

Revisiting T. Uno and M. Yagiura's Algorithm^{*}

(Extended Abstract)

Binh-Minh Bui Xuan, Michel Habib, and Christophe Paul

CNRS - LIRMM, 161 rue Ada, 34392 Montpellier Cedex 5, France
{buixuan, habib, paul}@lirmm.fr

Abstract. In 2000, T. Uno and M. Yagiura published an algorithm that computes all the K common intervals of two given permutations of length n in $\mathcal{O}(n + K)$ time. Our paper first presents a decomposition approach to obtain a compact encoding for common intervals of d permutations. Then, we revisit T. Uno and M. Yagiura's algorithm to yield a linear time algorithm for finding this encoding. Besides, we adapt the algorithm to obtain a linear time modular decomposition of an undirected graph, and thereby propose a formal invariant-based proof for all these algorithms.

1 Introduction

T. Uno and M. Yagiura's algorithm [23] computes all the K *common intervals* of two permutations of length n in $\mathcal{O}(n + K)$ time. Therein, each genome is regarded as a permutation on a finite set of genes, and a common interval of two genomes refers to a set of genes that are consecutive on each genome. This notion formalises the concept of a gene cluster. Afterwards, F. de Montgolfier pointed out strong relationships between *modules* of a *permutation graph* and common intervals of any of its *realiser* (made of two permutations) [12]. This allows to define the *common interval decomposition tree* for this case. From recent works, the tree turns out to own some important biological meaning [2, 19]. Particularly, common intervals help out with finding evolutionary distances between the corresponding species [4, 14, 19]. Finally, common intervals can be interpreted as pieces of each genome that have been *conserved* all along an evolutionary scenario between the involved species and their common ancestor [2].

The seminal algorithmic result on common intervals is due to T. Uno and M. Yagiura (Fig. 1). This really is a masterpiece among combinatorial algorithms as it uses a unique scan on one of the two permutations and could be seen as an application of a sweep plane paradigm as used in computational geometry [11]. However, its correctness proof is tough to understand. Later, S. Heber and J. Stoye pointed out a smaller and generating sub-family, so-called the family of *irreducible* common intervals, and succeeded in adapting T. Uno and M. Yagiura's algorithm to find all irreducible common intervals of d permutations in $\mathcal{O}(d \times n)$ time [17]. Besides, generating all the K common intervals from this sub-family is in $\mathcal{O}(K)$ time [17]. While they used T. Uno and M. Yagiura's

^{*} Full version available at <http://www.lirmm.fr/~buixuan> as RR-LIRMM-05049

T. Uno and M. Yagiura's general scheme:

1. Let **Potential** be an empty list
2. **For** $i = n$ down to 1 **Do**
3. (Filter): Remove all known boundaries r in **Potential** such that
 for all $l \leq i$, (l, r) is not a common interval
4. (Add): Add i to the head of **Potential**
5. (Extract): While there still is some boundary r of **Potential** such
 that (i, r) is a common interval, output (i, r)
6. **End of for**

Fig. 1. A list **Potential** is used. It contains at each step i all boundaries $r \geq i$ such that there is some $l \leq i$ with (l, r) a common interval. Then, **Potential** is traced to output the common intervals of the form (i, r) . The main difficulty of such approach relies on the linear time complexity while the idea is based on a double iteration.

scheme as a black box, they did not give further explanations for the correctness proof. Recently, A. Bergeron et al. bypassed this difficult issue and devised an alternative algorithm together with its combinatorial proof [3].

In this paper, we propose a complete invariant-based proof of T. Uno and M. Yagiura's algorithm, as well as its complexity analysis. We also show how it can easily be adapted to compute in $\mathcal{O}(n)$ time a tree representation of all common intervals of two permutations on n elements. Then, Section 3 generalises T. Uno and M. Yagiura's algorithm, and uses it as a central step for modular decomposition algorithms of undirected graphs.

2 Common Interval Decomposition

Let us denote $\mathbb{N}_n = \llbracket 1, n \rrbracket = \{1, 2, \dots, n\}$. A permutation π on a finite set V is regarded indifferently as a bijection from $\mathbb{N}_{|V|}$ to V , a total order on V , or a word in V^* without multiple occurrence. The support of a factor of π is called an *interval of π* , noted $\pi(\llbracket l, r \rrbracket)$ with $l, r \in \mathbb{N}_{|V|}$ its left and right boundaries. A *common interval* of two permutations on V is interval of each (see Figure 2). There could be a quadratic number of those, e.g. when the permutations are identical. The decomposition addressed in this paper is based on the seminal works on weakly partitive families [8, 22]. Let us recall some useful formalisms.

