Quadratic Time Algorithms for Finding Common Intervals in Two and More Sequences

Thomas Schmidt[1] and Jens Stoye[2]

[1] International NRW Graduate School in Bioinformatics and Genome Research, Center of Biotechnology, Universität Bielefeld, 33594 Bielefeld, Germany
Thomas.Schmidt@CeBiTec.Uni-Bielefeld.de
[2] Technische Fakultät, Universität Bielefeld, 33594 Bielefeld, Germany
Stoye@TechFak.Uni-Bielefeld.de

Abstract. A popular approach in comparative genomics is to locate groups or clusters of orthologous genes in multiple genomes and to postulate functional association between the genes contained in such clusters. To this end, genomes are often represented as permutations of their genes, and common intervals, i.e. intervals containing the same set of genes, are interpreted as gene clusters. A disadvantage of modelling genomes as permutations is that paralogous copies of the same gene inside one genome can not be modelled.
In this paper we consider a slightly modified model that allows paralogs, simply by representing genomes as sequences rather than permutations of genes. We define common intervals based on this model, and we present a simple algorithm that finds all common intervals of two sequences in $\Theta(n^2)$ time using $\Theta(n^2)$ space. Another, more complicated algorithm runs in $O(n^2)$ time and uses only linear space. We also show how to extend the simple algorithm to more than two genomes, and we present results from the application of our algorithms to real data.

1 Introduction

The availability of completely sequenced genomes for an increasing number of organisms opens up new possibilities for information retrieval by whole genome comparison. The traditional way in genome annotation is establishing orthologous relations to well-characterized genes in other organisms on nucleic-acid or protein level. In the field of high-level genome comparison the attention is directed to gene order and content in related genomes, instead. During the course of evolution, speciation results in the divergence of genomes that initially have the same gene order and content. If there is no selective pressure, successive rearrangements that are common in prokaryotic genomes will eventually lead to a randomized gene order. Therefore the presence of a region of conserved gene order is a source of evidence for some non-random signal that allows, e.g., the prediction of groups of functionally associated genes [13].

Usually, two closely related prokaryotes share many *gene clusters*, which are sets of genes in close proximity to each other, but not necessarily contiguous

nor in the same order in both genomes [9]. The existence of such gene clusters has been explained in different ways: by functional selection [8], operon formation [3,7], and other processes in evolution which affect the gene order and content [10]. These papers show that the conservation of gene order is a source of information for many fields in genomic research. Unfortunately, the definition of gene clusters differs as the case arises, and models are based on heuristic algorithms which depend on very specific parameters like the size of gaps between genes. Also all of these approaches lack a statistical analysis to test the significance if an observed gene cluster occurs just by chance. Such an analysis was performed by Durand and Sankoff [5], who present probabilistic models to determine the significance of gene clusters, but leave open the question how to detect these gene clusters in two or more given genomes.

The first rigorous formulation of the concept of a gene cluster was given by Uno and Yagiura [12]. They introduced the notion of common intervals as contiguous regions in each of two permutations containing the same elements, and gave an optimal $O(n + K)$ time algorithm for finding all K common intervals in two permutations of n elements. Heber and Stoye [6] extended this result to common intervals of $k \geq 2$ permutations. But the simplicity of the model makes it unsuitable to be used on real data. Aspects like coding direction, paralogous genes, or the size of interleaving non-coding regions are ignored. On the other hand, model extensions quickly increase the computational complexity of algorithms for detecting gene clusters. As one step of extending the model while still staying within feasible computation time, in this paper we address the integration of paralogous genes, i.e. multiple copies of the same gene in a genome, into the model of common intervals, implying that we work on strings instead of permutations.

In [1], Amir *et al.* developed an algorithm applicable to our problem, using an efficient coding (fingerprints) of the sub-alphabets of substrings. The time complexity of their algorithm is $O(n|\Sigma| \log n \log |\Sigma|)$ where $|\Sigma|$ is the alphabet size. In our application, though, where the number of different genes (the alphabet size) is closely related to the length of the genome (we will always assume that $|\Sigma| \in \Theta(n)$), this becomes $O(n^2 \log^2 n)$. A recent algorithm, presented by Didier in [4], solves our problem using a tree-like data structure in $O(n^2 \log n)$ time, independent of the alphabet size. This algorithm will be further discussed in Section 5, where we show how its running time can be reduced to $O(n^2)$.

