

Finding Nested Common Intervals Efficiently

Guillaume Blin* David Faye† Jens Stoye‡

Abstract

In this paper, we study the problem of efficiently finding gene clusters formalized by nested common intervals between two genomes represented either as permutations or as sequences. Considering permutations, we give several algorithms whose running time depends on the size of the actual output rather than the output in the worst case. Indeed, we first provide a straightforward cubic time algorithm for finding all nested common intervals. We reduce this complexity by providing a quadratic time algorithm computing an irredundant output. We then show, by providing a third algorithm, that finding only the maximal nested common intervals can be done in linear time. Finally, we prove that finding approximate nested common intervals is fixed parameter tractable. Considering sequences, we provide solutions (modifications of previously defined algorithms and a new algorithm) for different variants of the problem, depending on the treatment one wants to apply to duplicated genes. This includes a polynomial-time algorithm for a variant implying a matching of the genes in the cluster, a setting that for other problems often leads to hardness.

*Université Paris-Est, LIGM - UMR CNRS 8049, France, gblin@univ-mlv.fr

†Université Paris-Est, LIGM - UMR CNRS 8049, France, Université Gaston-Berger, LANI, Saint-Louis, Senegal, dfaye@igm.univ-mlv.fr

‡Technische Fakultät, Universität Bielefeld, Germany, stoye@techfak.uni-bielefeld.de

1 Introduction and related work

Computational comparative genomics is an active field of bioinformatics. One of the problems arising in this domain consists in comparing two or more species by seeking for *gene clusters* between their genomes. A gene cluster refers to a set of genes appearing, in spatial proximity along the chromosome, in at least two genomes. Genomes evolved from a common ancestor tend to share some gene clusters. Therefore, they may be used for reconstructing recent evolutionary history and inferring putative functional assignments for genes.

The genome evolution process, including – among others – fundamental evolutionary events such as gene duplication and loss (Ohno, 1970), has given rise to various genome models and cluster definitions. Indeed, genomes may be either represented as *permutations* (allowing one-to-one correspondence between genes of different genomes) or *sequences* – where the same letter (i.e. gene) may occur more than once (a more realistic model but with higher complexity). In both those models, there may exist, or not, genes not shared between two genomes (often called *gaps*).

Moreover, when modeling genomes for gene order analysis, one may consider either *two* or *multiple* genomes, seeking for *exact* or *approximate* occurrences, finding *all* or just non-extensible (i.e. *maximal*) occurrences.

There are numerous ways of mathematical formalizations of gene clusters. Among others, one can mention *common substrings* (which require a full conservation), *common intervals* (Uno and Yagiura, 2000; Schmidt and Stoye, 2004; Didier et al., 2007; Bergeron et al., 2008a) (genes must occur consecutively, regardless of their order), *conserved intervals* (Bergeron and Stoye, 2006; Angibaud et al., 2009) (common intervals, framed by the same two genes), *gene teams* (Bergeron et al., 2002; He and Goldwasser, 2005; Zhang and Leong, 2009) (genes in a

cluster must not be interrupted by long stretches of genes not belonging to the cluster), and *approximate common intervals* (Rahmann and Klau, 2006; Böcker et al., 2009) (common intervals that may contain few genes from outside the cluster). For more details, please refer to (Bergeron et al., 2008b).

In this article, we focus on another model – namely the *nested common intervals* – which was mentioned in (Hoberman and Durand, 2005). In this model, an additional constraint – namely the *nestedness* – (observed in real data (Kurzik-Dumke and Zengerle, 1996)) is added to the cluster definition. Hoberman and Durand (Hoberman and Durand, 2005) argued that, depending on the dataset, if the nestedness assumption is not excluding clusters from the data, then it can strengthen the significance of detected clusters since it reduces the probability of observing them by chance.

As far as we know, (Hoberman and Durand, 2005) was the only attempt to take into account the nestedness assumption in a gene cluster model (namely gene teams) and yields to a quadratic-time greedy bottom-up algorithm. In the following, we will give some efficient algorithms to find all nested common intervals, for different definitions of such intervals, between two genomes.

After providing basic definitions (Section 2), we will provide in Section 3 complexity results for the problems of computing irredundant and maximal nested common intervals in output-optimal time. In Section 4, we will provide algorithms to handle genomes represented as sequences considering three different variants of the problem. Finally, in Section 5, we will consider a natural extension of the problem, namely allowing gaps in the definition of nested common intervals.

2 Definitions and previous works

Let π_1 and π_2 be our genomes, represented as permutations over $N := \{1, \dots, n\}$. For any $i \leq j$, $\pi[i, j]$ will refer to the sequence of elements $(\pi[i], \pi[i+1], \dots, \pi[j])$. Let $\mathcal{CS}(\pi[i, j]) := \{\pi[k] \mid k \in [i, j]\}$ denote the *character set* of the interval $[i, j]$ of π . A subset $C \subseteq N$ is called a *common interval* of π_1 and π_2 if and only if there exist $1 \leq i_1 < j_1 \leq n$ and $1 \leq i_2 < j_2 \leq n$ such that $C = \mathcal{CS}(\pi_1[i_1, j_1]) = \mathcal{CS}(\pi_2[i_2, j_2])$. Note that this definition purposely excludes common intervals of size one since they would not be considered in the more general nested common interval definition. The intervals $[i_1, j_1]$ and $[i_2, j_2]$ are called the *locations* of C in π_1 and π_2 , respectively.

Note also that in case of multiple gene copies (i.e. considering sequences instead of permutations), these definitions will be slightly different, as detailed in the beginning of Section 4.

Given two common intervals C and C' of π_1 and π_2 , C *contains* C' if and only if $C' \subseteq C$. This implies that the location of C' in π_1 (resp. π_2) is included in the location of C in π_1 (resp. π_2).

