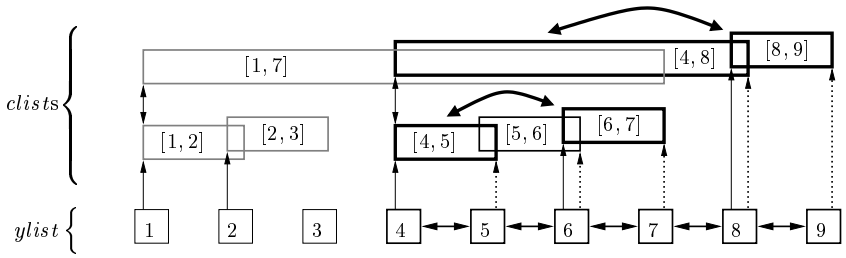

Fig. 2. Sketch of *ylist* and the *clist*s while processing element $x = 4$ of π_3 for $\Pi = (\pi_1, \pi_2, \pi_3)$ as in Example 1. Shaded boxes represent sleeping intervals, boxes with thick solid lines represent active intervals, and boxes with thin lines represent satisfied intervals. Thick arrows connect the elements of the active sublists, solid vertical arrows denote vertical lists (the vertical pointer of index 5 was deleted after reporting interval $[5, 6]$ in iteration step $x = 5$), and dotted vertical arrows are the right end pointers.

per interval. Since $I_{\Pi_{i-1}}$ contains $O(n)$ intervals and since each interval is activated exactly once, this step takes overall $O(n)$ time. Moreover, each index y that is deleted from the *ylist* can cause the merge of two intervals. Since merging two neighbors in a doubly-linked list takes constant time, and since each of the in total n elements of *ylist* is deleted at most once, this part takes overall $O(n)$ time as well.

As in Algorithm RC, the time required for reporting the output is proportional to the size of the output, here $|I_{\Pi_i}| < n$. Using vertical list and active sublist, the first active interval is found in constant time. Hence, and since the removal of interval $[x', y]$ from the active sublist in line 6 is a constant-time operation as well, the loop in lines 4–7 takes overall $O(n)$ time.

Putting things together, Algorithm 5 takes $O(n)$ time and space. Since at any point of Algorithm 4 we need to store only two permutations π_1 and π_i and the current I_{Π_i}, we have

Theorem 2. *The irreducible intervals of k permutations of n elements can be found in optimal $O(kn)$ time and $O(n)$ additional space.*

Combining this result with Algorithm 3, we get

Corollary 1. *The K common intervals of k permutations of n elements can be found in optimal $O(kn + K)$ time and $O(n)$ additional space.*

Acknowledgments

We wish to thank Richard Desper, Dan Gusfield, and Christian N. S. Pedersen for helpful comments.

References

1. K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976.
2. R. M. Brady. Optimization strategies gleaned from biological evolution. *Nature*, 317:804–806, 1985.
3. D. Fulkerson and O. Gross. Incidence matrices with the consecutive 1s property. *Bull. Am. Math. Soc.*, 70:681–684, 1964.
4. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.
5. S. Kobayashi, I. Ono, and M. Yamamura. An efficient genetic algorithm for job shop scheduling problems. In *Proc. of the 6th International Conference on Genetic Algorithms*, pages 506–511. Morgan Kaufmann, 1995.
6. E. M. Marcotte, M. Pellegrini, H. L. Ng, D. W. Rice, T. O. Yeates, and D. Eisenberg. Detecting protein function and protein-protein interactions from genome sequences. *Science*, 285:751–753, 1999.
7. H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer. Evolution algorithms in combinatorial optimization. *Parallel Comput.*, 7:65–85, 1988.
8. R. Overbeek, M. Fonstein, M. D'Souza, G. D. Pusch, and N. Maltsev. The use of gene clusters to infer functional coupling. *Proc. Natl. Acad. Sci. USA*, 96(6):2896–2901, 1999.
9. B. Snel, G. Lehmann, P. Bork, and M. A. Huynen. STRING: A web-server to retrieve and display the repeatedly occurring neigbourhood of a gene. *Nucleic Acids Res.*, 28(18):3443–3444, 2000.
10. T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.