The main result of this paper is a worst-case optimal $\Theta(n^2)$ time and space algorithm based on elementary data structures that detects all common intervals of two strings. We also sketch how this algorithm can be extended to find gene clusters in more than two or in a subset of k' out of k genomes. The application of these algorithms on real data presented in Section 6 shows that the incorporation of paralogous genes and regions of internal duplication is a new source of information for research in the field of comparative genomics.

2 Basic Definitions

Given a string S over the finite alphabet of integers $\Sigma := \{1, ..., m\}$, $|S|$ is the length of S, $S[i]$ refers to the ith character of S, and $S[i, j]$ is the substring of S that starts with the ith and ends with the jth character of S. For convenience it will always be assumed for a string S that $S[0] = S[|S| + 1] = m + 1$ are characters not occurring elsewhere in S, so that border effects can be ignored when speaking of the left or right neighbor of a character in S. In our application of comparative genomics, the characters from Σ represent the genes. We will refer to S as a genome or a string interchangeably.

Definition 1 (character set). *Given a string S, the* character set *of a substring $S[i, j]$ is defined by*

$$\mathcal{CS}(S[i, j]) \ := \ \{S[k] \mid i \leq k \leq j\} \subset \Sigma.$$

A character set represents the set of all genes occurring in a given interval of a genome, where the order and the number of occurrences of paralogous copies of a gene is irrelevant.

Definition 2 ($\mathcal{CS}$-location, maximal). *Given a string S over an alphabet Σ and a subset $C \subseteq \Sigma$, the pair (i, j) is a* $\mathcal{CS}$-location *of C in S if and only if $\mathcal{CS}(S[i, j]) = C$. A $\mathcal{CS}$-location (i, j) of C in S is* left-maximal *if $S[i - 1] \notin C$, it is* right-maximal *if $S[j + 1] \notin C$, and it is* maximal *if it is both left- and right-maximal.*

A $\mathcal{CS}$-location of a subset C of Σ represents a contiguous region in a genome that contains exactly the genes contained in C, allowing for possible multiplicities. Note that C has a $\mathcal{CS}$-location in S if and only if C has a *maximal* $\mathcal{CS}$-location in S.

Definition 3 (common $\mathcal{CS}$-factor of k strings). *Given a collection of k strings $\mathcal{S} = (S_1, S_2, \ldots, S_k)$ over an alphabet Σ, a subset $C \subseteq \Sigma$ is a* common $\mathcal{CS}$-factor *of $\mathcal{S}$ if and only if C has a $\mathcal{CS}$-location in each S_l, $1 \leq \ l \leq k$.*

A common $\mathcal{CS}$-factor of k genomes represents a gene cluster that occurs in each of the k genomes. This concept is similar to a common interval of k permutations, but it allows the presence of paralogous genes in the genomes and particularly within a gene cluster.

These definitions motivate the following two problems:

Problem 1. Given a collection of k strings $\mathcal{S} = (S_1, S_2, \ldots, S_k)$, find all its common $\mathcal{CS}$-factors.

Problem 2. For each common $\mathcal{CS}$-factor of $\mathcal{S}$, find all its maximal $\mathcal{CS}$-locations in each of the S_l, $1 \leq \ l \leq k$.

Note that the solution of Problem 2 implies a solution of Problems 1. In this paper we present algorithms that solve both of these problems in optimal time and space.

3 A Simple Pairwise Algorithm

For $k = 2$ sequences, the best known algorithm so far solving Problems 1 and 2 requires $O(n^2 \log n)$ time and linear space [4] where n is the length of the longer of the two strings. Here we present an algorithm "Connecting Intervals" (CI) that solves the two problems in $\Theta(n^2)$ time and requires $\Theta(n^2)$ space. Moreover, we will show in the next section how this algorithm can easily be generalized to more than two genomes.