2.1 Combinatorial Decomposition Aspects

Let V be a given finite set. Two subsets of V *overlap* when none of their intersection and differences is empty. A family $\mathcal{F} \subseteq 2^V$ is *weakly partitive* if and only if $\emptyset \notin \mathcal{F}$, \mathcal{F} contains the trivial subsets (singletons and V), and \mathcal{F} is closed by intersection, union, and differences on overlapping subsets. It is *partitive* if weakly partitive and closed by symmetric difference on overlapping subsets [8, 22]. A weakly partitive family can have $\mathcal{O}(n!)$ members, e.g. with $\mathcal{F} = 2^V$.

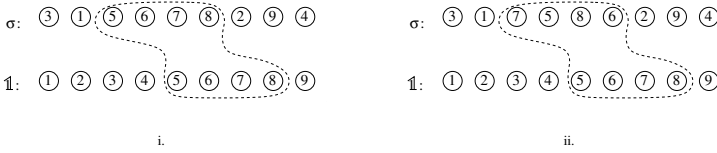


Fig. 2. In both examples, $\sigma(\llbracket 3, 6 \rrbracket) = \{5, 6, 7, 8\} = \mathbb{1}(\llbracket 5, 8 \rrbracket)$ is a common interval



Fig. 3. Common interval decomposition tree. “L” stands for *Linear* and “P” for *Prime*.

Let $\mathcal{F} \subseteq 2^V$ be weakly partitive. A member $S \in \mathcal{F}$ is *strong* when it does not overlap any other $F \in \mathcal{F}$. The subset of \mathcal{F} containing all strong members of \mathcal{F} is denoted $\mathcal{S}_{\mathcal{F}}$. The members of $\mathcal{S}_{\mathcal{F}}$ can be organised by inclusion order in a tree, so-called the *decomposition tree* and noted $\mathcal{T}_{\mathcal{F}}$. The size of $\mathcal{T}_{\mathcal{F}}$ is $\mathcal{O}(n)$.

Theorem 1. [8, 22] *Except for binary nodes, an internal node in $\mathcal{T}_{\mathcal{F}}$ satisfies one and only one of the following: (Prime node) no union of children belongs to \mathcal{F} , except for the node itself; (Degenerate node) all union of children belongs to \mathcal{F} ; (Linear node) there is a children ordering such that a union of children belongs to \mathcal{F} if and only if they are consecutive in this order.*

Roughly, the tree $\mathcal{T}_{\mathcal{F}}$ is a (compact) encoding of \mathcal{F} from which all members of \mathcal{F} can easily be generated. A permutation σ is *factorising* for \mathcal{F} if and only if any strong subset $S \in \mathcal{S}_{\mathcal{F}}$ is an interval of σ [7]. In other words, a factorising permutation is a visit-order of the leaves of $\mathcal{T}_{\mathcal{F}}$ by a depth-first graph search. Though the following property is trivial, it yields a formal decomposition framework for common intervals. Fig. 3 exemplifies the common interval decomposition.

Property 1. *The family \mathcal{CI} of common intervals of two permutations σ_1 and σ_2 satisfies three following properties: \mathcal{CI} is weakly partitive; $\mathcal{T}_{\mathcal{CI}}$ has no Degenerate nodes; and both σ_1 and σ_2 are factorising.*

A common interval is *reducible* if it is union of consecutively overlapping non-trivial common intervals, and is *irreducible* when not reducible [17]. This notion can easily be generalised to any weakly partitive family. Now, it is straightforward from the definitions that the irreducible common intervals exactly are *Prime* nodes and pairs of consecutive children of *Linear* nodes of the decomposition tree. Hence, one can compute in $\mathcal{O}(n)$ time the family of irreducible common intervals from the decomposition tree and conversely. In this paper, we would rather focus on the notion of *right-strong intervals*. However, notice that both notions of irreducibility and right-strong interval merely are combinatorial

tools to remove the term “ K ” in the raw $\mathcal{O}(n + K)$ common intervals computing time. Fortunately, both of them can be adapted in T. Uno and M. Yagiura's sweep paradigm.

2.2 Right-Strong Intervals

Let $\sigma = \sigma_1$ and σ_2 be two permutations on V . Let \mathcal{CI} refer to the family of their common intervals. Then, σ is factorising for \mathcal{CI} . W.l.o.g., from now on, *intervals* will stand for intervals of σ . By definition, a common interval is an interval.