Definition 1 *A common interval C is called a nested common interval of π_1 and π_2 if either $|C| = 2$, or if $|C| > 2$ and it contains a nested common interval of size $|C| - 1$.*

Note that this recursive definition implies that for any nested common interval C there exists a series of nested common intervals such that $C_2 \subseteq C_3 \subseteq \dots \subseteq C$ with $|C_i| = i$.

Definition 2 *A nested common interval of size ℓ is maximal if it is not contained in a nested common interval of size $\ell + 1$.*

A maximal nested common interval can however still be contained in a larger nested

common interval. For example, considering $\pi_1 := (3, 1, 2, 4, 5, 6)$ and $\pi_2 := (1, 2, 3, 4, 5, 6)$, the maximal nested common interval $[4, 6]$ in π_1 is contained in $[1, 6]$.

The general NESTED COMMON INTERVALS problem may be defined as follows: *Given two genomes, find all their nested common intervals.* One can then consider genomes either as permutations or sequences and might also be interested in finding only the maximal nested common intervals and/or allowing gaps. In the two following sections, we will give efficient algorithms for both permutations and sequences without considering gaps. We will, then, provide FPT solutions consisting of an extension of previously defined algorithms for finding maximal nested common intervals considering gaps in both permutations and sequences (except for the last variant that we leave as an open problem).

In (Bergeron et al., 2008a), a theoretical framework was introduced for computing common intervals based on a linear space basis. Of importance here is the technique proposed in order to generate the PQ-tree (Booth and Lueker, 1976; Landau et al., 2005) corresponding to a linear space basis for computing all the common intervals of K permutations. Generating this basis can be done in $O(n)$ time for two permutations of size n . Then one can, by a browsing of the tree, generate all the common intervals in $O(n + z)$ time where z is the size of the output. One can adapt this algorithm in order to find nested common intervals in $O(n + z)$ time.

In this work, we do not follow that approach, but instead provide algorithms for a more direct detection of the nested common intervals.

3 Nested common intervals on permutations

As described in (Uno and Yagiura, 2000; Schmidt and Stoye, 2004; Didier et al., 2007), when considering permutations, both common substrings and common intervals can be found in optimal, essentially linear time. As we will show, not surprisingly, finding nested common intervals on permutations can also be done efficiently.

3.1 Finding all nested common intervals

First, one has to notice that the number of nested common intervals can be quadratic in n (e.g. when $\pi_1 = \pi_2$). However, in many practical cases the number of nested common intervals may be much smaller, such that one can still achieve lower time complexity by developing methods whose running time depends on the size of the actual output and not of the output in the worst case.

In the following, we will w.l.o.g. assume that π_1 is the identity permutation and rename π_2 by π for ease of notation. A naive bottom-up algorithm, inspired from the one given in (Hoberman and Durand, 2005) and straightforwardly following the definition of nested common intervals, can be defined as in Algorithm 1.

Algorithm 1 uses a bottom-up approach; namely it tries to compute any nested common interval from smaller ones by an *extension* procedure. More precisely, starting from a nested common interval of size one, the algorithm tries to extend this last interval both to the left and to the right by iteratively trying to add the left (resp. right) neighbor if it still induces a common interval (*i.e.* $l--$, $l++$, $r--$, $r++$).

Clearly such an algorithm requires $O(n + z)$ time to report all nested common intervals where z is the size of the output. However, the output may be highly redundant as several

Algorithm 1 Find all nested common intervals

```
1: for  $i \leftarrow 1, \dots, n$  do
2:    $l \leftarrow i, r \leftarrow i$ 
3:   repeat
4:      $l' \leftarrow l, r' \leftarrow r$ 
5:     if  $\pi[l' - 1] = \min(\mathcal{CS}(\pi[l', r'])) - 1$  or  $\pi[r' + 1] = \max(\mathcal{CS}(\pi[l', r'])) + 1$  then
6:       while  $\pi[l - 1] = \min(\mathcal{CS}(\pi[l, r'])) - 1$  do  $l--$  done
7:       while  $\pi[r + 1] = \max(\mathcal{CS}(\pi[l', r])) + 1$  do  $r++$  done
8:     else
9:       while  $\pi[l - 1] = \max(\mathcal{CS}(\pi[l, r'])) + 1$  do  $l--$  done
10:      while  $\pi[r + 1] = \min(\mathcal{CS}(\pi[l', r])) - 1$  do  $r++$  done
11:      end if
12:      report all intervals  $[l'', r'']$  with  $l \leq l'' \leq l'$  and  $r' \leq r'' \leq r$  except  $[l', r']$ 
13:    until  $l = l'$  and  $r = r'$ 
14:  end for
```

intervals will be identified more than once. The worst case is when one considers $\pi = (1, 2, \dots, n)$. More precisely, in this case some of the $O(n^2)$ nested common intervals will be reported up to n times, giving a total worst-case runtime of $O(n^3)$.

Therefore, one may be interested in computing an irredundant output. The main improvement we propose consists in a simple preprocessing step that will speed up our algorithm for nested common intervals. Let us define a *run* of two permutations π_1 and π_2 as a pair of intervals $([i_1, j_1], [i_2, j_2])$ such that $\pi_1[i_1, j_1] = \pi_2[i_2, j_2]$ or $\pi_1[i_1, j_1] = \overleftarrow{\pi_2[i_2, j_2]}$ where $\overleftarrow{x} := (x_k, x_{k-1}, \dots, x_1)$ denotes the reverse of sequence $x = (x_1, x_2, \dots, x_k)$. A run is *maximal* if it cannot be extended to the left or right. Since a run can also be of size one, two permutations can always be decomposed into their maximal runs with respect to each other. For example, in the following the maximal runs are underlined: $\pi_1 = (1, 2, 3, 4, \underline{5}, \underline{6}, \underline{7}, \underline{8}, \underline{9})$ and $\pi_2 = (\underline{4}, \underline{3}, \underline{2}, \underline{1}, \underline{5}, \underline{9}, \underline{6}, \underline{7}, \underline{8})$.

