JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 16, Number 8, 2009

Pp. 1085–1099
DOI: 10.1089/cmb.2009.0098

Research Article

Computation of Median Gene Clusters

SEBASTIAN BÖCKER,[1] KATHARINA JAHN,[2] JULIA MIXTACKI,[2] and JENS STOYE[3]

ABSTRACT

Whole genome comparison based on gene order has become a popular approach in comparative genomics. An important task in this field is the detection of gene clusters, i.e., sets of genes that occur co-localized in several genomes. For most applications, it is preferable to extend this definition to allow for small deviations in the gene content of the cluster occurrences. However, relaxing the equality constraint increases the computational complexity of gene cluster detection drastically. Existing approaches deal with this problem by using simplifying constraints on the cluster definition and/or allowing only pairwise genome comparison. In this article, we introduce a cluster concept named *median gene clusters* that improves over existing models, present efficient algorithms for their computation and show experimental results on the detection of approximate gene clusters in multiple genomes.

Key words: algorithms, combinatorics, comparative genomics, gene cluster detection.

1. INTRODUCTION

THE INCREASING AVAILABILITY of completely sequenced and assembled genomes opens the opportunity to compare whole genomes based on their gene order. It is well known that, during the course of evolution, rearrangement events, gene loss and gene duplications lead to a divergence of genomes that initially had the same gene order and gene content. If no selective pressure was acting on these processes, gene order and content would be randomized over time. Therefore, the existence of conserved regions is used as a source of information for comparative genomics (Dandekar et al., 1998). For that purpose genomes are modeled as strings or permutations of integers so that genes belonging to the same gene family are encoded by the same integer. A recent approach in this context is the computation of *gene clusters*, which are sets of genes that occur as single contiguous blocks in several genomes. Variable gene order and multiple occurrences of the same gene within the blocks are usually allowed. Gene clusters of this type are known as *common intervals* and there exist efficient algorithms for their computation (Uno and Yagiura, 2000; Heber and Stoye, 2001a,b; Schmidt and Stoye, 2004; Bergeron et al., 2005; Didier et al., 2007).

However, for most applications the requirement of exact occurrences of gene clusters in the genomes turned out to be too strict. Hence, the concept of *approximate gene clusters* arose recently, which allows for small deviations in the gene content of cluster locations. The problem of this model extension is that the

[1]Institut für Informatik, Friedrich-Schiller-Universität Jena, Jena, Germany.
[2]International NRW Graduate School in Bioinformatics and Genome Research and [3]Technische Fakultät, Universität Bielefeld, Bielefeld, Germany.

search space of approximate gene cluster detection increases exponentially—depending on the cluster concept—either with the number of allowed deviations (Chauve et al., 2006) or the number of compared sequences (He and Goldwasser, 2005).

One approach to handle deviations of the gene content is by imposing constraints on the cluster locations: For example, *max-gap clusters* (Bergeron et al., 2002; He and Goldwasser, 2005) allow for an arbitrary number of gaps in the cluster locations, each up to a certain length, but find no approximate locations that have lost some genes of the cluster. Despite these restrictions, the complexity of this problem increases exponentially with the number of sequences, but is in $O(n^2)$ for two sequences, where n is the length of the longest sequence.

Another approach with a constrained cluster definition is an algorithm presented in Amir et al., (2007) that computes gene clusters with a perfect location (reference interval) in one genome and an approximate occurrence in another sequence in $O(n^3 + occ)$ time using $O(n^3)$ space, where occ is the output size. Computation of gene clusters restricted in this way is a subproblem of our approach to median gene cluster computation. We introduce an algorithm that solves this problem in $O(n^2(1+\delta)^2)$ time and $O(n^2)$ space, where $\delta \ll n$.

A less constrained model was presented in Rahmann and Klau (2006), resulting in a very general gene cluster model, including most other existing ones. In their approach, the authors solve the approximate gene cluster problem by an integer linear program.

In this article, we introduce a new cluster concept, named *median gene clusters*, that constrains only the sum of errors that may occur in the approximate occurrences of a gene cluster. This means that we take from each genome the best location of a gene cluster and sum over the missing and interrupting genes in these locations. In the main part of this article (Sections 3–6), we present an approach for the efficient computation of all median gene clusters in an arbitrary number of genomes; in Section 7, we apply our method to different genomic datasets, compare it to the approaches presented in He and Goldwasser (2005) and Rahmann and Klau (2006) and show its applicability to multiple genomes. This article is an extended version of the conference article (Böcker et al., 2008).

2. BASIC DEFINITIONS

In our context, a genome is a string of integers over a finite alphabet $\Sigma = \{1, \ldots, \sigma\}$. Genes belonging to the same gene family are represented by the same integer value. Given a string S, $|S|$ denotes the length of the string and $S[i]$ refers to its ith character. By $S[i, j]$ we refer to the substring of S that starts with its ith and ends with its jth character, $1 \le i \le j \le |S|$. We define the *character set* of a substring $S[i, j]$ of S as

$$\mathcal{CS}(S[i,j]) = \{S[m] | i \le m \le j\}.$$

Inversely, a substring $S[i, j]$ is called a *location* of a character set $C \subseteq \Sigma$ if and only if $C = \mathcal{CS}(S[i,j])$. Substrings $S_1[i_1,j_1], \ldots, S_k[i_k,j_k]$ of two or more strings $S_1, \ldots, S_k$ of equal character content $\mathcal{CS}(S_1[i_1,j_1]) = \ldots = \mathcal{CS}(S_k[i_k,j_k])$ are called *common intervals* of $S_1, \ldots, S_k$.

To simplify the notation of the following definitions, we assume that a sequence S of length n is extended on both ends by a terminal character $S[0] = S[n+1] \notin \Sigma$. A substring $S[i, j]$ is *left-maximal* with respect to a character set C if $S[i-1] \notin C$, *right-maximal* with respect to C if $S[j+1] \notin C$ and *maximal* with respect to C if it is both left- and right-maximal with respect to C. A substring $S[i, j]$ that is maximal with respect to its own character set $\mathcal{CS}(S[i, j])$ is called maximal.

We define the following metric on two character sets $C, C' \subseteq \Sigma$, called the *symmetric set distance*:

$$D(C, C') = |C \backslash C'| + |C' \backslash C|.$$

A *d-location* of a character set C in a sequence S is a substring $S[i, j]$ such that $D(C, \mathcal{CS}(S[i, j])) \le d$.

A character set $C \subseteq \Sigma$ is a *median* of a set of k character sets $C_1, \ldots, C_k \subseteq \Sigma$ if and only if $\sum_{l=1}^{k} D(C, C_l) \le \sum_{l=1}^{k} D(C', C_l)$ for all $C' \subseteq \Sigma$. Note that a median in this context is not necessarily unique. This is due to the fact that for even k a character occurring in the median can occur in exactly half of the k character sets. When removing this character from the median, the total distance to the character sets stays unchanged and the remaining characters form an alternative median.

The problem considered in this article is the following.

Problem 1. *Given k sequences $S_1, \ldots, S_k$, a minimum cluster size s and a distance threshold δ, compute all sets $C \subseteq \Sigma$ with $|C| \geq s$ for which there exist $S_1[i_1, j_1], \ldots, S_k[i_k, j_k]$ with pairwise intersecting character sets and C is a median of $\mathcal{CS}(S_1[i_1, j_1]), \ldots, \mathcal{CS}(S_k[i_k, j_k])$ with*

$$\sum_{l=1}^{k} D(C, \mathcal{CS}(S_l[i_l, j_l])) \leq \delta. \tag{1}$$

Such a set C is called a median gene cluster *of $S_1, \ldots, S_k$.*

Defining gene cluster properties that are biologically meaningful and algorithmically feasible is a delicate task; a survey of different cluster properties can be found in Hoberman and Durand (2005) and Bergeron et al. (2007). Therefore, variants of the above problem formulation and additional cluster properties will be discussed in Section 8.