3.1 Basic Algorithm

The input for Algorithm CI are two strings S_1 and S_2, each of length $\leq n$, with characters drawn from the set $\Sigma = \{1, \ldots, m\}$, $m \leq 2n$. Its output are the pairs of $\mathcal{CS}$-locations of all common $\mathcal{CS}$-factors of S_1 and S_2. Pseudocode is given in Algorithm 1.

In a pre-processing step, the algorithm constructs two simple data structures, illustrated in Fig. 1. The first data structure, POS, contains for each character $c \in \Sigma$ a list $POS[c]$ that holds the positions of occurrence of c in sequence S_1 in ascending order, see Fig. 1 (a). The second data structure, NUM, is a $|S_1| \times |S_1|$ table where entry $NUM(i, j)$ contains the number $|\mathcal{CS}(S_1[i, j])|$ of *different* characters in the interval $S_1[i, j]$ for each $1 \leq i \leq j \leq |S_1|$, see Fig. 1 (b). Clearly, POS requires linear space and can be computed in linear time by a simple scan over S_1, while NUM requires $\Theta(n^2)$ space and its computation takes $\Theta(n^2)$ time.

(a)
$POS[1] = 2, 5$
$POS[2] = 3, 7$
$POS[3] = 1, 4$
$POS[4] = empty$
$POS[5] = 6$
$POS[6] = 8$

(b) $NUM(i, j)$:

$i \backslash j$	1	2	3	4	5	6	7	8
1	1	2	3	3	3	4	4	5
2		1	2	3	3	4	4	5
3			1	2	3	4	4	5
4				1	2	3	4	5
5					1	2	3	4
6						1	2	3
7							1	2
8								1

Fig. 1. Pre-processing of $S_1 = (3, 1, 2, 3, 1, 5, 2, 6)$ with $\Sigma = \{1, \ldots, 6\}$: **(a)** for each character $c \in \Sigma$, $POS[c]$ holds the positions at which c occurs in S_1; **(b)** the table NUM holding the values $|\mathcal{CS}(S_1[i, j])|$.

On a high level, Algorithm CI can be described as follows (see Fig. 2): For a fixed position i in S_2, while reading the substring of S_2 starting at that position, the observed characters in S_1 are marked and simultaneously maximal intervals of marked characters are tracked. This is iterated for all start positions i of substrings in S_2.

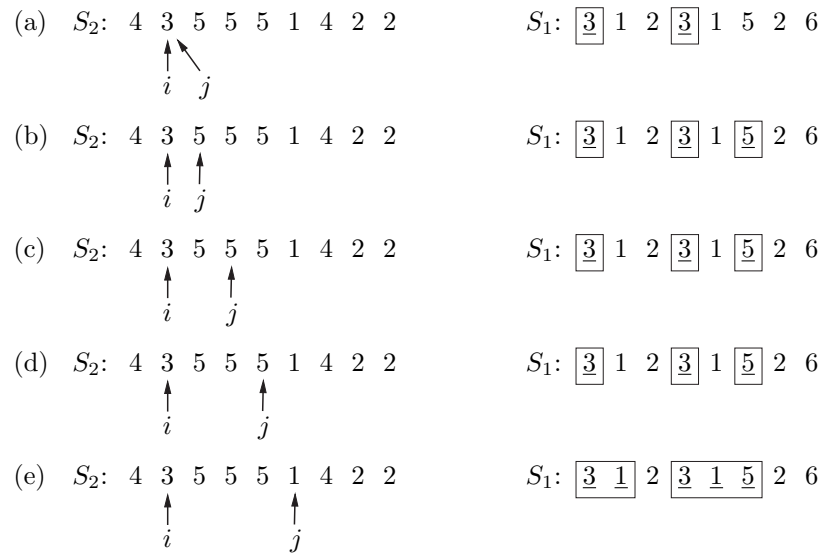

Fig. 2. Algorithm CI at a high level: Position $i = 2$ of S_2 is fixed as the left end of the increasing interval $[i, j]$. While moving j to the right, the observed characters are marked (underlined) in S_1, and maximal intervals of marked characters are tracked (the boxes).