Given a decomposition of two permutations into their maximal runs with respect to each other, a *breakpoint* will refer to any pair of neighboring elements that belong to different runs. In the above example, π_1 (resp. π_2) contains three breakpoints $(4, 5)$, $(5, 6)$ and $(8, 9)$

(resp. $(1, 5)$, $(5, 9)$ and $(9, 6)$). By definition, the number of breakpoints is one less than the number of runs. When considering one of π_1 and π_2 as being the identity permutation then a run may be defined as a single interval.

Algorithm 2, hereafter defined, computes irredundant output by making use of the two following simple observations. In fact, it tries to extend the actual nested common interval by more than one position at each step using the notion of run.

Lemma 1 *All subintervals of length at least 2 in a run are nested common intervals.*

Lemma 2 *In the procedure of constructing incrementally a nested common interval by extension, when one reaches up to one end of a run such that the begin/end of this run can be included, then the whole run can be included and all subintervals ending/beginning in this run can be reported as nested common intervals.*

Proof. By definition, the elements of a run $[i, j]$ in π with respect to the identity permutation are consecutive integers strictly increasing or decreasing. Therefore, in an incremental construction by extension of a nested common interval nc that has a run $[i, j]$ as its right (resp. left) neighbor, if one may extend nc by i (resp. j) then, by definition, $\pi[i]$ (resp. $\pi[j]$) is the minimal or maximal element among the elements of the extended interval. Thus, all the elements of the run $[i, j]$ may be added one by one, each leading to a new nested common interval. \square

Consequently, by identifying the runs in a preprocessing step (which can be easily done in linear time), whenever during an extension a border of a run is included, the whole run is added at once and all sub-intervals are reported. The details are given in Algorithm 2 which employs a data structure *end* defined as follows: if i is the index of the first or last element

of a run in π then $end[i]$ is the index of the other end of that run; $end[i] = 0$ otherwise.

Algorithm 2 Find all nested common intervals, irredundant version

```

1: decompose  $\pi$  into maximal runs w.r.t.  $id$  and store them in a data structure  $end$ 
2: for  $i \leftarrow 1, \dots, n - 1$  do
3:   if  $end[i] > i$  then
4:      $l \leftarrow i, r \leftarrow end[i]$ 
5:     report all intervals  $[l'', r'']$  with  $l \leq l'' < r'' \leq r$ 
6:     repeat
7:        $l' \leftarrow l, r' \leftarrow r$ 
8:       if  $\pi[l'-1] = \min(\mathcal{CS}(\pi[l', r'])) - 1$  or  $\pi[r'+1] = \max(\mathcal{CS}(\pi[l', r'])) + 1$  then
9:         if  $\pi[l-1] = \min(\mathcal{CS}(\pi[l, r'])) - 1$  then  $l \leftarrow end[l-1]$  end if
10:        if  $\pi[r+1] = \max(\mathcal{CS}(\pi[l', r])) + 1$  then  $r \leftarrow end[r+1]$  end if
11:        else
12:          if  $\pi[l-1] = \max(\mathcal{CS}(\pi[l, r'])) + 1$  then  $l \leftarrow end[l-1]$  end if
13:          if  $\pi[r+1] = \min(\mathcal{CS}(\pi[l', r])) - 1$  then  $r \leftarrow end[r+1]$  end if
14:          end if
15:          report all intervals  $[l'', r'']$  with  $l \leq l'' \leq l'$  and  $r' \leq r'' \leq r$  except  $[l', r']$ 
16:        until  $l = l'$  and  $r = r'$ 
17:    end if
18: end for

```

Theorem 1 *Algorithm 2 has an $O(n^2)$ time complexity.*

Proof. It is easily seen that the time complexity remains in $O(n+z)$ with z being the output size, except that this time the output is irredundant (i.e. each nested common interval is reported exactly once). Since the number of nested common intervals is at most quadratic, the complexity follows. \square

When applied to $\pi := (1, 2, 3, 6, 4, 5)$ for example, one may check that Algorithm 2 will report locations $[1, 2], [1, 3], [2, 3]$ when $i = 1$, locations $[5, 6], [4, 6], [1, 6], [2, 6], [3, 6]$ when $i = 5$, and nothing for the other values of i . Let us prove this interesting property of Algorithm 2.

Lemma 3 *In Algorithm 2, any breakpoint is considered at most once during the extension procedure described from lines 8 to 14.*

Proof. In the following, a breakpoint will be *passed through* from left (resp. from right) when considered during an extension procedure to the right (resp. left). Assume $bp = (\pi[x], \pi[y])$ is a breakpoint in π with respect to the identity permutation. Then $y = x + 1$ and there are only two possibilities for bp to be passed through twice: either (1) once from the left and once from the right, or (2) twice from the same side.

Let us first consider the case where bp is passed through from both left and right. Therefore assume that

$$\pi = (\dots, \pi[X], \dots, \pi[x], \pi[y], \dots, \pi[Y], \dots)$$

where $C_1 := \mathcal{CS}(\pi[X, x])$ and $C_2 := \mathcal{CS}(\pi[y, Y])$ are nested common intervals (otherwise bp would not be passed through more than once). Further assume w.l.o.g. that bp is passed in the extension of interval $[X, x]$ (a similar proof handles the extension of $[y, Y]$) whose maximum (or minimum) element is $M := \max(C_1)$ (resp. $m := \min(C_1)$).

Then, in order for $[X, x]$ to be extensible, either $\pi[y] = M + 1$ or $\pi[y] = m - 1$. Let us assume that $\pi[y] = M + 1$ (the case where $\pi[y] = m - 1$ can be shown similarly). We will show that bp cannot be passed through in the other direction, i.e. in an extension of $[y, Y]$. Since $\pi[y] = M + 1$ and each of the two intervals $[X, x]$ and $[y, Y]$ consists of consecutive integers, we have that all elements in C_1 are smaller than any element in C_2 . Thus, for an extension in the left direction across bp , the largest element of C_1 must be at its right end, i.e. $\pi[x] = M$. However, if this was the case, then $bp = (\pi[x], \pi[y]) = (M, M + 1)$ would not be a breakpoint, a contradiction.