3. A THREE-STEP APPROACH TO MEDIAN GENE CLUSTERS

Our strategy for finding all median gene clusters is based on the observation that whenever inequality (1) holds, the distances between the character sets of the involved substrings are limited by the following upper bound:

Lemma 1. *Let $S_1, \ldots, S_k$ be sequences with substrings $S_1[i_1, j_1], \ldots, S_k[i_k, j_k]$ such that for a given $\delta \geq 0$ there exists a $C \subseteq \Sigma$ with $\sum_{l=1}^{k} D(C, \mathcal{CS}(S[i_l, j_l])) \leq \delta$. Then, there is at least one substring $S_m[i_m, j_m]$, $1 \leq m \leq k$, with $C' = \mathcal{CS}(S_m[i_m, j_m])$ and*

$$\sum_{l=1}^{k} D(C', \mathcal{CS}(S_l[i_l, j_l])) \;\leq\; 2\frac{k-1}{k}\delta. \tag{2}$$

Proof. Among the substrings $S_1[i_1, j_1], \ldots, S_k[i_k, j_k]$ choose $S_m[i_m, j_m]$, $1 \leq m \leq k$, such that $D(C, \mathcal{CS}(S_m[i_m, j_m])) \leq \frac{\delta}{k}$. Let $C' = \mathcal{CS}(S_m[i_m, j_m])$. From the triangle inequality we infer:

$$\begin{aligned}
\sum_{l=1}^{k} D\big(C', \mathcal{CS}(S_l[i_l, j_l])\big) &\leq \sum_{l \neq m} \big(D(C', C) + D(C, \mathcal{CS}(S_l[i_l, j_l]))\big) \\
&\leq (k-2)\frac{\delta}{k} + \sum_{l=1}^{k} D(C, \mathcal{CS}(S_l[i_l, j_l])) \\
&\leq (k-2)\frac{\delta}{k} + \delta \leq 2\frac{k-1}{k}\delta.
\end{aligned}$$

■

Character sets such as the above C' are used to filter the search space of potential median gene clusters and are therefore named *cluster filters*.

Lemma 1 gives rise to the following approach, consisting of three steps:

1. First, we compute the set of all cluster filters C' for $S_1, \ldots, S_k$. For that purpose we test for all substrings of the k sequences whether their corresponding character set meets the conditions of a cluster filter.
2. In the second step, for each cluster filter C' we compute k-tuples of the form $(S_1[i_1, j_1], \ldots, S_k[i_k, j_k])$ where all $S_l[i_l, j_l]$, $1 \leq l \leq k$, are δ-locations of C' and at least one of them is a location of C' and inequality (2) holds.
3. Finally, we compute for each k-tuple from Step 2 the median(s) of the corresponding character sets. Medians that comply with the distance threshold of inequality (1) are reported as median gene clusters.

The restriction to δ-locations in Step 2 is feasible due to the triangle inequality of the metric D:

Observation 1. *Given two character sets C, $C' \subseteq \Sigma$ with $D(C,C') > \delta$, then there is no $M \subseteq \Sigma$ with $D(M,C) + D(M,C') \leq \delta$.*

Hence, no k-tuple with an element that is not a δ-location of its cluster filter has a median that complies with distance threshold δ.

4. COMPUTATION OF CLUSTER FILTERS (STEP 1)

For the computation of all cluster filters in a set of k sequences of length at most n, it is sufficient to test for each of the $O(kn^2)$ substrings whether there are δ-locations in the other $k-1$ sequences such that inequality (2) is fulfilled for the given distance threshold δ. A naive algorithm can determine the cluster filters in $O(k^2n^4)$ time by computing the pairwise distances between all pairs of substrings. In this section, we present two better approaches that are based on the algorithm *Connecting Intervals* (CI) (Schmidt and Stoye, 2004) for the computation of common intervals in a pair of sequences.

For simplicity, we give the detailed description of our algorithm for just two sequences. The extension to multiple sequences is straightforward and will be briefly addressed in Section 4.4. In the following, let $d = 2\frac{k-1}{k}\delta$. For $k = 2$ sequences this cancels out to $d = \delta$. At first, we will review the basic concepts of the original algorithm CI, before we show in Sections 4.2 and 4.3 how it can be adapted to find cluster filters.

4.1. The Connecting Intervals (CI) algorithm

Algorithm CI, presented in Schmidt and Stoye (2004), finds all common intervals of two sequences S_1 and S_2 of length at most n in $O(n^2)$ time and space.

In a preprocessing step an array called *POS* and a table called *NUM* are computed. *POS* is of length $|\Sigma|$ and lists for each character $c \in \Sigma$ all positions where it occurs in S_2. *NUM* is a $|S_2| \times |S_2|$ table in which *NUM*[*i, j*] contains the number of different characters that occur in the substring $S_2[i, j]$, $NUM[i, j] = |\mathcal{CS}(S_2[i, j])|$. For an example, see Figure 1.

The basic idea of the CI algorithm is that while going systematically through all maximal substrings $S_1[i, j]$ of the first sequence, using the array *POS* one iteratively generates and extends marked intervals in S_2 that consist only of characters occurring in the current interval $S_1[i, j]$. An example of this procedure is given in Figure 2.

Common intervals are retrieved by comparing the character content of $S_1[i, j]$ and the marked intervals in S_2. Since by construction the character sets of the marked intervals are subsets of $\mathcal{CS}(S_1[i, j])$, this can be tested by comparing their size, using the table *NUM*, and keeping track of the current size of $\mathcal{CS}(S_1[i, j])$. Only those intervals in S_2 that were extended by the latest character of the current $S_1[i, j]$ need to be considered for this test. (Other intervals do not contain this character and thus have a different character set.) Because of the systematic traversal of the maximal substrings of S_1, where for a fixed i the maximal substrings starting at i are processed one after the other for increasing values of j, each character is at most $|S_1|$ times the latest character of a substring of S_1. Hence, each position in S_2 becomes at most $[S_1]$ times newly marked, which involves each time either an extension of a marked interval, the merging of two marked intervals or the generation of a new marked interval, if both neighboring positions are not in $\mathcal{CS}(S_1[i,j])$ (Fig. 2). Thus, there are at most $|S_1| \cdot |S_2|$ changes of marked intervals and the same number of character set comparisons. In total this algorithm takes $O(n^2)$ time and $O(n^2)$ space.

$POS[1] = 1$
$POS[2] = 4, 7, 11$
$POS[3] = 2, 9$
$POS[4] = 5, 8$
$POS[5] = 3, 6, 12$
$POS[6] = 10$

$NUM[i, j]$:

$i \backslash j$	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	5	5	5	5	6	6	6
2		1	2	3	4	4	4	4	4	5	5	5
3			1	2	3	3	3	3	4	5	5	5
4				1	2	3	3	3	4	5	5	5
5					1	2	3	3	4	5	5	5
6						1	2	3	4	5	5	5
7							1	2	3	4	4	5
8								1	2	3	4	5
9									1	2	3	4
10										1	2	3
11											1	2
12												1

FIG. 1. For $S_2 = (1, 3, 5, 2, 4, 5, 2, 4, 3, 6, 2, 5)$ with $\Sigma = \{1, \ldots, 6\}$, the positions of each occurrence of a character c are stored in *POS*[c]. The entries of the table *NUM*[*i, j*] equal $|\mathcal{CS}(S_2[i,j])|$.