Definition 1 (Right-Strong Interval). *Given a factorising permutation σ for a (weakly) partitive family $\mathcal{F} \subseteq 2^V$, an interval $\sigma(\llbracket i, j \rrbracket) \in \mathcal{F}$ is right-strong if and only if it does not overlap on its right any other interval of σ that belongs to \mathcal{F} , namely if and only if $i < i' \leq j < j'$ implies $\sigma(\llbracket i', j' \rrbracket) \notin \mathcal{F}$.*

Roughly, a right-strong interval of \mathcal{CI} is a member of \mathcal{CI} that does not overlap any other member of \mathcal{CI} on its right in the order σ . Their number is bounded by $2 \times n$ from Corollary 1 below. To formalise their computation, let us define $\mathbf{Select}(i) = \{j \mid \sigma(\llbracket i, j \rrbracket) \text{ is a right-strong interval}\}$ for all $n \geq i \geq 1$.

Definition 2 (Useless Boundary). *While inspecting σ from n down to 1, $\sigma(\llbracket l, r \rrbracket)$ is visited at step i if $i < l$, unvisited otherwise. Then, $r \in \llbracket i, n \rrbracket$ is useless w.r.t. i if none of the unvisited right-strong intervals is of the form $\sigma(\llbracket l, r \rrbracket)$.*

Lemma 1. *Let m_i be the maximum boundary such that $\sigma(\llbracket i, m_i \rrbracket) \in \mathcal{F}$. Then, $m_i = \max \mathbf{Select}(i)$ and for all $i < r < m_{i+1}$, r is useless w.r.t. i .*

Proof. If $\sigma(\llbracket i, m_i \rrbracket)$ overlaps $\sigma(\llbracket i', m' \rrbracket)$ on its right, then $\sigma(\llbracket i, m' \rrbracket) \in \mathcal{F}$ (partitivity) and m_i is not maximum. Therefore, $m_i \in \mathbf{Select}(i)$. Then, $m_i = \max \mathbf{Select}(i)$ is trivial. Besides, for all $l < i + 1 \leq r < m_{i+1}$, $\sigma(\llbracket l, r \rrbracket)$ overlaps $\sigma(\llbracket i + 1, m_{i+1} \rrbracket)$ on its right. □

Corollary 1. $|\mathbf{Select}(1)| + \dots + |\mathbf{Select}(n)| \leq 2 \times n$.

Proof. From Lemma 1, the sets $\mathbf{Select}(i) \setminus \{\max \mathbf{Select}(i)\}$ ($1 \leq i \leq n$) are pairwise disjoint and their total cardinal is bounded by n . □

2.3 Right-Strong Intervals of Two Permutations Computation

With a slight modification, i.e. by adding an one-line routine, T. Uno and M. Yagiura's algorithm computes in $\mathcal{O}(n)$ time the family of *right-strong intervals* of two permutations $\sigma = \sigma_1$ and σ_2 on V , where $n = |V|$. However, we will detail its correctness, since the original version is tough to understand. The sets $\mathbf{Select}(i)$ ($n \geq i \geq 1$) will be computed using a list **Potential**. At each step i , this list contains the right boundaries $r \geq i$ of all unvisited right-strong intervals.

Potential is initialised as an empty list. Each step $n \geq i \geq 1$ aims at removing from **Potential** as many useless boundaries w.r.t. i as possible. For this purpose, let $C_2(i, j)$ refer to the convex hull in σ_2 of $\sigma(\llbracket i, j \rrbracket)$, i.e. $C_2(i, j) = \sigma_2(\llbracket l, r \rrbracket)$

where $l = \min\{k \mid \sigma_2(k) \in \sigma(\llbracket i, j \rrbracket)\}$ and $r = \max\{k \mid \sigma_2(k) \in \sigma(\llbracket i, j \rrbracket)\}$. We define $\mathcal{S}_{\sigma(\llbracket i, j \rrbracket)} = C_2(i, j) \setminus \sigma(\llbracket i, j \rrbracket)$ as the *splitter set* of $\sigma(\llbracket i, j \rrbracket)$. Roughly, a splitter makes an interval not a common interval. Let $s(\sigma(\llbracket i, j \rrbracket)) = |\mathcal{S}_{\sigma(\llbracket i, j \rrbracket)}| = s_i(j)$. We define $\delta_i(p_j) = s_i(p_{j+1}) - s_i(p_j)$ if a member p_j of **Potential** has a successor p_{j+1} . Otherwise, $\delta_i(p_j) = +\infty$. Then, Theorem 2 below is fundamental and most results thereafter rely on it. However, from our standpoint, the theorem is easier to prove and most comprehensive when generalised to Theorem 4 in Section 3.1.