Now let us consider the case where bp is passed through twice from the same side, starting

with different runs. Therefore assume that

$$\pi = (\dots, \pi[X'], \dots, \pi[X], \dots, \pi[x], \pi[y], \dots)$$

where $C_1 := \mathcal{CS}(\pi[X, x])$ and $C_2 := \mathcal{CS}(\pi[X', x])$ are nested common intervals derived from different runs (i.e. reported from two different values of i), one in the interval $[X, x]$ and the other in the interval $[X', X - 1]$. Further assume w.l.o.g. that bp is passed through in the extension of $[X, x]$ (a similar proof handles the extension of $[X', x]$) whose maximum (or minimum) element is $M := \max(C_1)$ (resp. $m := \min(C_1)$).

Then, in order for $[X, x]$ to be extensible, either $\pi[y] = M + 1$ or $\pi[y] = m - 1$. Let us assume that $\pi[y] = M + 1$ (the case where $\pi[y] = m - 1$ can be shown similarly). We will show that bp cannot be passed again in this direction, i.e. in an extension of $[X', x]$. Since $\pi[y] = M + 1$ and, by construction, $C_1 \subset C_2$, we have that all elements in $\mathcal{CS}(\pi[X', X - 1])$ are smaller than any element in C_1 . Moreover, since bp is a breakpoint, we have that $\pi[x] \neq M$ and, more precisely, $\pi[x] < M$. Then, any extension of $[X', X - 1]$ would not be able to include M since at least one necessary intermediate element (namely $\pi[x]$) would not have been previously included. Thus, all cases are covered and the lemma is proved. \square

Irredundancy of the locations of nested common intervals returned by Algorithm 2 follows immediately.

Lemma 4 *In Algorithm 2, starting from two different runs (cf. line 4) cannot yield to reporting the same nested common interval twice.*

Proof. In order to be possibly reported twice, an interval would have to cover (i.e. include both) two different runs. However, in order to yield the interval, the breakpoint(s) between

these two runs would have to be passed through twice, which is not possible by Lemma 3.

□

3.2 Finding all maximal nested common intervals

As previously mentioned, one might also be interested in finding only the maximal nested common intervals in optimal time $O(n+z)$ where z is the number of maximal nested common intervals of π_1 and π_2 , since there will be fewer. In fact, we will first prove that the number of maximal nested common intervals is in $O(n)$ leading to an overall linear time algorithm.

Lemma 5 *Every element of π is contained in at most three different maximal nested common intervals.*

Proof. This follows immediately from the correctness of Lemma 3. Indeed, according to Lemma 3 each position can be reached from at most two directions. Thus, the only case where an element of π may be contained in exactly three different maximal nested common intervals $nc_1 = [i_1, j_1]$, $nc_2 = [i_2, j_2]$ and $nc_3 = [i_3, j_3]$ is when $i_1 \leq i_2 \leq i_3 \leq j_1 \leq j_2 \leq j_3$. For example, considering $\pi_1 = (2, 1, 3, 4, 6, 5)$ and $\pi_2 = (1, 2, 3, 4, 5, 6)$, the element 3 in π_1 is contained in $(2, 1, 3, 4)$, $(3, 4)$ and $(3, 4, 6, 5)$. □

In order to get only the locations of maximal nested common intervals, one has to modify Algorithm 2 such that only the locations at the end of an extension are reported. To do so, one has simply to (1) remove from Algorithm 2 lines 5 and 15 and (2) report the unique interval $[l, r]$ – which is by definition maximal – just after the end of the **repeat ... until** loop (currently line 16).

Clearly, the time complexity of this slightly modified version of Algorithm 2 is unchanged; that is $O(n + z)$ where z is the size of the output. Lemma 5 implies that the number of maximal nested common intervals is in $O(n)$, leading to an overall linear time.

4 Nested common intervals on sequences

In this section, we will give algorithms to handle genomes represented as sequences (i.e. genes may be duplicated). In the following, we will assume that our genomes, denoted by S_1 and S_2 , are defined over a bounded integer alphabet $\Sigma = \{1, \dots, \sigma\}$ and have maximal length n . The precise definition of nestedness in sequences is subtle. Therefore, we propose three different variants of the problem, depending on the treatment one wants to apply when, during the extension of an interval, an element that is already inside the interval is met once again. The definitions and the differences are illustrated by the running example of two sequences $S_1 = (3, 1, 2, 3, 4, 5, 6)$ and $S_2 = (2, 3, 1, 3, 2, 4, 6, 3, 5)$, see also Figure 1.

First, one may just extend the interval “for free”, only caring about the “innermost occurrence”; all other occurrences are considered as not contributing to the cluster content. This definition follows the same logic as earlier ones used for common intervals (Schmidt and Stoye, 2004; Didier et al., 2007) and for approximate common intervals (Böcker et al., 2009). In our example, starting from the left in S_1 with the characters 3 and 1 will immediately yield three different locations of the character set $C_2 = \{1, 3\}$: $([1, 2], [2, 3])$, $([1, 2], [2, 4])$ and $([1, 2], [3, 4])$. Adding the next character in S_1 , 2, will also allow to add the following 3 (at index 4) since a 3 already belonged to the cluster. Combined with the five possible intervals in S_2 that can be obtained by extending the three intervals containing $\{1, 3\}$, this yields ten cluster locations for $C_3 = \{1, 2, 3\}$: $([1, 3], [1, 3])$, $([1, 3], [1, 4])$, $([1, 3], [1, 5])$, $([1, 3], [2, 5])$, $([1, 3], [3, 5])$, $([1, 4], [1, 3])$, $([1, 4], [1, 4])$, $([1, 4], [1, 5])$, $([1, 4], [2, 5])$, $([1, 4], [3, 5])$.