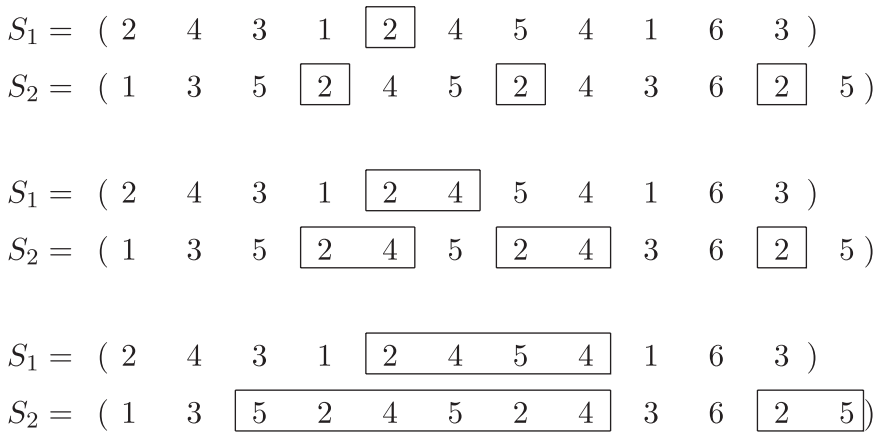

FIG. 2. Iteration through the substrings of S_1 beginning at position 5 and generation of intervals in S_2 that consist only of characters occurring in $S_1(5,j)$, $j \geq 5$: (a) $S_1(5, 5)$, three new blocks are marked in S_2; (b) $S_1(5, 6)$, two interval extensions; (c) $S_1(5, 7)$ is not right-maximal; $S_1(5, 8)$, two interval extensions and one interval merging; no further extension $S_1(5, j)$, $j > 8$, is left-maximal, so the next start position can be processed.

4.2. *An* $O(n^2(n + \delta^2))$ *time algorithm for cluster filter detection*

Our first algorithm for cluster filter detection is a straightforward extension of Algorithm CI that we call *Connecting Intervals with Errors* (CIE). Pseudocode is given in Algorithm 1. It uses the same preprocessing tables *NUM* and *POS* for S_2 as described above.

In the main part of the algorithm, we iterate through all maximal substrings $S_1[i, j]$ of S_1. We refer to the current $S_1[i, j]$ as *reference interval*. With array *OCC* and counter $|OCC|$, we keep track of the characters occurring in the current reference interval. In variable *minDist*, we store the minimal distance found so far between $CS(S_1[i, j])$ and the marked intervals in S_2. Like in the Connecting Intervals algorithm for each latest character c in $S_1[i, j]$, we mark each position p in the other sequence where this character occurs (lines 16, 17 of Algorithm 1). But then we have to do some extra work: While marking a position p in S_2, there is no need to keep track of maximal intervals of marked positions. Instead, positions to the left and right of p with increasing numbers $x, y \geq 1$ of unmarked characters are computed:

$$l_x = l_x(p) = \max(\{l \mid S_2[l, p] \text{ contains } x \text{ different unmarked characters}\} \cup \{0\})$$
$$r_y = r_y(p) = \min(\{r \mid S_2[p, r] \text{ contains } y \text{ different unmarked characters}\} \cup \{|S_2| + 1\})$$

By definition, the intervals $S_2[l_x + 1, r_y - 1]$ then contain at most $x + y - 2$ characters not occurring in $S_1[i, j]$ and are maximal. Hence, in order to find all occurrences of $S_1[i, j]$ around p with up to δ errors, it suffices to consider intervals $S_2[l_x + 1, r_y - 1]$ with $1 \leq x, y \leq \delta + 1$. An example is illustrated in Figure 3.

In line 21, we compute the distance of each of these $(\delta + 1)^2$ intervals to $\mathcal{CS}(S_1[i, j])$, i.e. $D(\mathcal{CS}(S_1[i, j]), \mathcal{CS}(S_2[l_x + 1, r_y - 1]))$. This is equal to the value of $|OCC| - NUM[l_x + 1, r_y - 1]$ plus twice the number of different unmarked characters in $S_2[l_x + 1, r_y - 1]$. In case this value is smaller than the current value of *minDist*, we update *minDist*.

Since we are also interested in intervals with missing characters, we need to consider intervals that do not contain c at all. But for these we know that their distance to the current substring $S_1[i, j]$ equals the distance to the previous $S_1[i, j']$ plus 1, with $j' < j$. We account for this in line 15 by increasing the value of *minDist*

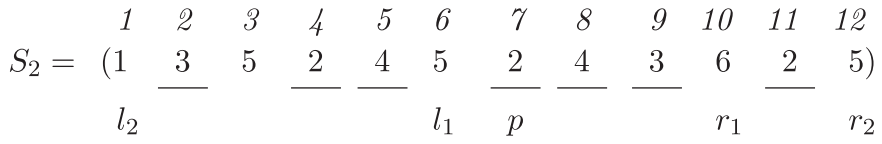

FIG. 3. For a substring of S_1 with character set {2, 3, 4}, its characters are marked in S_2. For $c = 2$ being the latest marked character and position $p = 7$, we have $l_1(7) = 6$, $l_2(7) = 1$ and $r_1(7) = 10$, $r_2(7) = 12$. Occurrences around p with up to $\delta = 1$ errors that need to be checked are $S_2[2, 9]$, $S_2[7, 9]$ and $S_2[7, 11]$.

by 1 after each extension of the reference interval. When we have finished all occurrences of c, we check the value of *minDist* to decide whether the current $S_1[i, j]$ qualifies as a cluster filter.

The crucial part in the analysis of Algorithm CIE is the **for** loop in line 16. From the analysis of Algorithm CI it follows immediately that each position p in S_2 is marked $O(n)$ times so that in total we mark $O(n^2)$ times a position. For each such position, we search for the positions $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$ (line 18). Performing this search in single steps takes $O(n)$ time. Then, we test for each of the $(\delta+1)^2$ pairs whether it fulfills the distance constraints (line 22), which can be done in constant time if we keep track of the number of unmarked characters in the substrings $S_2[l_x+1, r_y-1]$ while going through the **for** loop in line 19. In total, we thus have an $O(n^2(n+\delta^2))$ time algorithm, using $O(n^2)$ space for table *NUM*.

Remark: If we assume an upper bound b for the number of repetitions of each character in sequence S_2, the number of steps to locate the positions $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$ for a position p is bounded by $O(\min\{n, b\delta\})$. Hence, the overall running time decreases to $O(n^2(1+\min\{n, b\delta\}+\delta^2))$. This is especially relevant for sequences of gene labels, where the value of b is usually very small as it refers to the number of copies of a single gene in a genome.

4.3. An $O(n^2(1+\delta^2))$ time algorithm for cluster filter detection

The running time of the algorithm introduced in the previous section can be reduced to $O(n^2(1+\delta^2))$ when additional space of size $O(n\delta)$ is available. The speed-up is based on the observation that for each position p in sequence S_2 the values l_x and r_y are the same for all reference intervals $S_1[i, j]$ with a common left border. In a preprocessing step, we compute for the left-most left border in S_1, i.e. $i=1$, for each position p in S_2 the values $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$. These are stored in two tables L and R of size $\delta \times |S_2|$ each. The values of these tables need to be updated each time the left border i in S_1 is moved to the right which happens $O(n)$ times.

