Finding Nested Common Intervals Efficiently

Guillaume Blin[1] and Jens Stoye[2]

[1] Université Paris-Est, LIGM - UMR CNRS 8049, France. gblin@univ-mlv.fr
[2] Technische Fakultät, Universität Bielefeld, Germany.
stoye@techfak.uni-bielefeld.de

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 $O(n^3)$ time algorithm for finding all nested common intervals. We reduce this complexity by providing an $O(n^2)$ time algorithm computing an irredundant output. Finally, we show, by providing a third algorithm, that finding only the maximal nested common intervals can be done in linear time. 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.

1 Introduction and related work

Computational comparative genomics is a recent and 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 the same varieties of 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 [12], 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* [1, 7, 14, 15] (genes must occur consecutively, regardless of their order), *conserved intervals* [4] (common intervals, framed by the same two genes), *gene teams* [16, 2, 8] (genes in a cluster must not be interrupted by long stretches of genes not belonging to the cluster), and *approximate common intervals* [6, 13] (common intervals that may contain few genes from outside the cluster). For more details, please refer to [3].

In this article, we focus on another model – namely the *nested common intervals* – which was mentioned in [9]. In this model, an additional constraint – namely the *nestedness* – (observed in real data [10]) is added to the cluster definition. Hoberman and Durand [9] 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, [9] 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 fact, no explicit algorithmic analysis is given in [9], which might be a bit dangerous in view of the fact that genomes may consist of up to 25,000 and more genes. In the following, we will give some efficient algorithms to find all gene clusters – considering nested common intervals – between two genomes.

Let π_1 and π_2 be our genomes, represented as permutations over $N := \{1, \ldots, n\}$. For any $i \leq j$, $\pi[i, j]$ will refer to the sequence of elements $(\pi[i], \pi[i+1], \ldots, \pi[j])$. 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.

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). 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

$C_2 \subseteq C_3 \subseteq \cdots \subseteq C$ with $|C_i| = i$. 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 but will leave the case considering gaps as an open problem.

In [1], Bergeron et al. proposed a theoretical framework for computing common intervals based on a linear space basis. Of importance here is the technique proposed in [1] in order to generate the PQ-tree [11] 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 did not follow that approach, since (1) PQ-trees are a heavy machinery and quite space consuming in practice, and (2) our aim was to provide easy-to-implement algorithms with small constants in the O-notation. Moreover, algorithms based on PQ-trees do not easily generalize to sequences.

2 Nested common intervals on permutations

As described in [7, 14, 15], 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.

2.1 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 [9] and straightforwardly following the definition of nested common intervals, can be defined as in Algorithm 1.

Algorithm 1 Find all nested common intervals

1: **for** $i \leftarrow 1, \ldots, n$ **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**

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 intervals will be identified more than once. The worst case is when one considers $\pi = (1, 2, \ldots, n)$. 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 $\overleftarrow{x} := (x_k, x_{k-1}, \ldots, x_1)$ denotes the reverse of sequence $x = (x_1, x_2, \ldots, x_k)$. 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: $\pi_1 = (\underline{1,2,3,4},\underline{5},\underline{6,7,8},\underline{9})$ and $\pi_2 = (\underline{4,3,2,1},\underline{5},\underline{9},\underline{6,7,8})$.

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.

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. □

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.

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). 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 $[5,6]$, $[4,6]$, $[1,6]$,

Algorithm 2 Find all nested common intervals, irredundant version

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

$[2, 6]$, $[3, 6]$ when $i = 5$, and nothing for the other values of i. Let us prove this interesting property of Algorithm 2.

Proposition 1. *In Algorithm 2, any breakpoint is considered (i.e. passed through) at most once during the extension procedure described from lines 8 to 14.*

Proof. 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

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

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 can be easily provided for 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

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

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 can be easily provided for 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 proposition is proved. □

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

Proposition 2. *In Algorithm 2, two different runs cannot yield the same nested common interval.*

Proof. In order to be possibly reported twice, an interval would have to be a superinterval of 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 Proposition 1. □

2.2 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.