A slight modification of our naive Algorithm 1 leads to an $O(n^3)$ algorithm. Indeed, since the sequences may contain duplicates, one has to start the extension procedure with all possible pairs $(S_1[i], S_2[j])$ where $1 \leq i \leq |S_1|$ and $1 \leq j \leq |S_2|$. Moreover, after each

extension step all genes that are already members of the cluster have to be “freely” included. This can be tested efficiently by storing the elements belonging to the current cluster in a bit vector $c[1, \dots, \sigma]$.

The resulting Algorithm 3 – which clearly runs in $O(n^3)$ time as each pair of index positions (i, j) is considered at most once and for each of them the extension cannot include more than n steps – only reports maximal gene clusters as previously done for permutations.

Second, one may forbid the inclusion of a second copy of a gene in a nested common interval. In our example, for $C_2 = \{1, 3\}$ at indices 1 and 2 in S_1 this would give only the two locations $([1, 2], [2, 3])$ and $([1, 2], [3, 4])$, and the extension by character 2 would be allowed in only one of the two possible directions, giving for $C_3 = \{1, 2, 3\}$ the locations $([1, 3], [1, 3])$ and $([1, 3], [3, 5])$. Note, however, that if one starts with the intervals $[2, 3]$ or $[3, 4]$ in S_1 , more locations can be obtained for $C_3 = \{1, 2, 3\}$.

This problem variant can also be solved easily, by a quite similar algorithm which stops any extension when a gene already contained in the common interval is encountered.

Finally, one may be interested in finding a bijection (sometimes called *matching* in the computational comparative genomics literature) where, inside a nested common interval, each gene occurrence in S_1 must match a unique gene occurrence in S_2 from the same gene family. In our example, of the ten locations for $C_3 = \{1, 2, 3\}$ listed above, only those six with the same cardinality of characters are eligible as nested common intervals of this type: $([1, 3], [1, 3])$, $([1, 3], [3, 5])$, $([1, 4], [1, 4])$, $([1, 4], [2, 5])$. Moreover, among the possible matchings, only those are allowed for which the necessary smaller intervals exist. For example, the pair of locations $([1, 4], [1, 4])$ as an extension of the locations $([1, 3], [1, 3])$ implies that the 3 at index 4 in S_1 is matched with the 3 at index 4 in S_2 and not with the

Algorithm 3 Find all maximal nested common intervals in two sequences

```
1: for  $i \leftarrow 1, \dots, |S_1|$  do
2:   for each occurrence  $j$  of  $S_1[i]$  in  $S_2$  do
3:     for each  $k \leftarrow 1, \dots, \sigma$  do  $c[k] \leftarrow (k = S_1[i])$  done
4:      $l_1 \leftarrow i, r_1 \leftarrow i$ 
5:      $l_2 \leftarrow j, r_2 \leftarrow j$ 
6:     repeat
7:       while  $c[S_1[l_1 - 1]] = \text{true}$  do  $l_1--$  done
8:       while  $c[S_1[r_1 + 1]] = \text{true}$  do  $r_1++$  done
9:       while  $c[S_2[l_2 - 1]] = \text{true}$  do  $l_2--$  done
10:      while  $c[S_2[r_2 + 1]] = \text{true}$  do  $r_2++$  done
11:       $l'_1 \leftarrow l_1, r'_1 \leftarrow r_1$ 
12:      if  $S_1[l_1 - 1] = S_2[l_2 - 1]$  or  $S_1[r_1 + 1] = S_2[r_2 + 1]$  then
13:        while  $S_1[l_1 - 1] = S_2[l_2 - 1]$  do
14:           $l_1--, l_2--, c[S_1[l_1]] \leftarrow \text{true}$ 
15:          while  $c[S_1[l_1 - 1]] = \text{true}$  do  $l_1--$  done
16:          while  $c[S_2[l_2 - 1]] = \text{true}$  do  $l_2--$  done
17:        end while
18:        while  $S_1[r_1 + 1] = S_2[r_2 + 1]$  do
19:           $r_1++, r_2++, c[S_1[r_1]] \leftarrow \text{true}$ 
20:          while  $c[S_1[r_1 + 1]] = \text{true}$  do  $r_1++$  done
21:          while  $c[S_2[r_2 + 1]] = \text{true}$  do  $r_2++$  done
22:        end while
23:      else
24:        while  $S_1[l_1 - 1] = S_2[r_2 + 1]$  do
25:           $l_1--, r_2++, c[S_1[l_1]] \leftarrow \text{true}$ 
26:          while  $c[S_1[l_1 - 1]] = \text{true}$  do  $l_1--$  done
27:          while  $c[S_2[r_2 + 1]] = \text{true}$  do  $r_2++$  done
28:        end while
29:        while  $S_1[r_1 + 1] = S_2[l_2 - 1]$  do
30:           $r_1++, l_2--, c[S_1[r_1]] \leftarrow \text{true}$ 
31:          while  $c[S_1[r_1 + 1]] = \text{true}$  do  $r_1++$  done
32:          while  $c[S_2[l_2 - 1]] = \text{true}$  do  $l_2--$  done
33:        end while
34:      end if
35:      until  $l_1 = l'_1$  and  $r_1 = r'_1$ 
36:      report  $([l_1, r_1], [l_2, r_2])$ 
37:    end for
38:  end for
```

one at index 2.