The details of the initialization and update of the tables L and R are given in the following. For simplicity, as in Didier et al. (2007), we re-name the characters in the sequences S_1 and S_2 by the rank of their first occurrence in the concatenated string $S_1[i,|S_1|]S_2$, initially for $i=1$, and after each shift of the left border i. This re-naming is a bijection RANK $\Sigma \to \{1, \ldots, |\Sigma|\}$. The consequence of the re-naming is that at the time the positions of a character c in S_2 are marked, the remaining unmarked characters c' are such that RANK $(c') >$ RANK (c).

The initialization of tables L (and R) is as follows: For each position p in S_2, we go to its left (and right) and look for the first $\delta+1$ different characters with a rank greater than RANK$(S_2(p))$. We store as $l_1, \ldots, l_{\delta+1}$ (and $r_1, \ldots, r_{\delta+1}$) the positions where a new different character is found. An example for RANK and the tables L and R is given in Figure 4.

When the left border in the substring of S_1 is shifted from i to $i+1$, the rank for all characters occurring between i and the next occurrence of the character $S_1[i]$ decreases by one while the rank of $c_{old}=S_1[i]$ increases by the number of different characters between the two occurrences. The tables L and R change in the following way. At positions belonging to occurrences of c_{old} in S_2 the table entries can change completely due to a possibly large change in the character number. We compute these entries anew by going through S_2 once from left to right and once from right to left and remembering the positions of the $\delta+1$ last read different characters with a rank greater than the new number of c_{old}. If a character is read

(a)

RANK$[1]=4$
RANK$[2]=1$
RANK$[3]=3$
RANK$[4]=2$
RANK$[5]=5$
RANK$[6]=6$

(b)

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
$L\backslash S_2'$	4	3	5	1	2	5	1	2	3	6	1	5
l_1	0	1	0	3	3	0	6	6	6	0	10	10
l_2	0	0	0	2	2	0	5	2	1	0	9	0
l_3	0	0	0	1	1	0	2	1	0	0	8	0
$R\backslash S_2'$	4	3	5	1	2	5	1	2	3	6	1	5
r_1	3	3	10	5	6	10	8	9	10	13	12	13
r_2	10	10	13	6	9	13	9	10	12	13	13	13
r_3	13	13	13	9	10	13	10	12	13	13	13	13

FIG. 4. Initialization of the rank for all characters (**a**) and the tables L and R (**b**). The characters of $S_1=(2, 4, 3, 1, 2, 4, 5, 4, 1, 6, 3)$ and $S_2=(1, 3, 5, 2, 4, 5, 2, 4, 3, 6, 2, 5)$ are re-named by the bijection RANK, defined by their first occurrence in the concatenated string $S_1[1,11]S_2$. The tables L and R are computed for the re-named sequence S_2'.

more than once, we only remember its latest occurrence. Once we reach a position of c_{old} in S_2 we fill the corresponding entries in L (respectively R) with the remembered positions. For positions in S_2 with a character different from c_{old} the entries in L and R can only change if the rank of the character is smaller than the new value of c_{old}. For these positions, we need to check whether an occurrence of c_{old} is close enough to become an entry in L and/or R. We test this by going through S_2 once from left to right and once from right to left and remembering the latest position of the character c_{old} in S_2. Once we reach a position with a character of smaller rank than the new value of c_{old}, we go through its entries in L (respectively R) and insert the remembered position of c_{old} at the right position in the field.

The initialization takes $O(n^2\delta)$ time and the update $O(n\delta)$ time for each increment of i. Combined with the unmodified rest of Algorithm CIE, the overall running time becomes $O(n^2\delta + n^2(1+\delta^2)) = O(n^2(1+\delta^2))$. The space consumption is $O(n\delta + n^2) = O(n^2)$.

4.4. Extension to multiple genomes

In this section, we show how the computation of cluster filters can be generalized to more than two genomes. First, note that in order not to miss any possible cluster filter C', we have to consider all substrings of *any* of the strings $S_1, \ldots, S_k$ as reference intervals, and not just substrings of S_1.

A reference interval $S_m[i, j]$ qualifies as a cluster filter if the sum of the minimal distances to the other $k-1$ sequences does not exceed $d = 2\frac{k-1}{k}\delta$. The threshold for pairwise distances is still δ, otherwise the total distance to the median exceeds δ due to Observation 1. Hence, we need to examine for the character most recently added to $S_m[i, j]$ all its occurrences in the other $k-1$ sequences and compute for each occurrence the distance of the corresponding $(\delta+1)^2$ intervals to $S_m[i, j]$. While doing so, we keep track of the minimum distances found in each of the $k-1$ sequences separately. If in the end they sum up to a value smaller or equal to d, we have found a new cluster filter. Due to this approach, we have to store the data structures *POS* and *NUM* and, if required, also L and R for $k-1$ sequences at a time.

It follows from these modifications that the running time multiplies by $O(k^2)$ for both variants of the algorithm while space requirements increase to $O(kn^2)$.

5. GENERATION OF k-TUPLES FROM δ-LOCATIONS OF A CLUSTER FILTER (STEP 2)

In the second step of the overall algorithm, for each cluster filter C' its maximal δ-locations in each of the sequences $S_1, \ldots, S_k$ are searched in order to form all k-tuples $(S_1[i_1,j_1], \ldots, S_k[i_k,j_k])$ with pairwise intersecting character sets that satisfy inequality (2).

5.1. Collection of δ-locations of a cluster filter

Maximal δ-locations of C' can be found efficiently using a similar search strategy as in Algorithm 1, but there is one essential difference. For Step 1, the computation of the minimal distance between the character set of the reference interval $S_m[i, j]$ and the substrings of another sequence, it was sufficient to consider explicitly only those substrings that contain the latest character c of $S_m[i, j]$ and are maximal with respect to $\mathcal{CS}(S_m[i, j])$.

When searching not only for the best but for all maximal δ-locations of C', Algorithm 1 needs to be modified such that also substrings not containing c and substrings that are not maximal with respect to C' are considered. The modified algorithm is shown in Algorithm 2. The data structures *POS*, *NUM*, RANK, L and R are computed by Algorithm 1. In order not to miss any δ-locations, we iterate for each character $c \in C'$ through all of its occurrences *pos* in S_2 (lines 1 and 2). Then we find borders $l_{\delta+1}(p)$ and $r_{\delta+1}(p)$ as in Algorithm 1 either by direct search or by a look-up in tables L and R. We iterate through all substrings between $l_{\delta+1}(p)+1$ and $l_{\delta+1}(p)-1$ that contain position p (lines 4 and 11) and keep track of the number of the characters not occurring in C' (lines 8 and 15). Finally, we compute the distance of the current substring to C' (line 17) to test whether it is a δ-location. The tests in lines 5, 7, 12 and 14 secure that no substring is considered more than once by stopping the interval extension once we either read another character from C' with higher rank than $S[p]$ or find another occurrence of $S[p]$ farther left than p.

We analyze running times of Algorithm 2 for *one* cluster filter: When tables L, R and *NUM* are available, the running time of Algorithm 2 is $O(n^2)$, since each substring of S_2 is processed at most once and in

constant time. While the asymptotic running time is independent of δ, the practical running time clearly depends on the number of δ-locations that have to be considered, and this number grows quadratically in δ. For multiple sequences, computation of maximal δ-locations has to be done for the $k-1$ sequences that are not the origin of the cluster filter. This can be done independently for each of these sequences, so that the running time is $O(kn^2)$ for multiple sequence comparison.