Property 2. [23] $\sigma(\llbracket i, j \rrbracket)$ is a common interval if and only if $s_i(j) = 0$.

Theorem 2. [23] $\delta_i(p_j) < 0$ implies p_j is useless w.r.t. i .

At each step i , assume that some Update-Detect routine provides 1. for each p_j in **Potential** a pointer to the value of $s_i(p_j)$; and 2. a list **Detected** of pointers to all p_j with $\delta_i(p_j) < 0$, and possibly to some other useless boundaries w.r.t. i . Besides, assume that the pointed $p_{j_1} < \dots < p_{j_h}$ are organised increasingly.

Then, **Potential** is filtered twice. The first filtering (Pre-Filter) is our only addition to the original algorithm. It follows from Lemma 1, which states that it is possible to move apart some useless boundaries w.r.t. i even before considering $\sigma(i)$. Concisely, a pointer to $r_0 = \max \mathbf{Select}(i + 1)$ is maintained. Then, if r_0 has some predecessors in **Potential**, they are removed and r_0 receives the mark *Eaten*, which is for use in Section 2.4. The second filtering (Customised Filter) backtracks **Detected** from p_{j_h} down to p_{j_1} . Each p_{j_k} is removed from **Potential** if still there. If some removing makes the next-left boundary p' have $\delta_i(p') < 0$, p' is also removed and so on. Thus, only useless boundaries w.r.t. i are removed, and all remaining boundaries have positive δ_i . Both filtering takes linear time on the number of removed boundaries. The boundary i is then added to the head of **Potential** (Add) and the update of step i is complete. Notice that $\delta_i(i) \geq 0$.

Invariant 1. After the update of step i , let p_{j_0} be the first member of **Potential** with $s_i(p_{j_0}) \neq 0$. Then, $\mathbf{Select}(i) = \{r < p_{j_0} \mid r \text{ is a member of Potential}\}$.

Proof. After the update, all p_j have $\delta_i(p_j) \geq 0$. If $r \in \mathbf{Select}(i)$, then $s_i(r) = 0$ and $r < p_{j_0}$. Besides, $\sigma(\llbracket i, r \rrbracket)$ is unvisited at step i . Hence, r still is a member of **Potential**, and it is strictly before p_{j_0} . Conversely, any member $r < p_{j_0}$ of **Potential** after the update satisfies $s_i(r) = 0$. If $\sigma(\llbracket i, r \rrbracket)$ overlaps some $\sigma(\llbracket i', r' \rrbracket)$ on its right, then $i < i' \leq r < r'$, $\sigma(\llbracket i', r \rrbracket) \in \mathcal{CI}$, $\sigma(\llbracket i', r' \rrbracket) \in \mathcal{CI}$ and the Pre-Filter at step i' would remove r from **Potential** if it was still there. \square

Outputting **Selected**(i) from the list **Potential** (Extract) follows from Invariant 1. Its computing time obviously is linear on the size of the output.

Turning our attention to complexity issues, Corollary 1 and the fact that each boundary is inserted exactly once in **Potential** imply the following.

Result 1. The right-strong intervals computing time is $\mathcal{O}(n)$ if Update-Detect runs in linear time on the size of the output **Detected** at each iteration step i .

T. Uno and M. Yagiura's algorithm revisited:

1. Let **Potential** be an empty list and $\text{Select}(n+1) = \emptyset$
2. **For** $i = n$ down to 1 **Do**
3. (Update-Detect): Collect all known useless boundaries w.r.t. i
4. (Pre-Filter): If there are some $r < r_0 (= \max \text{Select}(i+1))$ in **Potential**, remove them and mark r_0 as *Eaten*
5. (Customised Filter): Remove all known useless boundaries w.r.t. i
6. (Add): Add the boundary i to the head of **Potential**
7. (Extract): Find the right-most r_q in **Potential** with $s_i(r_q) = 0$ and output $\text{Select}(i) = \{r_1 \dots r_q\}$
8. **End of for**

Update-Detect can be as follows [23]. Let $\text{Potential} = [p_1 (= i+1), \dots, p_l]$ at the beginning of step i . The routine updates two lists $\text{Min} = [\text{Min}_1, \dots, \text{Min}_s]$ and Max . Each $1 \leq \text{Min}_j \leq n$ is a boundary with two pointers $\text{first}(\text{Min}_j)$ and $\text{last}(\text{Min}_j)$ to two members of **Potential**. All p_j between these two members satisfy $\text{Min}_j = \min\{k \mid \sigma_2(k) \in \sigma(\llbracket i, p_j \rrbracket)\}$. Besides, each p_j in **Potential** has a pointer $\text{Min}(p_j)$ to the corresponding member of Min . It is analogous for Max . By supposing $V = \llbracket 1, n \rrbracket$, computing $s_i(p_j)$ from this structure is in $\mathcal{O}(1)$ time.