Surprisingly, the nestedness constraint leads to a polynomial time algorithm whereas

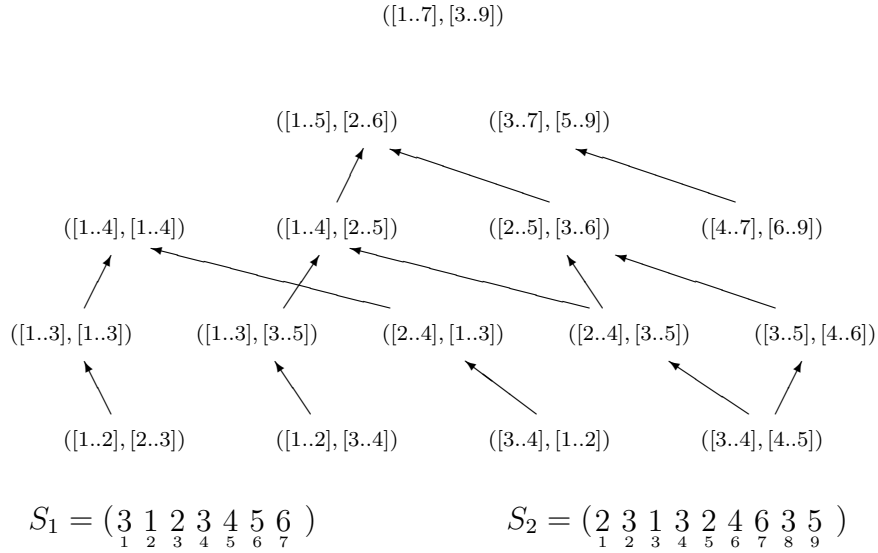


Figure 1: Graph G for sequences $S_1 = (3, 1, 2, 3, 4, 5, 6)$ and $S_2 = (2, 3, 1, 3, 2, 4, 6, 3, 5)$.

for many other paradigms, considering matching and duplicates leads to hardness (Blin et al., 2007). Unfortunately, we only know a very inefficient algorithm described hereafter. The main idea is to, first, construct a directed acyclic graph G whose vertices correspond to pairs of intervals $([i_1, j_1], [i_2, j_2])$, one from S_1 and one from S_2 , that contain the same multiset of characters. In G , an edge is drawn from a vertex $v = ([i_1, j_1], [i_2, j_2])$ to a vertex $v' = ([i'_1, j'_1], [i'_2, j'_2])$ if and only if the corresponding interval pairs differ by one in length, i.e. $|(j_1 - i_1) - (j'_1 - i'_1)| = 1$, and the shorter one is contained in the longer one, i.e. $((i_1 = i'_1)$ or $(j_1 = j'_1))$ and $((i_2 = i'_2)$ or $(j_2 = j'_2))$. An illustration is given in Figure 1.

Since, for a given multiset of cardinality ℓ there are at most $(n - \ell + 1)^2$ vertices in the graph, the total number of vertices in G is bounded by $O(n^3)$. Moreover, by definition, each vertex has an output degree of at most four, hence the number of edges is also bounded by $O(n^3)$. Finally, G can clearly be constructed in polynomial time.

One can easily see that there is a correspondence between nested gene clusters and directed paths in G starting from vertices corresponding to multisets of size 2. Indeed, a

path $(ci_1, ci_2, \dots, ci_k)$ in this DAG, where ci_1, ci_2, \dots, ci_k are common intervals, induces that $ci_1 \subseteq ci_2 \subseteq \dots \subseteq ci_k$ and for all j , $1 \leq j < k$, we have $|ci_j| + 1 = |ci_{j+1}|$. Therefore, since any common interval of size 2 is a nested common interval, in any such path, if ci_1 is of size 2 then, by definition, any common interval of this path is a nested common interval.

Thus, the nested gene clusters can be reported in polynomial time. Indeed, building the DAG can be done in $O(n^3)$ time. Then, one has to browse any path starting from a vertex corresponding to a common interval of size 2. Since there are at most $n - 1$ such common intervals, and each vertex has an output degree of at most four, on the whole the number of such paths is bounded by $(n - 1) \cdot 4(n - 2)$; i.e. $O(n^2)$.

5 Finding all maximal approximate nested common intervals

As a natural extension of the problem, we will consider allowing gaps in the definition of nested common intervals. Let us recall that an approximate common interval is a common interval that may contain an overall bounded number of genes not belonging to the cluster (called gaps). The natural extension of nested common intervals – approximate nested common intervals – may thus be defined as follows: an approximate common interval $[i, j]$ is an approximate nested common interval if removing the gaps from $[i, j]$ produces a nested common interval.

The general APPROXIMATE NESTED COMMON INTERVALS problem may be defined as follows: *Given two genomes, find all their approximate nested common intervals.* As we will show, this last problem is fixed parameter tractable (FPT) in the number of allowed gaps in any approximate nested common interval.

Theorem 2 *Given two genomes as permutations (resp. sequences), finding all their approximate nested common intervals can be done in $O(n \cdot 3^k)$ (resp. $O(n^3 \cdot 15^k)$) where k is the maximal number of allowed gaps.*

We leave the problem of determining the complexity of the APPROXIMATE NESTED COMMON INTERVALS problem as an open problem.

5.1 Considering permutations

One can slightly modify Algorithm 2 in order to both, only report maximal occurrences and allowing a bounded number of gaps. In order to find all maximal approximate nested common intervals in two permutations, one has to call the `expansion` function defined in Algorithm 4 for all the starting positions of any maximal run (i.e. each position i , $1 \leq i \leq |\pi|$, such that $end[i] \geq i$):

$$\text{expansion}(maxgap, i, end[i], \min(\pi[i], \pi[end[i]]), \max(\pi[i], \pi[end[i]])).$$