Proposition 3. *Every element of π is a member of at most three different maximal nested common intervals.*

Proof. This follows immediately from the correctness of Proposition 1. Indeed, according to Proposition 1 each position can be reached from at most two directions. Thus, the only case where an element of π may be member of 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 member of $(2, 1, 3, 4)$, $(3, 4)$ 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. Proposition 3 implies that the number of maximal nested common intervals is in $O(n)$, leading to an overall linear time.

3 Nested common intervals on sequences

In this section, we will give algorithms (mainly ideas due to space constraints) 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 $\Sigma = \{1, \ldots, \sigma\}$ 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.

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 [7, 14] and for approximate common intervals [6].

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 $c[1, \ldots, \sigma]$.

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. This problem 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. Surprisingly, the nestedness constraint leads to a polynomial time algorithm whereas for many other paradigms, considering matching and duplicates leads to hardness [5].

For this last variant, we unfortunately 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 $v' = ([i'_1, j'_1], [i'_2, j'_2])$ iff 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 $(ci_1, ci_2, \ldots ci_k)$ in this DAG, where $ci_1, ci_2, \ldots ci_k$ are common intervals, induces that $ci_1 \subseteq ci_2 \subseteq \ldots \subseteq ci_k$ and $\forall 1 \leq j < k$ we have

Algorithm 3 Find all maximal nested common intervals in two sequences

```
1: for i ← 1, ..., |S_1| do
2:    for each occurrence j of S_1[i] in S_2 do
3:       for each k ← 1, ..., σ do c[k] ← (k = S_1[i]) done
4:       l_1 ← i, r_1 ← i
5:       l_2 ← j, r_2 ← j
6:       repeat
7:          while c[S_1[l_1 − 1]] = true do l_1 ← l_1 − 1 done
8:          while c[S_1[r_1 + 1]] = true do r_1 ← r_1 + 1 done
9:          while c[S_2[l_2 − 1]] = true do l_2 ← l_2 − 1 done
10:         while c[S_2[r_2 + 1]] = true do r_2 ← r_2 + 1 done
11:         l'_1 ← l_1, r'_1 ← 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]] ← true
15:               while c[S_1[l_1 − 1]] = true do l_1−− done
16:               while c[S_2[l_2 − 1]] = 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]] ← true
20:               while c[S_1[r_1 + 1]] = true do r_1++ done
21:               while c[S_2[r_2 + 1]] = 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]] ← true
26:               while c[S_1[l_1 − 1]] = true do l_1−− done
27:               while c[S_2[r_2 + 1]] = 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]] ← true
31:               while c[S_1[r_1 + 1]] = true do r_1++ done
32:               while c[S_2[l_2 − 1]] = 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
```