Let $\text{Min} = [\text{Min}'_1, \dots, \text{Min}'_{s'}]$ and $\text{Max} = [\text{Max}'_1, \dots, \text{Max}'_{t'}]$ at the beginning of step i . Suppose inductively that $C_2(i+1, p_j) = \sigma_2(\llbracket \text{Min}(p_j), \text{Max}(p_j) \rrbracket)$ for all p_j and that Min , resp. Max , is strictly decreasing, resp. increasing. Notice that $\sigma_2(\text{Min}'_1) = \sigma_2(\text{Max}'_1) = \sigma(i+1)$. Now, i' with $\sigma_2(i') = \sigma(i)$ can be obtained in $\mathcal{O}(1)$ time. Then, either $i' < \text{Min}'_1$ and Max will be unchanged, or $\text{Max}'_1 < i'$ and Min unchanged. We trace Min , resp. Max , from $j = 1$ until finding the first j^* with $\text{Min}'_{j^*} \leq i' < \text{Max}'_1$, resp. $\text{Min}'_1 < i' \leq \text{Max}'_{j^*}$. Notice that $j^* > 1$ and let $p_{j_0} = \text{first}(\text{Min}'_{j^*-1})$, resp. $p_{j_0} = \text{first}(\text{Max}'_{j^*-1})$.

Lemma 2. [23] p_j is useless w.r.t. i if $s_i(p_j) - s_{i+1}(p_j) > s_i(p_{j+1}) - s_{i+1}(p_{j+1}) \geq 0$.

Invariant 2. (equivalent to Lemma 2) p_j with $1 \leq j < j_0$ is useless w.r.t. i .

W.l.o.g. $\text{Min}'_{j^*} \leq i' < \text{Max}'_1$, we set Min'_{j^*-1} to i' ; point $\text{first}(\text{Min}'_{j^*-1})$ to p_1 ; and for all $1 \leq j < j_0$, point $\text{Min}(p_j)$ to Min'_{j^*-1} . Thus, each p_j satisfies $C_2(i, p_j) = \sigma_2(\llbracket \text{Min}(p_j), \text{Max}(p_j) \rrbracket)$. It is straightforward to maintain this fact until the end of step i , and the inductive hypothesis for the next step holds. Finally, **Detected** is defined as a list of pointers to $p_1 < \dots < p_{j_0-1}$. Now, the only member of **Potential** where δ_i can be negative that is not pointed by **Detected** is $p_{j_1} = \text{last}(\text{Min}'_{j^*-1})$. Thus, if $\delta_i(p_{j_1}) < 0$, we add a pointer to p_{j_1} to the end of **Detected**. The running time is $\mathcal{O}(j_0 + j^*) = \mathcal{O}(j_0) = \mathcal{O}(|\text{Detected}|)$.

Result 2. Right-strong intervals of two permutations computing time is $\mathcal{O}(n)$.

Remark 1. Ideally, at each step i , **Potential** would contain *only* the right boundaries $r \geq i$ of all unvisited right-strong intervals. Is it true ?

2.4 Common Interval Decomposition of d Permutations

After the right-strong intervals computation, a symmetric sweep from left-to-right generates the strong common intervals. We recall that those are the nodes of the decomposition tree. Moreover, the sweep organises them by interval inclusion. Hence, constructing the tree is in $\mathcal{O}(n)$ time. Then, the labelling can use the following remarks. Since there are only *Prime* and *Linear* nodes, the strong common intervals that are marked *Eaten* by the right-strong intervals computation also have this mark in the left-strong intervals computation. Besides, a node has *Eaten* if and only if it is *Linear*. Finally, Property 2, Theorem 2, and Lemma 2 can be generalised to the case of d permutations if one replaces $C_2(i, p_j)$ with $C_j = \mathcal{S}_{\sigma(\llbracket i, p_j \rrbracket)} \uplus \sigma(\llbracket i, p_j \rrbracket) = \cup_{h=2}^d C_h(i, p_j)$. Then, at each step i in the new Update-Detect, one has to maintain C_j rather than just $C_2(i, p_j)$. The hitch lays on the fact that $C_h(i, p_j)$ ($2 \leq h \leq d$) are not pairwise disjunctive. However, as an element $\sigma(i')$ can be added to some $C_h(i'', p'_j)$ only once throughout the computation, the total maintenance can be done in $\mathcal{O}(d \times n)$ time.