The maximal intervals of marked characters in S_1 are candidates for common $\mathcal{CS}$-factors with the current interval $[i, j]$ of S_2. It only needs to be tested (i) if the character set of a candidate interval coincides with that of $S_2[i, j]$, and (ii) if the interval $S_2[i, j]$ is a maximal $\mathcal{CS}$-location of its character set.

In fact, to test (i) it suffices to compare the number of different characters in the two intervals. We know that the maximal marked intervals in S_1 contain a subset of the characters in $S_2[i, j]$, hence if the character sets have equal size, they must be equal. The number of different characters in $S_2[i, j]$ can be tracked while reading the substring of S_2 starting at position i. (In Algorithm 1 we use a binary vector OCC plus a counter $|OCC|$ that counts the number of ones in OCC.) The number of different characters in a maximal marked interval in S_1 can be read from the table NUM that was computed in the preprocessing phase.

Test (ii) is performed implicitly by the way how the value of j is incremented and the `while`-loop starting in line 5 of Algorithm 1 is terminated. Clearly, during the process of increasing j, once the interval $S_2[i, j]$ is not left-maximal for some $j \geq i$ (i.e. $S_2[i-1] = S_2[j']$ for some $j' \in \{i, \ldots, j\}$), it will never be left-maximal for any $j'' > j$. Hence it is a valid action to terminate the while loop as soon as $S_2[i, j]$ is not left-maximal, and left-maximality is guaranteed whenever the body of the while loop is entered. Right-maximality is explicitly tested in line 8 of Algorithm 1. This can be done in constant time by testing if $OCC[S_2[j+1]] = false$.

This establishes the correctness of the algorithm. For the analysis we have to show how the marking and tracking of maximal intervals in S_1 is performed. Obviously, marking the r occurrences of character $c = S_2[j]$ in S_1 is possible in $O(r)$ time using the list $POS[c]$. Further, if for each maximal interval of marked positions in S_1 the interval boundaries $(start, end)$ are stored at the left and right end of the interval, then it is easy to test, whenever a position p of S_1 is newly marked, if it connects to already existing intervals (ending at position $p-1$ or starting at position $p+1$ or both), and to increase these intervals by index p (if p connects to only one interval) or merge the two intervals (if p connects to two intervals). All this can be done in constant time for each newly marked position p of S_1.

Algorithm 1 Connecting Intervals (CI)

```
 1: pre-processing: build data structures POS and NUM
 2: for i = 1, ..., |S2| do
 3:    OCC[c] ← 0 for each character c in Σ, |OCC| ← 0
 4:    j ← i
 5:    while j ≤ |S2| and (i, j) is left-maximal in S2 do
 6:       c ← S2[j]
 7:       OCC[c] ← 1
 8:       while (i, j) is not right-maximal in S2 do
 9:          j ← j + 1
10:       end while
11:       for each position p in POS[c] do
12:          mark position p in S1
13:          find the maximal interval (start, end) of positions marked so far that con-
             tains position p
14:          if NUM(start, end) = |OCC| and (start, end) is maximal in S1 then
15:             output the pair ((i, j), (start, end))
16:          end if
17:       end for
18:       j ← j + 1
19:    end while
20: end for
```

Theorem 1. *Algorithm CI outputs all common $\mathcal{CS}$-factors of S_1 and S_2, in form of pairs of their maximal $\mathcal{CS}$-locations, in $\Theta(n^2)$ time using $\Theta(n^2)$ space.*