Indeed, since one may use a gap to extend any maximal run, one has to start from every maximal run, even from those of size one. Lines 4 to 21 of Algorithm 4 are similar to the code of Algorithm 2 and correspond to the maximal expansion from positions l and r considering min and max as the current minimum and maximum values. Indeed, since there may be some gaps in $[l, r]$, computing the minimum and maximum values from the character set is not possible anymore. After this expansion process, we may report an approximate nested common interval if it contains at least two elements (i.e. $min \neq max$) and then try to extend it again by considering either the left neighbor (i.e. $l - 1$) or the right one (i.e. $r + 1$) or both as gaps (lines 25 to 35). This last part leads to the exponential complexity of the algorithm. The overall time complexity is in $O(n \cdot 3^k)$ where k is the maximal number of allowed gaps. This is straightforward to see since when allowing gaps, Lemma 5 still holds for the extension procedure and in lines 25 to 35, one has to call `expansion` procedure for each combination of $\{(l - 1, r), (l, r + 1), (l - 1, r + 1)\}$ until $nbgap$ becomes null. On the whole, in the worst case, the execution of the algorithm may needs 3^k calls to `expansive` (a call tree with nodes

Algorithm 4 Expansion function

```
1: expansion(nbgap, l, r, min, max) {
2: //try to extend the approximate nested common interval [l, r] with at most nbgap
3: //considering that min and max are resp. the minimum and maximum elements of [l, r]
4: repeat
5:    $l' \leftarrow l, r' \leftarrow r$ 
6:   if  $\pi[l'-1] = \text{min} - 1$  or  $\pi[r'+1] = \text{max} + 1$  then
7:     if  $\pi[l-1] = \text{min} - 1$  then
8:        $\text{min} \leftarrow \min(\text{min}, \pi[\text{end}[l-1]]); l \leftarrow \text{end}[l-1]$ 
9:     end if
10:    if  $\pi[r+1] = \text{max} + 1$  then
11:       $\text{max} \leftarrow \max(\text{max}, \pi[\text{end}[r+1]]); r \leftarrow \text{end}[r+1]$ 
12:    end if
13:  else
14:    if  $\pi[l-1] = \text{max} + 1$  then
15:       $\text{max} \leftarrow \max(\text{max}, \pi[\text{end}[l-1]]); l \leftarrow \text{end}[l-1]$ 
16:    end if
17:    if  $\pi[r+1] = \text{min} - 1$  then
18:       $\text{min} \leftarrow \min(\text{min}, \pi[\text{end}[r+1]]); r \leftarrow \text{end}[r+1]$ 
19:    end if
20:  end if
21: until  $l = l'$  and  $r = r'$ 
22: if  $\text{min} \neq \text{max}$  then
23:   report [min, max]
24: end if
25: if  $\text{nbgap} > 0$  then
26:   if  $l - 1 \geq 0$  then
27:     expansion( $\text{nbgap} - 1, l - 1, r, \text{min}, \text{max}$ )
28:   end if
29:   if  $r + 1 \leq |\pi|$  then
30:     expansion( $\text{nbgap} - 1, l, r + 1, \text{min}, \text{max}$ )
31:   end if
32:   if  $l - 1 \geq 0$  and  $r + 1 \leq |\pi|$  and  $\text{nbgap} > 1$  then
33:     expansion( $\text{nbgap} - 2, l - 1, r + 1, \text{min}, \text{max}$ )
34:   end if
35: end if
36: }
```

of degree 3 and depth *nbgap* ; i.e. *k*).

5.2 Considering sequences

Considering sequences, one can similarly slightly modify Algorithm 3 in order to handle the first two variants. In order to find all maximal approximate nested common intervals in two sequences, one has to call the `seq_expansion` function defined in Algorithm 5 for each position i , $1 \leq i \leq |S_1|$, and for each occurrence j of $S_1[i]$ in S_2 such that for all k , $1 \leq k \leq \sigma$, $c[k] = S_1[i]$:

`seq_expansion(i, i, j, j, c).`

As done in Algorithm 3, we use a bit vector $c[1, \dots, \sigma]$ in order to store the elements belonging to the current cluster and test efficiently if an element has already been added to it.

Lines 4 to 34 are quite similar to the code of Algorithm 3 and correspond to the maximal expansion from positions l_1, r_1 and l_2, r_2 considering c as the character set of the intervals without including gaps. After this expansion process, we may report an approximate nested common interval and then try to extend it again by considering (in lines 25 to 35) any combination of left neighbors (i.e. $l_1 - 1$ and $l_2 - 1$) and right ones (i.e. $r_1 + 1$ and $r_2 + 1$) as gaps.

This last part leads to the exponential complexity of the algorithm. The overall time complexity is in $O(n^3 \cdot 15^k)$ where k is the maximal number of allowed gaps. This is straightforward to see since the expansion can still be done in $O(n^3)$ time, whereas one has to test