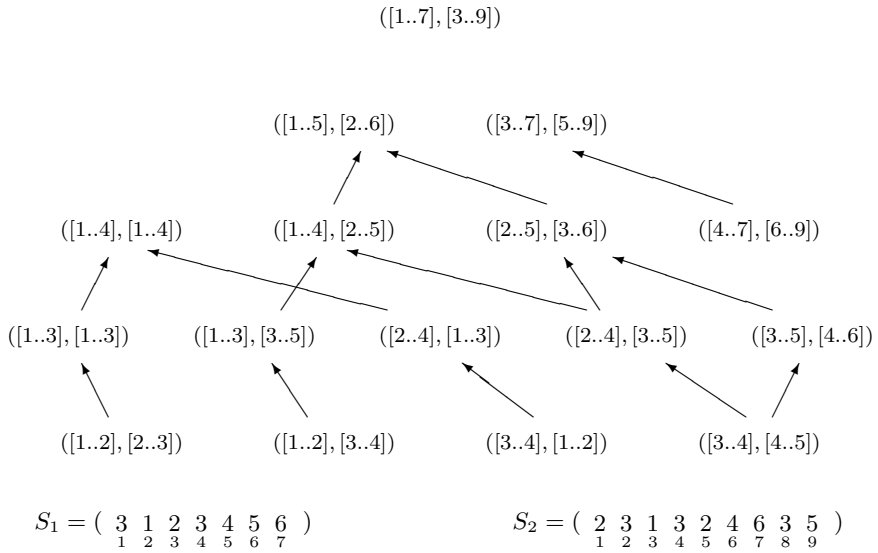

Fig. 1. Graph G for sequences $S_1 = (1, 3, 2, 3, 4, 5, 6)$ and $S_2 = (2, 3, 1, 3, 2, 4, 6, 3, 5)$.

$|ci_j| + 1 = |ci_{j+1}|$. 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. $O(n^2)$.

4 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) allowing clusters to be interrupted by up to g consecutive genes from outside the cluster, as in [9].

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-trees aspects.

References

1. A. Bergeron, C. Chauve, F. de Montgolfier, and M. Raffinot. Computing common intervals of k permutations, with applications to modular decomposition of graphs. *SIAM J. Discret. Math.*, 22(3):1022–1039, 2008.
2. A. Bergeron, S. Corteel, and M. Raffinot. The algorithmic of gene teams. In *Proceedings of WABI 2002*, volume 2452 of *LNCS*, pages 464–476, 2002.
3. A. Bergeron, Y. Gingras, and C. Chauve. Formal models of gene clusters. In I.I. Mandoiu and A. Zelikovsky, editors, *Bioinformatics Algorithms: Techniques and Applications*, chapter 8, pages 177–202. Wiley, 2008.
4. A. Bergeron and J. Stoye. On the similarity of sets of permutations and its applications to genome comparison. *J. Comp. Biol.*, 13(7):1340–1354, 2006.
5. G. Blin, C. Chauve, G. Fertin, R. Rizzi, and S. Vialette. Comparing genomes with duplications: a computational complexity point of view. *ACM/IEEE Trans. Comput. Biol. Bioinf.*, 14(4):523–534, 2007.
6. S. Böcker, K. Jahn, J. Mixtacki, and J. Stoye. Computation of median gene clusters. In *Proceedings of RECOMB 2008*, volume 4955 of *LNBI*, pages 331–345, 2008.
7. G. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. *J. Discr. Alg.*, 5(2):330–340, 2007.
8. X. He and M. H. Goldwasser. Identifying conserved gene clusters in the presence of homology families. *J. Comp. Biol.*, 12(6):638–656, 2005.
9. R. Hoberman and D. Durand. The incompatible desiderata of gene cluster properties. In *Proceedings of Recomb-CG 2005*, volume 3678 of *LNBI*, pages 73–87, 2005.
10. U. Kurzik-Dumke and A. Zengerle. 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, 1996.
11. G. M. Landau, L. Parida, and O. Weimann. Using pq trees for comparative genomics. In *Proceedings of CPM 2005*, volume 3537 of *LNCS*, pages 128–143, 2005.
12. S. Ohno. *Evolution by gene duplication*. Springer Verlag, 1970.
13. S. Rahmann and G. W. Klau. Integer linear programs for discovering approximate gene clusters. In *Proceedings of WABI 2006*, volume 4175 of *LNBI*, pages 298–309, 2006.
14. T. Schmidt and J. Stoye. Quadratic time algorithms for finding common intervals in two and more sequences. In *Proceedings of CPM 2004*, volume 3109 of *LNCS*, pages 347–358, 2004.
15. T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.
16. M. Zhang and H. W. Leong. Gene team tree: A compact representation of all gene teams. In *Proceedings of Recomb-CG 2008*, volume 5267 of *LNBI*, pages 100–112, 2008.