5.2. Generation of k-tuples from δ-locations

While the number of maximal δ-locations is in $O(kn^2)$, the number of k-tuples can be exponential in k even for small δ as the following example shows: For $\delta = 0$, $s = 3$ and k sequences of the form $S_l = (abcx_l)^{n/4}$, $1 \le l \le k$ and $x_i \ne x_j$ for $i \ne j$, there are $O(n^k)$ k-tuples for the cluster $C := \{a,b,c\}$. However, for sequences of gene labels where $|\Sigma|$ is in $\Theta(n)$ and reasonable values of δ, our experience shows that this approach is feasible.

For enumerating all candidate k-tuples for a cluster filter C', assume that the information about its δ-locations is stored in a table δ-loc of size $k \times (\delta + 1)$ that stores in position δ-loc$[l][dist]$ the list of δ-locations in S_l with distance exactly $dist$ to C'. For the reference sequence S_m, the origin of C', the list at position δ-loc$[m][0]$ contains exactly one element (i, j) with $\mathcal{CS}(S_m[i, j]) = C'$. All other lists at positions δ-loc$[m][dist]$, $dist > 0$, are empty. We can then use a recursive traversal of δ-loc to build all combinations of δ-locations of C' that contain exactly one δ-location from each of the k sequences. Due to Equation 2, we can use a branch-and-bound technique in this recursion that starts backtracking once the accumulated distances to C' in the current branch of the traversal exceed $d = 2\frac{k-1}{k}\delta$.

6. COMPUTATION OF MEDIAN GENE CLUSTERS FROM k-TUPLES (STEP 3)

The computation of the median M of a k-tuple of character sets $C_1, \ldots, C_k$ consists of a simple majority vote over the elements of $\mathcal{C} = \bigcup_{1 \le l \le k} C_l$, i.e. a gene occurring in at least half of the tuple elements becomes an element of the median. However, not every median must necessarily fulfill the distance constraint of inequality (1). Therefore, one needs to check whether the obtained median fulfills the distance constraint of inequality (1). Only then it is reported as median gene cluster. Computation of this distance can be done character-wise: Let p_c be the number of sets C_l that contain gene c, let n_c be the number of sets that do not contain gene c, then $p_c + n_c = k$. Let $d_c := \min\{p_c, n_c\}$ then

$$\sum_{l=1}^{k} d(M, C_l) = \sum_{c \in \mathcal{C}} d_c. \tag{3}$$

The running time of this step is $O(k|\mathcal{C}|)$, as the computation of both the median and the total distance can be done by a simple iteration through the characters occurring in the elements of the k-tuple.

6.1. Iterative median computation

While a median computation itself is not very time-consuming, it is the sheer number of k-tuples to be tested that make this step of the algorithm expensive. Remember that the number of k-tuples that originate from a single cluster filter can be exponential in k. However, these k-tuples are not all disjoint but share some elements as they originate from the recurrence through the table δ-loc in which the δ-locations are stored as described in Section 5.2. We can take advantage of this observation by computing medians iteratively, adding one element of the k-tuple after the other and computing intermediate medians after each step. Updating the distance to the intermediate median can then be done according to the following lemma:

Lemma 2. *Let $C_1, \ldots, C_k$ be character sets over Σ and let M_j be the median of $\{C_1, \ldots, C_j\}$, $1 \le j \le k$. Define $p_{c,j} = |\{l \mid c \in C_l, 1 \le l \le j\}|$. and $n_{c,j} = |\{l \mid c \notin C_l, 1 \le l \le j\}|$. Then the following equation holds:*

$$\sum_{l=1}^{j} d(M_j, C_l) = \sum_{l=1}^{j-1} d(M_{j-1}, C_l) + |C_j \setminus M_{j-1}| + |\{c \in M_{j-1} \setminus C_j : n_{c,j-1} \ne p_{c,j-1}\}|. \tag{4}$$

Proof. Based on equation (3) the proof can be done separately for each character $c \in \mathcal{C}$. Let $d_{c,j} = min\{n_{c,j}, p_{c,j}\}$. There are four cases to be considered: (i) $c \notin M_{j-1}$ and $c \notin C_j$, (ii) $c \in M_{j-1}$ and $c \in C_j$, (iii) $c \notin M_{j-1}$ and $c \in C_j$ and (iv) $c \in M_{j-1}$ and $c \notin C_j$.

(i) It holds that $n_{c,j-1} > p_{c,j-1}$ and therefore $d_{c,j-1} = p_{c,j-1}$. From $c \notin C_j$, we get $n_{c,j} = n_{c,j-1} + 1$ and $p_{c,j} = p_{c,j-1}$ so that $d_{c,j} = d_{c,j-1}$.

(ii) It holds that $p_{c,j-1} > n_{c,j-1}$ and therefore $d_{c,j-1} = n_{c,j-1}$. From $c \in C_j$, we get $p_{c,j} = p_{c,j-1} + 1$ and $n_{c,j} = n_{c,j-1}$ so that $d_{c,j} = d_{c,j-1}$.

(iii) It holds that $n_{c,j-1} > p_{c,j-1}$ and therefore $d_{c,j-1} = p_{c,j-1}$. From $c \in C_j$, we get $p_{c,j} = p_{c,j-1} + 1$ and $n_{c,j} = n_{c,j-1}$ so that

$$d_{c,j} = \begin{cases} p_{c,j} & \text{if } p_{c,j-1} < n_{c,j-1} - 1 \\ n_{c,j} & \text{if } p_{c,j-1} = n_{c,j-1} - 1 \end{cases}$$

For both cases, we get $d_{c,j} = d_{c,j-1} + 1$ since the distance increases by 1 either because $c \notin M_j$ or, in case $c \in M_j$, it follows that $d_{c,j} = n_{c,j} = d_{c,j-1} + 1$.

(iv) It holds that $p_{c,j-1} \geq n_{c,j-1}$ and therefore $d_{c,j-1} = n_{c,j-1}$. From $c \notin C_j$, we get $n_{c,j} = n_{c,j-1} + 1$ and $p_{c,j} = p_{c,j-1}$ so that

$$d_{c,j} = \begin{cases} n_{c,j} & \text{if } n_{c,j-1} < p_{c,j-1} \\ p_{c,j} & \text{if } n_{c,j-1} = p_{c,j-1} \end{cases}$$

In the first case, we get $d_{c,j} = d_{c,j-1} + 1$ as $c \in M_j$ but $\notin C_j$. For the second case with $c \in M_{j-1}$ but $c \notin M_j$, we get $d_{c,j} = p_{c,j} = p_{c,j-1} = n_{c,-1} = d_{c,j-1}$. Since there are no other cases, the value of $d_{c,j}$ increases by one if and only if $c \in C_j \setminus M_{j-1}$ or $c \in M_{j-1} \setminus C_j$ with $n_{c,j-1} \neq p_{c,j-1}$. In all other cases $d_{c,j} = d_{c,j-1}$ holds. ■

We can use the iterative distance computation of Lemma 2 to speed up the recursion through the lists of δ-locations of the k sequences. When adding a new substring S_l (i,j), we additionally compute how the distance to the intermediate median changes and start backtracking once this distance is bigger than δ. To compute the changes in the intermediate distance efficiently, we keep track of n_c and p_c for each character c that occurs in at least one of the substrings already added to the k-tuple. Then, we can test for each character in constant time whether one of the two cases occurs where the intermediate distance increases. It is not necessary to generate the intermediate medians explicitly. Instead, only the final median is computed once the k-tuple is filled completely.

The speed-up of the iterative median computation is two-fold: Firstly, we can stop adding new δ-locations once the distance to an intermediate median is greater than δ, since by adding another character set the distance never decreases. Secondly, for k-tuples that share the first $l < k$ elements the first l intermediate distance computations are performed only once.