Result 3. *The common interval decomposing time of d permutations is $\mathcal{O}(d \times n)$.*

3 Modular Decomposition

Let $G = (V, E)$ be a loopless simple undirected graph with $n = |V|$ and $m = |E|$. A vertex $v \in V \setminus X$ exterior to $X \neq \emptyset$ is *adjacent to X* if it is adjacent to each vertex of X , *non-adjacent to X* if non-adjacent to each vertex of X . In both cases, v is *uniform to X* . Otherwise, v is a *splitter of X* . X is a *module* if it has no splitters. Roughly, the family \mathcal{M} of modules of G refers to the set of subgraphs of G that behave as one single vertex. It is well-known that \mathcal{M} is partitive [8, 22], and finding efficient algorithms for computing $\mathcal{T}_{\mathcal{M}}$ from G has been an important challenge of the last two decades [7, 8, 9, 12, 15, 21, 22]. The *factorising permutations of G* refer to the ones of \mathcal{M} . Linear time algorithms for obtaining one such permutation are available for chordal graphs [18], inheritance graphs [16], and even for arbitrary graphs [15]. The decomposition approach conducted by C. Capelle is as follows. First find a factorising permutation [15], then construct the modular decomposition tree [7]. Both computations run in $\mathcal{O}(n + m)$ time even if the latter [7] is somewhat heavily fathered.

Theorem 3. [12] *Both permutations of any realiser of a permutation graph are factorising for the graph.*

Corollary 2. *The family CI of common intervals of two permutations is included in the family \mathcal{M} of modules of the permutation graph that realises them. Besides, CI and \mathcal{M} share the same strong subsets, namely $\mathcal{S}_{CI} = \mathcal{S}_{\mathcal{M}}$. Finally, \mathcal{I}_{CI} is isomorphic to $\mathcal{T}_{\mathcal{M}}$, with Degenerate labels replaced by Linear ones.*

From Corollary 2, the algorithm of the previous section is an $\mathcal{O}(n)$ time modular decomposition algorithm for a permutation graph given by one realiser, yet $m = \theta(n^2)$. In this section, given a factorising permutation σ , we compute the modular decomposition tree of G . Let us first adapt Property 2 and Theorem 2.

3.1 Submodularity on the Size of the Splitter Sets

Let \mathcal{S}_X refer to the splitter set of a vertex subset $X \neq \emptyset$ and $s(X) = |\mathcal{S}_X|$ count the number of its splitters. We extend $s(\emptyset) = -n$. Property 3 and Corollary 3 below are our graph versions of respectively Property 2 and Theorem 2.

Property 3. $\sigma(\llbracket i, j \rrbracket)$ is a module if and only if $s_i(j) = s(\sigma(\llbracket i, j \rrbracket)) = 0$.

Definition 3 (Submodularity). (see e.g. [24]) A set function $\mu : 2^V \rightarrow \mathbb{R}$ is submodular when $\mu(X) + \mu(Y) \geq \mu(X \cup Y) + \mu(X \cap Y)$, for all $X, Y \subseteq V$.

Theorem 4 (Submodularity). The function s counting the splitters of modules of a graph is submodular.

Proof. Since $s(\emptyset) = -n$ and $s(X) \leq n$ for $X \subseteq V$ non-empty, the only tricky issue consists of proving the submodular inequality for a pair (X, Y) of overlapping subsets of V . To do this, we first notice that $\mathcal{S}_{X \cap Y} = (\mathcal{S}_{X \cap Y} \setminus Y, \mathcal{S}_{X \cap Y} \cap Y)$. Besides, $\mathcal{S}_{X \cup Y} = (\mathcal{S}_{X \cup Y} \setminus \mathcal{S}_X, \mathcal{S}_{X \cup Y} \cap \mathcal{S}_X)$ can be reduced by definition of splitters to $\mathcal{S}_{X \cup Y} = (\mathcal{S}_{X \cup Y} \setminus \mathcal{S}_X, \mathcal{S}_X \setminus (X \cup Y))$. Similarly, $\mathcal{S}_Y = (\mathcal{S}_Y \setminus \mathcal{S}_{X \cap Y}, \mathcal{S}_{X \cap Y} \setminus Y)$. Finally, $\mathcal{S}_X = (\mathcal{S}_X \setminus Y, (\mathcal{S}_X \cap Y) \setminus \mathcal{S}_{X \cap Y}, (\mathcal{S}_X \cap Y) \cap \mathcal{S}_{X \cap Y})$ can be reduced to $\mathcal{S}_X = (\mathcal{S}_X \setminus (X \cup Y), (\mathcal{S}_X \cap Y) \setminus \mathcal{S}_{X \cap Y}, \mathcal{S}_{X \cap Y} \cap Y)$. Hence, $|\mathcal{S}_X| + |\mathcal{S}_Y| - |\mathcal{S}_{X \cup Y}| - |\mathcal{S}_{X \cap Y}| = |(\mathcal{S}_X \cap Y) \setminus \mathcal{S}_{X \cap Y}| + |\mathcal{S}_Y \setminus \mathcal{S}_{X \cap Y}| - |\mathcal{S}_{X \cup Y} \setminus \mathcal{S}_X|$.