Algorithm 5 Find all maximal approximate nested common intervals in two sequences

```

1: seq_expansion(nbgap,  $l_1, r_1, l_2, r_2, c$ ) {
2: //try to extend the a.n.c.i. ( $[l_1, r_1], [l_2, r_2]$ ) with at most nbgap considering that
3: //c only contains elements of ( $[l_1, r_1], [l_2, r_2]$ ) (excluding gaps).
4: repeat
5:   while  $c[S_1[l_1 - 1]] = \text{true}$  do  $l_1--$  done
6:   while  $c[S_1[r_1 + 1]] = \text{true}$  do  $r_1++$  done
7:   while  $c[S_2[l_2 - 1]] = \text{true}$  do  $l_2--$  done
8:   while  $c[S_2[r_2 + 1]] = \text{true}$  do  $r_2++$  done
9:    $l'_1 \leftarrow l_1, r'_1 \leftarrow r_1$ 
10:  if  $S_1[l_1 - 1] = S_2[l_2 - 1]$  or  $S_1[r_1 + 1] = S_2[r_2 + 1]$  then
11:    while  $S_1[l_1 - 1] = S_2[l_2 - 1]$  do
12:       $l_1--, l_2--, c[S_1[l_1]] \leftarrow \text{true}$ 
13:      while  $c[S_1[l_1 - 1]] = \text{true}$  do  $l_1--$  done
14:      while  $c[S_2[l_2 - 1]] = \text{true}$  do  $l_2--$  done
15:    end while
16:    while  $S_1[r_1 + 1] = S_2[r_2 + 1]$  do
17:       $r_1++, r_2++, c[S_1[r_1]] \leftarrow \text{true}$ 
18:      while  $c[S_1[r_1 + 1]] = \text{true}$  do  $r_1++$  done
19:      while  $c[S_2[r_2 + 1]] = \text{true}$  do  $r_2++$  done
20:    end while
21:  else
22:    while  $S_1[l_1 - 1] = S_2[r_2 + 1]$  do
23:       $l_1--, r_2++, c[S_1[l_1]] \leftarrow \text{true}$ 
24:      while  $c[S_1[l_1 - 1]] = \text{true}$  do  $l_1--$  done
25:      while  $c[S_2[r_2 + 1]] = \text{true}$  do  $r_2++$  done
26:    end while
27:    while  $S_1[r_1 + 1] = S_2[l_2 - 1]$  do
28:       $r_1++, l_2--, c[S_1[r_1]] \leftarrow \text{true}$ 
29:      while  $c[S_1[r_1 + 1]] = \text{true}$  do  $r_1++$  done
30:      while  $c[S_2[l_2 - 1]] = \text{true}$  do  $l_2--$  done
31:    end while
32:  end if
33: until  $l_1 = l'_1$  and  $r_1 = r'_1$ 
34: report ( $[l_1, r_1], [l_2, r_2]$ )
35: for each valid combination of  $l'_1 \in l_1, l_1 - 1, l'_2 \in l_2, l_2 - 1, r'_1 \in r_1, r_1 + 1, r'_2 \in r_2, r_2 + 1$ 
do
36:   seq_expansion(nbgap',  $l'_1, r'_1, l'_2, r'_2, c$ ) with  $nbgap' = nbgap - |l'_1 - l_1| - |l'_2 - l_2| - |r'_1 - r_1| - |r'_2 - r_2|$ 
37: end for

```

any of the 15 combinations when allowing gaps.

6 Conclusion

In this article, we proposed a set of efficient algorithms considering the nestedness assumption in the common intervals model of gene clusters for genomes represented both as permutations and as sequences. Two main questions remain open: (1) finding a more efficient algorithm for the last variant of nested common intervals on sequences and (2) determining the complexity of the APPROXIMATE NESTED COMMON INTERVALS problem.

Acknowledgments

The authors wish to thank Ferdinando Cicalese, Martin Milanič and Mathieu Raffinot for helpful suggestions on the complexity of finding nested common intervals on sequences and on PQ-tree aspects.

References

- Angibaud, S., Fertin, G., Rusu, I., Thévenin, A., and Vialette, S. (2009). On the approximability of comparing genomes with duplicates. *J. Graph Algor. Appl.*, 13(1):19–53.
- Bergeron, A., Chauve, C., de Montgolfier, F., and Raffinot, M. (2008a). Computing common intervals of k permutations, with applications to modular decomposition of graphs. *SIAM J. Discret. Math.*, 22(3):1022–1039.
- Bergeron, A., Corteel, S., and Raffinot, M. (2002). The algorithmic of gene teams. In *Proceedings of WABI 2002*, volume 2452 of *LNCIS*, pages 464–476.
- Bergeron, A., Gingras, Y., and Chauve, C. (2008b). Formal models of gene clusters. In

Mandoiu, I. and Zelikovsky, A., editors, *Bioinformatics Algorithms: Techniques and Applications*, chapter 8, pages 177–202. Wiley.

Bergeron, A. and Stoye, J. (2006). On the similarity of sets of permutations and its applications to genome comparison. *J. Comp. Biol.*, 13(7):1340–1354.

Blin, G., Chauve, C., Fertin, G., Rizzi, R., and Vialette, S. (2007). Comparing genomes with duplications: a computational complexity point of view. *ACM/IEEE Trans. Comput. Biol. Bioinf.*, 14(4):523–534.

Böcker, S., Jahn, K., Mixtacki, J., and Stoye, J. (2009). Computation of median gene clusters. *J. Comput. Biol.*, 16(8):1085–1099.

Booth, K. S. and Lueker, G. S. (1976). Testing for the consecutive ones property, interval graphs and graph planarity using *PQ*-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379.

Didier, G., Schmidt, T., Stoye, J., and Tsur, D. (2007). Character sets of strings. *J. Discr. Alg.*, 5(2):330–340.

He, X. and Goldwasser, M. H. (2005). Identifying conserved gene clusters in the presence of homology families. *J. Comp. Biol.*, 12(6):638–656.

Hoberman, R. and Durand, D. (2005). The incompatible desiderata of gene cluster properties. In *Proceedings of Recomb-CG 2005*, volume 3678 of *LNBI*, pages 73–87.

Kurzik-Dumke, U. and Zengerle, A. (1996). Identification of a novel *Drosophila melanogaster* gene, *angel*, a member of a nested gene cluster at locus 59F4,5. *Biochim. Biophys. Acta*, 1308(3):177–181.

Landau, G. M., Parida, L., and Weimann, O. (2005). Gene proximity analysis across whole

genomes via PQ trees. *J. Comp. Biol.*, 12(10):1289–1306.

Ohno, S. (1970). *Evolution by gene duplication*. Springer Verlag.

Rahmann, S. and Klau, G. W. (2006). Integer linear programs for discovering approximate gene clusters. In *Proceedings of WABI 2006*, volume 4175 of *LNBI*, pages 298–309.

Schmidt, T. and Stoye, J. (2004). Quadratic time algorithms for finding common intervals in two and more sequences. In *Proceedings of CPM 2004*, volume 3109 of *LNCS*, pages 347–358.

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

Zhang, M. and Leong, H. W. (2009). Gene team tree: A hierarchical representation of gene teams for all gap lengths. *J. Comp. Biol.*, 16(10):1383–1398.