6.2. Ambiguities and redundancies in median gene cluster computation

As mentioned earlier, there is no one-to-one relationship between k-tuples and their medians. In the following, we discuss the reasons for this phenomenon and suggest ways of post-processing the results for dealing with its consequences.

Remember from Section 2 that there can be several medians of a k-tuple if k is even. This happens when there are ties because some characters occur in exactly $k/2$ of the elements. Since each tie adds $k/2$ to the sum of distances, a median exists only if the number of ties is less than or equal to $\frac{2\delta}{k}$. In this case, there exist $2^{\frac{2\delta}{k}}$ different medians as the example of Figure 5 illustrates. In case of such ambiguities, we suggest to represent the set of medians by a combination of its largest and smallest element. Then, the complete set can be reconstructed from these representatives.

Besides the ambiguity of medians, it can happen that the same character set is generated more than once, either by duplicate k-tuples that occur if more than one of the k-tuple elements is a cluster filter, or by different k-tuples that have the same median by chance. While the first type of duplicate medians are clearly unwanted, duplicates of the second type can be of importance as they together with their k-tuples can provide information on multiple occurrences of gene clusters. However, duplicate medians can also originate from very similar yet not identical k-tuples. This occurs if corresponding elements of the k-tuples,

$$
\begin{array}{lccccccccc}
S_1 = & (\underline{1} & \underline{2} & \underline{1} & 3 & 1 & 4 & 1 & 5) \\
S_2 = & (\underline{1} & 2 & \underline{4} & \underline{1} & \underline{2} & \underline{1} & 3) \\
S_3 = & (1 & 3 & 3 & \underline{1} & \underline{2} & \underline{1} & 2) \\
S_4 = & (\underline{1} & \underline{4} & \underline{1} & \underline{1}) &&&&
\end{array}
\qquad
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
\hline
1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 \\
\hline
1 & 1 & 0 & T & 0
\end{array}
$$

FIG. 5. For $\delta = 3$ and the cluster filter $C' = \{1, 2\}$, one of its 3-locations in each of the sequences is given by the underlined substrings. The tie of character 4 (denoted by T) yields the two medians $\{1, 2\}$ and $\{1, 2, 4\}$.

i.e. substrings of the same sequence, are mostly overlapping. To avoid this kind of duplicates, we suggest to discard all k-tuples and their medians that contain substrings $S_l[i, j]$, $1 \leq l \leq k$ that are either extendable with respect to the median or that have a border element, $S_l[i]$ or $S_l[j]$, that is not contained in the median. Such duplicates can be easily filtered away in a post-processing step.

Another type of redundancy occurs when the elements of different k-tuples are nested, in the sense that all elements of one k-tuple $kTup = ((i_1, j_1), \ldots, (i_k, j_k))$are substrings of the corresponding elements in another k-tuple $kTup' = ((i'_1, j'_1), \ldots, (i'_k, j'_k))$, i.e., for all $1 \leq l \leq k$ it holds that $i_l \geq i'_l$ and $i_l \leq j'_j$. Apparently, the median M of $kTup$, is then a subset of the median M' of $kTup'$. Non-maximal medians of the form of M indicate better conserved sub-clusters of M' in case they have a smaller total distance to the substrings defined by $kTup$ than the complete cluster M' has to the substrings defined by $kTup'$. Hence, we suggest not to filter away such sub-medians unless they have the same or a larger total distance than the complete median.

6.3. *Ranking of detected median gene clusters*

Even after removing redundancies as described in the previous subsection, the remaining set of detected median gene clusters can still be too large for experimental validation. In this case, it is desirable to process the most promising clusters first. In absence of other knowledge, these would be those clusters that are the most significant in the sense that they are the least likely to occur by chance in the given genomes. It is reasonable to assume that the likelihood of a gene cluster to occur by chance decreases with its size and its degree of conservation. However, without a true statistical evaluation it is per se not clear whether a small but well-conserved gene cluster should be preferred over a large cluster that is not so well conserved, or vice versa. We avoid this dilemma by sorting the detected clusters by their size first and ranking them then only within each size bucket based on their degree of conservation. For a more sophisticated ranking scheme in which the score of a gene cluster corresponds directly to its statistical significance, a statistical model can be applied that takes into account the gene content of the given sequences and the different occurrence frequencies of genes. We are currently working on such a model.

7. EXPERIMENTAL RESULTS

In an initial test, we compared the performance of our two algorithms for cluster filter detection on several datasets. Surprisingly, we found that running times are highly similar between these algorithms in practice (data not shown). The following results were achieved using the second of the two algorithms.

To demonstrate the ability of our method we applied it to approximate gene cluster detection in various genomic datasets. We compared it to previous approaches for gene cluster detection in two sequences and additionally show its applicability to multiple genomes. All computations reported in this section were performed with a 1.66-GHz Intel Core Duo T2300 processor with 520 Mb of main memory running under the Suse Linux operating system.

7.1. *Comparison to* HomologyTeams

We reproduced the gene clusters reported in He and Goldwasser (2005) with our program. The dataset consisting of the genomes of *E. coli* and *B. subtilis* annotated with COG numbers was downloaded from http://euler.slu.edu/~goldwasser/homologyteams/. Setting the parameters of our method to $s = 4$ and $\delta = 1$ we detected 481 non-nested median gene clusters in this dataset, falling into 51 classes of non-overlapping

Table 1. Genomes Used for Multiple Genome Comparison

Species name	*RefSeq number*	*No. of genes*
Buchnera aphidicola APS	NC_002528	564
Escherichia coli K 12	NC_000913	4183
Haemophilus influenzae Rd	NC_000907	1709
Pasteurella multocida Pm 70	NC_002663	2015
Xylella fastidiosa 9a5c	NC_002488	2680

gene clusters. Among these clusters are the ten operons studied in He and Goldwasser (2005). These findings show that our method finds a superset of the gene clusters detected by the HomologyTeams software.

7.2. Comparison to ILP approach

To reproduce the study in Rahmann and Klau (2006), we obtained the annotated genomes of *C. glutamicum* and *M. tuberculosis* where genes are labeled according to gene family membership from http://gi.cebitec.uni-bielefeld.de/comet.

Our program finds the gene cluster reported in Rahmann and Klau (2006) in 17 seconds using parameters $\delta = 27$ and $s = 60$ while the ILP using CPLEX 9.03 runs for more than one hour on a superior processor to compute this cluster, using size parameter $D = 51$. In order to detect this cluster, an approach based on *max-gap clusters* needs to set its gap-size threshold as big as twelve such that the longest gap of unmatched genes can be bridged.

To evaluate our method on a broader basis, we conducted a similar series of experiments as reported in Rahmann and Klau (2006) to find optimal gene clusters for each size between 5 and 150. Since our method finds gene clusters based on a distance threshold and not for a certain size, we had to run our algorithm several times for different minimal cluster sizes and distance thresholds. Despite this overhead our method was able to find all optimal gene clusters in this size range within 3 hours and 4 minutes.

7.3. Experimental results on multiple genomes

Although both of the approaches above are in general applicable to multiple genomes, no experimental results on the comparison of more than two genomes were shown in the respective publications. To show the applicability of our method to multiple genomes, we searched for approximate gene clusters in five bacterial genomes summarized in Table 1. For homology assignment in this dataset, we used the GhostFam tool (Schmidt and Stoye, 2007), which is a completely automatic method for grouping genes into families based on sequence similarity. Applying the standard parameters for sequence comparison, the program distributed the 11,184 genes occurring in the given genomes into 5086 gene families. Further preprocessing of the data set reduced this number to 5029 by merging blocks of genes, that are both: perfectly conserved in respect to gene content and gene order (apart from reversed reading direction) and unique in the sense that each of the genes in the block occurs exactly once in each of the given genomes.