To achieve proving the lemma, we prove that $\mathcal{S}_Y \setminus \mathcal{S}_{X \cap Y} \supseteq \mathcal{S}_{X \cup Y} \setminus \mathcal{S}_X$. Indeed, let $v \in \mathcal{S}_{X \cup Y} \setminus \mathcal{S}_X$. Then, v is exterior to X and is uniform to X . By symmetry, we suppose w.l.o.g. that $v \in N_X$. Since $X \cap Y \neq \emptyset$, there exists $w \in X \cap Y$ with $(v, w) \in E$. Now, v is a splitter of $X \cup Y$, implying $u \in Y \setminus X$ such that $(v, u) \notin E$. Hence, $v \in \mathcal{S}_Y$. It is trivial by $v \in N_X$ that $v \notin \mathcal{S}_{X \cap Y}$. \square

Corollary 3. Let $i \leq p_j < p_{j+1}$, and $\delta_i(p_j) = s_i(p_{j+1}) - s_i(p_j)$. Then, $\delta_i(p_j) < 0$ implies there is no $k \leq i$ such that $\sigma(\llbracket k, p_j \rrbracket)$ is a module.

Proof. If $\delta_i(p_j) < 0$, then the submodularity on the subsets $\sigma(\llbracket k, p_j \rrbracket)$ and $\sigma(\llbracket i, p_{j+1} \rrbracket)$ for all $k \leq i$ implies that $s_k(p_j) > s_k(p_{j+1}) \geq 0$. \square

3.2 Modular Decomposition Algorithm

The two latter generalisations state that Section 2.4's decomposition scheme can be used in the case of modules if one adapts the involved routines accordingly. Actually, the adaptation is straightforward, except for the Update-Detect routine and labelling the decomposition tree. For lake of space, we do not detail here the graph version of Update-Detect, which computes in $\mathcal{O}(n + m)$ global time. Now, let us show how to label the decomposition tree. First, it is well-known that a modular decomposition tree has no *Linear* nodes, and its *Degenerate* nodes are divided into *Serial* nodes – adjacency guaranteed between all children – and *Parallel* nodes – non-adjacency between children [8, 22]. Then, by analogous remarks as in Section 2.4, nodes marked *Eaten* are *Degenerate*, others are *Prime*. Besides, thanks to some *Adjacency* marks in the graph version of Update-Detect, which state the adjacency between the children of the node, we can differ the *Serial* and *Parallel* nodes in the labelling.

Result 4. *The modular decomposition is solved in $\mathcal{O}(n+m)$ time for any graph.*

There exists an $\mathcal{O}(n)$ time common interval decomposition algorithm of two permutations [20]. Unfortunately, the algorithm therein is not that simple and relies on a rather sophisticated algorithm [10]. Moreover their approach is not extended to general modular decomposition. To this aim one could use the algorithm proposed in [7]. However, this latter produces a rather heavy sequence of trees. On the other hand, our approach uses a unique paradigm for both computations of common interval and modular decomposition tree, and not only we unify the two corresponding domains but also provide very efficient algorithms.

4 Conclusion and Perspectives

We show the importance of graph layout approaches, e.g. with factorising permutations, which are based on a gateway between algorithms on permutations and those on graphs. Besides, we show strong potentials of generalising T. Uno and M. Yagiura’s algorithm to the case of weakly partitive families. Thus, the use of this algorithm would be an important crux for designing future algorithms. For instance, it would be interesting to adopt the same philosophy conducted throughout our paper to other combinatorial problems such as decomposition into “inheritance-block” of an inheritance graph in $\mathcal{O}(n+m)$ time, which would yield an alternative to the algorithms proposed in [6, 16]. Another example would be the modular decomposition in $\mathcal{O}(n)$ time of a bounded tolerance graph – trapezoid graph with solely parallelograms [5, 13] – when an intersection model is provided. Then, it would be very interesting to have an $\mathcal{O}(n)$ modular decomposition time for an interval or trapezoid graph on one of its intersection model, which would give interesting links to works on gene-teams [1].