Proof. The `for`-loop starting in line 2 of Algorithm 1 is executed $|S_2| \leq n$ times; and in the outer `while`-loop together with the `while`-loop in line 8, j is incremented at most $|S_1| \leq n$ times. More difficult is the analysis of the `for`-loop starting in line 11. Here, observe that due to the test for right-maximality in line 8, this `for`-loop is reached for each character $c = S_2[j]$ only once, and hence for each i the body of the loop is executed at most $\sum_{c\in\Sigma} |POS[c]| = |S_1| \leq n$ times, where $|POS[c]|$ is the number of occurrences of character c in S_1. Together with

the pre-processing, this yields the overall $\Theta(n^2)$ time and space complexity. Due to the fact that the number of common $\mathcal{CS}$-factors can be as large as $n(n+1)/2$, e.g. assume $S_1 = S_2 = (1, 2, \ldots, n)$, this algorithm is time-optimal in the sense of worst case analysis. □

This simple quadratic-time algorithm solves the problem of detecting all common $\mathcal{CS}$-factors of two strings. It can also easily be extended to more than two strings (see Section 4), and it provides a good opportunity to address variations of the model (e.g. intervening non-coding regions, coding directions, or pseudogenes) while still staying within feasible computation time. The price for this simplicity is paid in space consumption. The table NUM, which is calculated during the pre-processing, consumes $\Theta(n^2)$ space. In Section 5 we will discuss a quadratic-time solution for Problems 1 and 2 that uses only linear space. Before extending the algorithm, we shortly discuss the form of the output.

3.2 Generating Non-redundant Output

Algorithm CI outputs the common $\mathcal{CS}$-factors by their maximal $\mathcal{CS}$-locations in S_1 and S_2, leading to a redundant output for paralogous gene clusters. For example, given $S_1 = (1, 2, 3, 1, 2)$ and $S_2 = (1, 2, 4, 1, 2, 5, 1, 2)$, the algorithm outputs the $\mathcal{CS}$-locations for the common $\mathcal{CS}$-factor $\{1, 2\}$ in the following way:

$$((1,2),(1,2)), ((1,2),(4,5)), ((1,2),(7,8)), ((4,5),(1,2)), ((4,5),(4,5)), ((4,5),(7,8)).$$

A non-redundant output of the following form, should be preferred, though:

$$S_1 : (1,2),(4,5) \quad - \quad S_2 : (1,2),(4,5),(7,8).$$

This output can be obtained by a modification of Algorithm CI that we only sketch here. Two additional tables LOC_1 and LOC_2, each of size $|S_1| \times |S_1|$, are used to store lists of intervals.

In a first step, Algorithm CI is applied to S_1 as first *and* second input sequence, yielding the paralogous gene clusters within S_1. These are stored in LOC_1 such that if (i', j') is contained in list $LOC_1(i, j)$, then $\mathcal{CS}(S_1[i', j']) = \mathcal{CS}(S_1[i, j])$, in the following way. Initially, all lists $LOC_1(i, j)$ are empty. Whenever a common $\mathcal{CS}$-factor with maximal $\mathcal{CS}$-locations (i, j) and (i', j'), $i' \neq i$, of a paralogous cluster is detected, then the $\mathcal{CS}$-location (i', j') is appended to the list in $LOC_1(i, j)$, and the interval (i', j') is marked, so that it is not being tested again.

In the second step, Algorithm CI is applied to S_1 and S_2, detecting the orthologous gene clusters between these two genomes. Whenever a common $\mathcal{CS}$-factor with maximal $\mathcal{CS}$-locations (i, j) in S_1 and (k, l) in S_2 is found, the $\mathcal{CS}$-location (k, l) is appended to $LOC_2(i, j)$. Finally, the output for each non-empty entry $LOC_2(i, j)$ is

$$S_1 : (i,j), LOC_1(i,j) \quad - \quad S_2 : LOC_2(i,j).$$

4 Multiple Genomes

To solve Problems 1 and 2 for any given $k \geq 2$, Algorithm CI can easily be extended to more than two strings. The general idea is that a set of characters $C \subseteq \Sigma$ is a common $\mathcal{CS}$-factor of $\mathcal{S} = \{S_1, \ldots, S_k\}$ if and only if it is a (pairwise) common $\mathcal{CS}$-factor of one fixed sequence (w.l.o.g. S_1) and all other sequences in $\mathcal{S}$. Therefore, Algorithm CI is applied to each pair of input strings (S_1, S_r) with $S_r \in \mathcal{S}$ and $1 \leq r \leq k$. Since the first input string is always S_1, the pre-processing step has to be performed only once. The k-fold application of Algorithm CI leads to an overall worst-case time and space complexity of $O(kn^2)$.