In Table 2, results of approximate gene cluster detection in this dataset are given for different combinations of s and δ.

The combinatorial explosion of Step 2 for larger δ is clearly visible in the presented results, as the number of possible k-tuples increases rapidly with growing δ. However, for most of the parameter ranges tested in this study, computation times are still manageable, which needs to be attributed partially to the iterative median computation reducing the number of generated k-tuples drastically.

An analysis of the predicted gene clusters showed that our method is capable of detecting biologically meaningful gene clusters in multiple genomes, as a couple of well-known gene clusters were reported by our method—among them a cluster of genes involved in cell division and cell wall biosynthesis and the gene cluster for ATP biosynthesis.

8. ALTERNATIVE PROBLEM FORMULATIONS

In this section, we discuss some variations of the gene cluster model described above.

TABLE 2. EXPERIMENTAL RESULTS ON FIVE BACTERIAL GENOMES AND THE CORRESPONDING COMPUTATION TIMES FOR DIFFERENT COMBINATIONS OF s AND δ

	$\delta=0$	$\delta=1$	$\delta=5$	$\delta=8$	$\delta=12$	$\delta=18$
$s=5$						
Running time	16s	20s	69s	1605s	—	—
Time for Step 1 in %	99.9	99.8	77.8	5.8	—	—
# cluster filters	35	59	1658	6406	$3.9 \cdot 10^4$	$7.8 \cdot 10^4$
# k-tuples	35	6929	$1.8 \cdot 10^9$	$2.2 \cdot 10^{11}$	$4.4 \cdot 10^{14}$	$3.0 \cdot 10^{16}$
# generated k-tuples	35	395	$1.1 \cdot 10^6$	$1.0 \cdot 10^8$	—	—
# median gene clusters	7	18	478	3636	—	—
# optimal gene clusters	5	10	114	694	—	—
# gene cluster classes	5	5	13	25	—	—
$s=10$						
Running time	16s	20s	53s	96s	2074s	—
Time for Step 1 in %	100.0	100.0	99.7	94.8	7.6	—
# cluster filters	5	9	126	255	1037	$2.1 \cdot 10^4$
# k-tuples	5	116	$1.9 \cdot 10^7$	$1.9 \cdot 10^9$	$2.6 \cdot 10^{11}$	$1.67 \cdot 10^{14}$
# generated k-tuples	5	33	9833	$4.0 \cdot 10^5$	$6.9 \cdot 10^7$	—
# median gene clusters	1	2	16	69	166	—
# optimal gene clusters	1	2	10	14	42	—
# gene cluster classes	1	1	2	2	3	—
$s=15$						
Running time	16s	20s	53s	91s	169s	$1.0 \cdot 10^4$s
Time for Step 1 in %	100.0	100.0	99.8	99.5	92.3	2.8
# cluster filters	0	4	49	84	178	627
# k-tuples	0	4	$3.5 \cdot 10^5$	$9.8 \cdot 10^6$	$1.8 \cdot 10^9$	$1.2 \cdot 10^{12}$
# generated k-tuples	0	4	3740	$5.1 \cdot 10^4$	$9.4 \cdot 10^5$	$2.5 \cdot 10^8$
# median gene clusters	0	1	8	15	21	51
# optimal gene clusters	0	1	5	5	5	5
# gene cluster classes	0	1	1	1	1	1
$s=20$						
Running time	16s	20s	52s	90s	163s	550s
Time for Step 1 in %	100.0	100.0	99.9	99.6	94.7	52.3
# cluster filters	0	0	41	72	129	261
# k-tuples	0	0	$3.4 \cdot 10^5$	$9.7 \cdot 10^6$	$5.3 \cdot 10^8$	$2.9 \cdot 10^{10}$
# generated k-tuples	0	0	3661	$5.0 \cdot 10^4$	$7.6 \cdot 10^5$	$1.4 \cdot 10^7$
# median gene clusters	0	0	6	12	15	39
# optimal gene clusters	0	0	3	3	3	3
# gene cluster classes	0	0	1	1	1	1
$s=25$						
Running time	16s	20s	52s	89s	157s	461s
Time for Step 1 in %	100.0	100.0	99.9	99.9	98.0	61.8
# cluster filters	0	0	9	21	61	176
# k-tuples	0	0	$3.9 \cdot 10^4$	$3.8 \cdot 10^6$	$3.4 \cdot 10^8$	$2.5 \cdot 10^{10}$
# generated k-tuples	0	0	38	2156	$1.8 \cdot 10^5$	$8.2 \cdot 10^6$
# median gene clusters	0	0	1	1	2	9
# optimal gene clusters	0	0	1	1	1	1
# gene cluster classes	0	0	1	1	1	1

The total running time and the fraction spent on Step 1 for the detection of all cluster filters (# cluster filters) is shown. Furthermore, the number of k-tuples that can be generated from the lists of δ-locations (# k-tuples), the number of k-tuples actually generated during iterative median computation (# generated k-tuples), the number of median gene clusters (# median gene clusters), the number of median gene clusters not nested within a better cluster (# optimal gene clusters) and the number of completely different, i.e., non-overlapping gene clusters (# gene cluster classes) is shown. Note: for parameter combinations for which computation was unfinished after 24 hours, only results for Step 1 are given.

8.1. Transformation set distance

We can define a set distance based on the maximal set difference instead of the symmetric set difference:

$$D_T(C, C') = \max\{|C \setminus C'|, |C' \setminus C|\}.$$

We call this distance measure the *transformation set distance* between C and C'. The transformation set distance is a metric.

It is easy to derive a simple linear time algorithm that finds, for a given character set C and a sequence S, all starting positions of substrings in S that have a transformation set distance that is smaller or equal to a given distance threshold d. Therefore, we can compute gene cluster candidates for the transformation set distance in time $O(k^2n^3)$. But the problem with respect to application in gene cluster detection is that we lack efficient methods to compute the median of character sets under transformation set distance.

8.2. Center representative

While being computationally tractable, selection of median representatives is probably not the best approach for gene cluster computation. The problem with median representatives is that the distances between the median and single objects (in our case, sequences) are not directly restricted, but only via the sum of all distances. Hence, a rather large distance to a single sequence can be compensated by less than average distances to other sequences. Apparently, this effect can be the stronger the larger the number of sequences becomes. In an evolutionary context it makes possibly more sense to limit the distance between each of the sequences and their common ancestor:

$$\max_{1 \le l \le k} \{D(C, S_l)\} \le \delta.$$

Such a set C is called a *center representative*.

The approach described in Sections 4 and 5 is compatible with this new distance threshold. Step 1 is modified such that we search for substrings with distance at most 2δ to each other sequence, and in Step 2 we compute the k-tuples according to this distance threshold. This threshold is stronger than the one for median gene clusters since the value of δ will be chosen relatively small compared to the one for the median representative because it refers to a single distance and not to the sum of k distances.

However, the crucial point is that in Step 3 median computation needs to be replaced by the computation of the center sequence, which is known to be NP-hard (Frances and Litman, 1997). There exist fixed-parameter algorithms that run in polynomial time for a fixed distance (Gramm et al., 2003; Ma and Sun, 2008), but these are of limited use for this application, because this computation has to be repeated for thousands of candidates generated in Step 2 of our method.