Acknowledgements. We are grateful to T. Uno for helpful discussions.

References

1. M.-P. Béal, A. Bergeron, S. Corteel, and M. Raffinot. An algorithmic view of gene teams. *Theoretical Computer Science*, 320(2-3):395–418, 2004.
2. S. Bérard, A. Bergeron, and C. Chauve. Conservation of combinatorial structures in evolution scenarios. In *International Workshop on Comparative Genomics (RECOMB04)*, volume 3388 of *LNCS*, pages 1–14, 2004.
3. A. Bergeron, C. Chauve, F. de Montgolfier, and M. Raffinot. Computing common intervals of k permutations, with applications to modular decomposition of graphs. In *13th Annual European Symposium on Algorithms (ESA05)*, 2005. to appear in *LNCS*.
4. A. Bergeron and J. Stoye. On the similarity of sets of permutations and its applications to genome comparison. In *9th Annual International Conference on Computing and Combinatorics (COCOON03)*, volume 2697 of *LNCS*, pages 68–79, 2003.
5. K. P. Bogart, P. C. Fishburn, G. Isaak, and L. Langley. Proper and unit tolerance graphs. *Discrete Applied Mathematics*, 60:99–117, 1995.

6. C. Capelle. Block decomposition of inheritance hierarchies. In *23rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG97)*, volume 1335 of *LNCS*, pages 118–131, 1997.
7. C. Capelle, M. Habib, and F. de Montgolfier. Graph decomposition and factorizing permutations. *Discrete Mathematics and Theoretical Computer Science*, 5(1):55–70, 2002.
8. M. Chein, M. Habib, and M.C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, 37(1):35–50, 1981.
9. A. Cournier and M. Habib. A new linear algorithm for modular decomposition. In *Trees in algebra and programming (CAAP 94)*, volume 787 of *LNCS*, pages 68–84, 1994.
10. E. Dahlhaus. Parallel algorithms for hierarchical clustering, and applications to split decomposition and parity graph recognition. *Journal of Algorithms*, 36(2):205–240, 2000.
11. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry*. Springer-Verlag, 1991.
12. F. de Montgolfier. *Décomposition modulaire des graphes. Théorie, extensions et algorithmes*. PhD thesis, Université Montpellier II, 2003.
13. S. Felsner. Tolerance graphs and orders. *Journal of Graph Theory*, 28(3):129–140, 1998.
14. M. Figeac and J.-S. Varré. Sorting by reversals with common intervals. In *4th International Workshop on Algorithms in Bioinformatics (WABI04)*, volume 3240 of *LNBI*, pages 26–37, 2004.
15. M. Habib, F. de Montgolfier, and C. Paul. A simple linear-time modular decomposition algorithm. In *9th Scandinavian Workshop on Algorithm Theory (SWAT04)*, volume 3111 of *LNCS*, pages 187–198, 2004.
16. M. Habib, M. Huchard, and J.P. Spinrad. A linear algorithm to decompose inheritance graphs into modules. *Algorithmica*, 13(6):573–591, 1995.
17. S. Heber and J. Stoye. Finding all common intervals of k permutations. In *12th Annual Symposium on Combinatorial Pattern Matching (CPM01)*, volume 2089 of *LNCS*, pages 207–218, 2001.
18. W.-L. Hsu and T.-M. Ma. Substitution decomposition on chordal graphs and applications. In *2nd International Symposium on Algorithms (ISA91)*, volume 557 of *LNCS*, pages 52–60, 1991.
19. G. M. Landau, L. Parida, and O. Weimann. Using pq trees for comparative genomics. In *16th Annual Symposium on Combinatorial Pattern Matching (CPM05)*, volume 3537 of *LNCS*, 2005.
20. R. M. McConnell and F. de Montgolfier. Algebraic Operations on PQ Trees and Modular Decomposition Trees. In *31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG05)*, 2005. to appear in *LNCS*.
21. R.M. McConnell and J.P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201:189–241, 1999.
22. R.H. Möhring and F.J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *Annals of Discrete Mathematics*, 19:257–356, 1984.
23. T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.
24. D. J. A. Welsh. Matroids: Fundamental concepts. In *Handbook of Combinatorics*, volume 1, pages 481–526. North-Holland, 1995.