Unfortunately, with an increasing number of genomes, the probability to have a conserved gene cluster in all genomes decreases rapidly. For the use on biological data, it is hence even more interesting to find gene clusters which appear in only a subset of at least k' out of k given genomes. Based on the iterated use of Algorithm CI for multiple strings, its improvement to detect such gene clusters can be done in a straightforward manner. This yields a worst-case time complexity of $O(k(1 + k - k')n^2)$. The space complexity is $O(kn^2)$ if non-redundant output is written, and if only Problem 1 is to be solved, it can be reduced to $\Theta(n^2)$.

5 Saving Space

The basic algorithm for two sequences presented in Section 3 uses $\Theta(n^2)$ space, because for each interval $[i, j]$ of S_1 we store the number of different characters in that interval in table NUM. In this section we present an algorithm that runs in quadratic time and uses only linear space.

This algorithm is a modified version of the $O(n^2 \log n)$ time algorithm by Didier [4]. We sketch Didier's algorithm here and in detail discuss only those parts that need to be modified in order to obtain the improved time bound.

Similar to the main structure of Algorithm CI, Didier's algorithm generates, for a fixed left index i and variable right index $j = i, i{+}1, \ldots$ of intervals of S_2, candidate intervals $(start, end)$ in S_1, and then tests which of these candidates are indeed maximal locations of common $\mathcal{CS}$-factors. Didier uses a stack algorithm for generating the candidates, but the way we generate the candidates in Algorithm CI could be used as well. The main difference then is that Didier stores the intervals in a hierarchical manner according to their overlap relationship. Indeed, in Fig. 2, one can see this hierarchy for the boxes on the right hand side of the figure.

The key idea in the testing phase of Didier's algorithm is the notion of an *i-path* which is defined in the following way:

Definition 4 (*i*-rank, left-/right-neighbor, *i*-distance, successor, *i*-path).

1. *For a fixed position i in S_2, associate to each character $c \in \Sigma$ its* i-rank $\mathbf{r}_i(c)$*, i.e. the position of c in the list of* different *characters as they occur in left-to-right order in the suffix of S_2 starting at position i, and $+\infty$ if c does not occur in this list.*

2. *If $k \leq k'$ are positions of S_1, the* i-distance $\mathbf{d}_i(k, k')$ *between k and k' is the maximum i-rank of characters occurring in the substring $S_1[k, k']$.*
3. *For any position k of S_1 with a finite i-rank $r = \mathbf{r}_i(S_1[k])$, the* left-neighbor *(resp. the* right-neighbor*) of k is the greatest position smaller (resp. the smallest position greater) than k with i-rank $r + 1$, if it exists.*
4. *For a position k in S_1 of finite i-rank, its* successor *is its (left or right) neighbor with smaller i-distance. If both neighbors have infinite i-distance, k does not have a successor.*
5. *The* i-path *of position k of S_1 is the sequence of positions $p = (p_1, p_2, \ldots, p_d)$ of S_1 such that $p_1 = k$ and p_j is the successor of p_{j-1} for all $1 < j \leq d$.*

An important observation is then the following.

Theorem 2 (Didier [4]). *An interval candidate $(start, end)$ in S_1 with i-distance $\mathbf{d}_i(start, end) = d$ is a maximal occurrence of a common $\mathcal{CS}$-factor with the interval $[i, j]$ in S_2 if and only if it contains an i-path $(p_1, p_2, \ldots, p_d)$ of length d.*