9. CONCLUSION

In this article, we introduced the concept of median gene clusters for the detection of approximate gene clusters in a set of k genomes based on gene order. We applied a filter method to narrow down the search space of potential clusters efficiently, allowing for fast detection of gene clusters in multiple genomes.

Our cluster model improves over *max-gap clusters* (Bergeron et al., 2002; He and Goldwasser, 2005) in two ways: The problem of low global cluster density reported in Hoberman and Durand (2005) does not arise as no fixed gap length needs to be specified. Unlike *max-gap clusters*, our method is capable of finding approximate clusters that contain genes that are missing in some cluster occurrences. This becomes important in particular for multiple genome comparison.

We also compared our method to an approach using an integer linear program for approximate gene cluster detection. While the underlying cluster models are similar, gene cluster computation was shown to be more efficient with our approach.

We believe that the main advantage of our method is its applicability to multiple genomes. Initial results show that the detection of gene clusters in multiple genomes is feasible, supporting our conjecture that the combinatorial explosion in Step 2 of our method does not occur with real-world data when parameters are

chosen reasonably. As the method is fastest when δ is small, we propose for practical applications to iteratively increase δ for some fixed s until clusters are detected that are potentially biologically meaningful.

Extending our method to detect median gene clusters that occur only in a subset of the input genomes is the natural next step of our work. Besides that, a study of the alternative cluster models introduced in Section 8 and the development of appropriate algorithms seems promising. For large-scale gene cluster prediction, the development of a more sophisticated statistical gene cluster ranking scheme is likely to become necessary.

REFERENCES

Amir, A., Gasieniec, L., and Shalom, R. 2007. Improved approximate common interval. *Inf. Process. Lett.* 103, 142–149.

Bergeron, A., Chauve, C., de Mongolfier, F., et al. 2005. Computing common intervals of k permutations, with applications to modular decomposition of graphs. *Proc. of ESA 2005*. 779–790.

Bergeron, A., Chauve, C., and Gingras, Y. 2007. Formal models of gene clusters. In: Mandoiu, I. and Zelikovsky, A., eds., *Bioinformatics Algorithms: Techniques and Applications*. Wiley, New York.

Bergeron, A., Corteel, S., and Raffinot, M. 2002. The algorithmic of gene teams. *Proc. of WABI 2002.* 464–476.

Böcker, S., Jahn, K., Mixtacki, J., et al. 2008. Computation of median gene clusters. *Proc. RECOMB 2008.* 331–345.

Chauve, C., Diekmann, Y., Heber, S., et al. 2006. On common intervals with errors [Report 2006–02]. Technische Fakultät der Universität Bielefeld, Abteilung Informationstechnik.

Dandekar, T., Snel, B., Huynen, M., et al. 1998. Conservation of gene order: a fingerprint of proteins that physically interact. *Trends Biochem. Sci*. 23, 324–328.

Didier, G., Schmidt, T., Stoye, J et al. 2007. Character sets of strings. *J. Discr. Alg.* 5, 330–340.

Frances, M., and Litman, A. 1997. On covering problems of codes. *Theor. Comput. Sci.* 30, 133–139.

Gramm, J., Niedermeier, R., and Rossmanith, P. 2003. Fixed-parameter algorithms for closest string and related problems. *Algorithmica* 37, 25–42.

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

Heber, S., and Stoye, J. 2001a. Algorithms for finding gene clusters. *Proc. of WABI 2001*. 252–263.

Heber, S., and Stoye, J. 2001b. Finding all common intervals of k permutations. *Proc. of CPM 2001*. 207–218.

Hoberman, R., and Durand, D. 2005. The incompatible desiderata of gene cluster properties. *Proc. of RCG 2005*. 73–87.

Ma, B., and Sun, X. 2008. More efficient algorithms for closest string and substring problems. *Proc. RECOMB 2008* 396–409.

Rahmann, S., and Klau, G. W. 2006. Integer linear programs for discovering approximate gene clusters. *Proc. of WABI 2006*. 298–309.

Schmidt, T., and Stoye, J. 2004. Quadratic time algorithms for finding common intervals in two and more sequences. *Proc. of CPM 2004*. 347–358.

Schmidt, T., and Stoye, J. 2007. Gecko and GhostFam—rigorous and efficient gene cluster detection in prokaryotic genomes, 165–182. In: Bergman, N., ed. *Comparative Genomics*. *Volume 396 of Methods in Molecular Biology*. Humana Press, Hillsdale, NJ.

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

Address correspondence to:
Katharina Jahn
Universität Bielefeld
Technische Fakultät
AG Genominformatik
D-33594 Bielefeld, Germany

E-mail: kjahn@cebitec.uni-bielefeld.de

(*Algorithms follow*)

Algorithm 1 Connecting intervals with errors (CIE)

build data structures *POS* and *NUM* for S_2
resultSet $\leftarrow \emptyset$
for $i = 1, \ldots, |S_1|$ **do**
  for each $c \in \Sigma$ **let** $OCC[c] \leftarrow 0$
  $|OCC| \leftarrow 0$
  minDist $\leftarrow 0$
  $j = i$
  while $j \leq |S_1|$ **and** $S_1(i, j)$ is left-maximal **do**
    $c \leftarrow S_1[j]$
    $OCC[c] \leftarrow 1$
    $|OCC| \leftarrow |OCC| + 1$
    while $S_1[i, j]$ is not right-maximal **do**
      $j \leftarrow j + 1$
    end while
    minDist $\leftarrow$ *minDist* $+ 1$
    for each position p in $POS[c]$ **do**
      mark position p in S_2
      find positions $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$
      for each pair (l_x, r_y) with $1 \leq x, y \leq \delta + 1$ **do**
        $z \leftarrow$ the number of different unmarked characters in $S_2[l_x + 1, r_y - 1]$
        dist $\leftarrow |OCC| - NUM[l_x + 1, r_y - 1] + 2z$
        if *dist* $<$ *minDist* **then**
          minDist $\leftarrow$ *dist*
        end if
      end for
    end for
    if *minDist* $\leq d$ **then**
      resultSet $\leftarrow$ *resultSet* $\cup$ (i, j))
    end if
    $j \leftarrow j + 1$
  end while
end for

Algorithm 2 Detection of all δ-locations of a cluster filter $C' = S_1[i, j]$ in S_2

for each $c \in C'$ **do**
  for each $pos \in POS\ [c]$ **do**
    $d_{left} = 0$
    for $l = pos, \ldots, L[pos][\delta + 1] + 1$ **do**
      if $S_2[l] \in C'$ **and** RANK $[S_2[l]] \geq$ RANK $[c]$ **and** $l \neq pos$ **then**
        break
      else if $S_2[l] \notin C'$ **and** $S_2[l] \notin \mathcal{CS}(S_2[l + 1, pos])$ **then**
        $d_{left} \leftarrow d_{left} + 1$
      end if
      $d_{total} = d_{left}$
      for $r = pos, \ldots, R[pos][\delta + 1] - 1$ **do**
        if $S_2[r] \in C'$ **and** RANK $[S_2[r]] >$ RANK$[c]$ **then**
          break
        else if $S_2[r] \notin C'$ **and** $S_2[r] \notin \mathcal{CS}(S_2[l, r - 1])$ **then**
          $d_{total} \leftarrow d_{total} + 1$
        end if
        if $dist = |C'| - NUM[l, r] + 2d_{total} < \delta$ **then**
          δ-locationSet[*dist*] $\leftarrow$ δ-locationSet[*dist*] $\cup$ (l, r)
        end if
      end for
    end for
  end for
end for