Based on this theorem, Didier's algorithm traverses for each position k with $S_1[k] = 1$ its i-path $(p_1, p_2, \ldots, p_d)$ and, for each position p_j on this path, it tests if all the positions traversed on the path are contained in the interval $(start, end)$ where *start* is the leftmost index $k' \leq p_j$ such that $\mathbf{d}_i(k', p_j) = j$ and *end* is the rightmost index $k' \geq p_j$ such that $\mathbf{d}_i(p_j, k') = j$. In order to avoid that paths are traversed more than once, positions of S_1 are marked whenever the test has been done for the first time, and whenever a path that started at another position k' enters a path that was already traversed before, the procedure is stopped. For a fixed value of i, this part of the algorithm runs in linear time. However, the algorithm suggested in [4] for finding the i-successors and hence the i-paths takes time $O(n \log n)$ since a binary search in the sorted list of i-ranks occurring between p_j and its left- respectively right-successor is performed in order to compute the two i-distances. Repeated for each i, this is the reason for the $O(n^2 \log n)$ overall time complexity.

However, the problem of computing the i-distances is an application of the Range Maximum Query problem for which Bender and Farach [2] have shown how it can be solved in constant time per query after linear time preprocessing.

Hence we can state the following theorem.

Theorem 3. *All common $\mathcal{CS}$-factors of two strings S_1 and S_2 of maximal length n can be found in $O(n^2)$ time using $\Theta(n)$ space.*

6 Experimental Results

In order to show the positive effect of our model extension (sequences instead of permutations), we applied our algorithms to five bacterial genomes: *Corynebacterium glutamicum*, *Bacillus subtilis*, *Bacillus halodurans*, *Pseudomonas aeruginosa*, and *Mesorhizobium meliloti*, selected due to their varying pairwise evo-

lutionary distance. All five genomes are included in the COG (Clusters of Orthologous Groups of proteins) database [11], and we assume that two genes are homologous (orthologous or paralogous) if they are in the same COG cluster.

For these five genomes Algorithm CI reported 3428 gene clusters[3], where 197 clusters (6%) contain at least one paralogous gene, 216 clusters (6%) cover at least one region of internal duplication, and 86 clusters (3%) belong to both groups, see Fig. 3 (a). This results in 499 clusters (15%) containing at least one paralogous gene or one region of internal duplication, which an algorithm based on permutations would not be able to find.

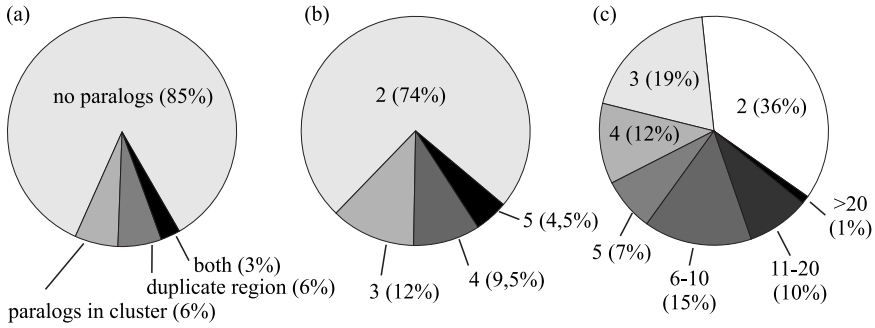

Fig. 3. Results of Algorithm CI applied to five bacterial genomes: (a) distribution of cluster type, (b) number of genomes where a cluster is found, (c) distribution of cluster sizes.

Fig. 3 (b) shows that the majority (74%) of gene clusters was found only between two genomes. However, a large portion of them (∼72%) stems from the comparison of *B. subtilis* and *B. halodurans* as a result of their very low evolutionary distance. Here, the prediction of functional roles from cluster conservation could be expected to fail, because the conservation results less from selection than from the fact that these genomes did not have enough time to diverge. A more detailed analysis of the gene clusters appearing in all five, or at least four sequences revealed that they contain so-called house-keeping genes, whose products are essential for the organism (e.g., ribosomal proteins, ABC transporters, or transcription related proteins). We also found some clusters of essential genes detected only very fragmentary, because the missing of just one gene in one of the genomes is sufficient to 'destroy' this cluster.

The evaluation of the cluster size, see Fig. 3 (c), showed that ∼90% of the clusters contain less than 10 genes. The gene cluster with the maximum number of genes was a highly conserved region from *B. subtilis* and *B. halodurans* con-

[3] Here, we call a common $\mathcal{CS}$-factor a *gene cluster* if it contains at least two different genes and has a $\mathcal{CS}$-location in at least two sequences.

taining 27 genes. Based on these numbers, it seems to be possible to limit the maximum length of a gene cluster to a fixed value, and thus reducing the time complexity of Algorithm CI to $O(n)$.

7 Discussion and Future Work

In this paper, we have presented a gene cluster model based on common intervals that includes the notion of paralogous genes and regions of internal duplication. We also presented Algorithm CI, which is the first quadratic time algorithm that detects these gene clusters in two genomes, and sketched the extension of this algorithm to be used on any given number of genomes. The evaluation on a set of five bacterial genomes revealed that Algorithm CI finds ∼15% more gene clusters than any algorithm working on permutations. To use gene clusters for functional prediction, it is necessary to use genomes with a sufficient evolutionary distance to avoid finding gene clusters that did not have enough time to diverge. The evaluation of the cluster size provides the opportunity to reduce the time complexity to linear by setting a fixed maximum cluster size. In future, this reduction possibly allows to generalize the model to report also gene clusters with a small symmetric set difference.

Acknowledgments

The authors wish to thank Gilles Didier, Mathieu Raffinot, Sven Rahmann, and David Sankoff for helpful discussions on the topic of gene clusters.

References

1. A. Amir, A. Apostolico, G.M. Landau, and G. Satta. Efficient text fingerprinting via parikh mapping. *J. Discr. Alg.*, 26:1–13, 2003.
2. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics, LATIN 2000*, volume 1776 of *LNCS*, pages 88–94. Springer Verlag, 2000.
3. P. Bork, B. Snel, G. Lehmann, M. Suyama, T. Dandekar, W. Lathe III, and M. A. Huynen. Comparative genome analysis: exploiting the context of genes to infer evolution and predict function. In D. Sankoff and J. H. Nadeau, editors, *Comparative genomics*, pages 281–294. Kluwer Academic Publishers, 2000.
4. G. Didier. Common intervals of two sequences. In *Proceedings of the Third International Workshop on Algorithms in Bioinformatics, WABI 2003*, pages 17–24.
5. D. Durand and D. Sankoff. Tests for gene clustering. *J. Comput. Biol.*, 10(3/4):453–482, 2002.
6. S. Heber and J. Stoye. Finding all common intervals of k permutations. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001*, pages 207–218, 2001.
7. W.C. Lathe III, B. Snel, and P. Bork. Gene context conservation of a higher order than operons. *Trends Biochem. Sci.*, 25:474–479, 2000.

8. R. Overbeek, M. Fonstein, M. D'Souza, G.D. Pusch, and N. Maltsev. The use of gene clusters to infer functional coupling. *Proc. Natl. Acad. Sci. USA*, 96:2896–2901, 1999.
9. I.B. Rogozin, K.S. Makarova, J. Murvai, E. Czabarka, Y.I. Wolf, R.L. Tatusov, L.A. Szekely, and E.V. Koonin. Connected gene neighborhoods in prokaryotic genomes. *Nucleic Acids Res.*, 30:2212–2223, 2002.
10. J. Tamames, G. Casari, C. Ouzounis, and A. Valencia. Conserved clusters of functionally related genes in two bacterial genomes. *J. Mol. Evol.*, 44:66–73, 1997.
11. R.L. Tatusov, D.A. Natale, I.V. Garkavtsev, T.A. Tatusova, U.T. Shankavaram, B.S. Rao, B. Kiryutin, M.Y. Galperin, N.D. Fedorova, and E.V. Koonin. The COG database: new developments in phylogenetic classification of proteins from complete genomes. *Nucleic Acids Res.*, 29:22–28, 2001.
12. T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26:290–309, 2000.
13. I. Yanai and C. DeLisi. The society of genes: networks of functional links between genes from comparative genomics. *Genome Biol.*, 3:0064.1–12, 2002.