Computation of Median Gene Clusters

Sebastian Böcker[1], Katharina Jahn[2], Julia Mixtacki[2], and Jens Stoye[3]

[1] Institut für Informatik, Friedrich-Schiller-Universität Jena, Germany. boecker@minet.uni-jena.de
[2] International NRW Graduate School in Bioinformatics and Genome Research, Universität Bielefeld, Germany. {kjahn,mixtacki}@cebitec.uni-bielefeld.de
[3] Technische Fakultät, Universität Bielefeld, Germany. stoye@techfak.uni-bielefeld.de

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 paper we introduce a cluster concept named *median gene clusters* that improves over existing models and present efficient algorithms for their computation that allow for the detection of approximate gene clusters in multiple genomes.

1 Introduction and related work

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 [5]. 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, for example [2, 6, 10, 11, 14, 15].

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 search space of approximate gene cluster detection increases exponentially - depending on the cluster concept - either with the number of allowed deviations [4] or the number of compared sequences [9].

One approach to handle deviations of the gene content is by imposing constraints on the cluster locations: For example, *max-gap clusters* [3, 9] 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 [1] 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. 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 [13], 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 paper 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 paper (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 [9] and [13] and show its applicability to multiple genomes.

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, $1 \leq i \leq j \leq |S|$. We define the *character set* of a substring $S[i, j]$ of S as

$$\mathcal{CS}(S[i, j]) \;=\; \{S[m] \mid i \leq m \leq j\}.$$

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 by a terminal character $S[0] = S[n + 1] \notin \Sigma$. A substring $S[i, j]$ is *left-maximal* if $S[i - 1] \notin \mathcal{CS}(S[i, j])$, *right-maximal* if $S[j + 1] \notin \mathcal{CS}(S[i, j])$ and *maximal* if it is both left- and right-maximal.

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

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

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

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 $\sum_{l=1}^{k} D(C, C_l) \leq \sum_{l=1}^{k} D(C', C_l)$ 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 paper is the following.

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

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

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 [12]). Therefore, variants of the above problem formulation and additional cluster properties will also be discussed in the Appendix.

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 $S_1[i_1, j_1], ..., S_k[i_k, j_k]$ such that for a given $\delta \geq 0$ there exists a $C \subseteq \Sigma$ with $\sum_{l=1}^{k} D(C, CS(S[i_l, j_l])) \leq \delta$. 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 $S_1[i_1, j_1], \ldots, S_k[i_k, j_k]$ chose $S_m[i_m, j_m]$, $1 \leq m \leq k$, 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:

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

□

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 sets meet the conditions given by lemma 1.
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 at least one of the elements 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.

4 Computation of cluster filters (Step 1)

In k sequences of length at most n there are $O(kn^2)$ substrings. 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) [14] 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 algorithm

Algorithm CI, presented in [14], 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 such that entry $NUM[i, j]$ contains the number of different characters that occur in the substring $S_2[i, j]$. For an example, see Figure 1.

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

$NUM[i, j]$:

$i \backslash j$	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 $|\mathcal{CS}(S_2[i, j])|$.

The basic idea of the main algorithm is that while going systematically through all maximal substrings $S_1[i, j]$ of the first sequence, using the array POS one generates and iteratively extends marked intervals in the second sequence that consist only of characters occurring in the current interval $S_1[i, j]$.

Common intervals are detected 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

Algorithm 1 Connecting Intervals with Errors (CIE)

```
 1: build data structures POS and NUM for S_2
 2: resultSet ← ∅
 3: for i = 1, ..., |S_1| do
 4:    for each c ∈ Σ let OCC[c] ← 0
 5:    |OCC| ← 0
 6:    minDist ← 0
 7:    j = i
 8:    while j ≤ |S_1| and S_1(i, j) is left-maximal do
 9:       c ← S_1[j]
10:       OCC[c] ← 1
11:       |OCC| ← |OCC| + 1
12:       while S_1[i, j] is not right-maximal do
13:          j ← j + 1
14:       end while
15:       minDist ← minDist + 1
16:       for each position p in POS[c] do
17:          mark position p in S_2
18:          find positions l_1, ..., l_{δ+1} and r_1, ..., r_{δ+1}
19:          for each pair (l_x, r_y) with 1 ≤ x, y ≤ δ + 1 do
20:             z ← the number of different unmarked characters in S_2[l_x + 1, r_y − 1]
21:             dist ← |OCC| − NUM[l_x + 1, r_y − 1] + 2z
22:             if dist < minDist then
23:                minDist ← dist
24:             end if
25:          end for
26:       end for
27:       if minDist ≤ d then
28:          resultSet ← resultSet ∪ (i, j))
29:       end if
30:       j ← j + 1
31:    end while
32: end for
```

of a substring of S_1. Hence, each position in S_2 becomes marked at most $|S_1|$ times and each time extends one marked interval, or merges two intervals or constitutes a new marked interval if its neighbors are not yet marked. Thus there are at most $|S_1| \cdot |S_2|$ interval extensions and the same number of character set comparisons. In total this algorithm takes $O(n^2)$ time and $O(n^2)$ space.

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 S_2. Like in the Connecting Intervals algorithm for each latest character c in $S_1[i, j]$ we mark each position p where this character occurs in the other sequence (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:

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

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 $1 \le x, y \le \delta+1$. An example is illustrated in Fig 2.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
$S_2 =$	(1	3	5	2	4	5	2	4	3	6	2	5)
	l_2					l_1	p			r_1		r_2

Fig. 2. 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 $S_2[2,9]$, $S_2[2,11]$, $S_2[7,9]$ and $S_2[7,11]$.

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 $minDist$ 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 1. 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 runtime decreases to $O(n^2(1+\min\{n, b\delta\}+\delta^2))$. This is especially relevant for genetic sequences, 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 runtime 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 arrays 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 arrays L and R are given in the following. For simplicity, as in [6] 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 $\textsc{Rank} : \Sigma \to \{1, \ldots, |\Sigma|\}$. 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 $\textsc{Rank}(c') > \textsc{Rank}(c)$.

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 $\textsc{Rank}(S_2(p))$. 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 Fig. 3.

(a)

$\textsc{Rank}[1] = 4$
$\textsc{Rank}[2] = 1$
$\textsc{Rank}[3] = 3$
$\textsc{Rank}[4] = 2$
$\textsc{Rank}[5] = 5$
$\textsc{Rank}[6] = 6$

(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. 3. Initialization of (a) the rank for all characters and (b) the tables L and R. The characters of $S_1 = (2, 4, 2, 3, 4, 1, 4, 5, 4, 3, 6)$ 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'.

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 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 runtime 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 $S_l[i, j]$ 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 the triangle inequality of symmetric set distance. Hence, we need to examine for the most recently added character $S_l[j]$ all its occurrences in the other $k-1$ sequences and compute for each occurrence the distance of the corresponding $(\delta+1)^2$ intervals to $S_l[i, j]$. 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.

From these modifications it follows that the runtime multiplies by $O(k^2)$ for each of the presented algorithms while space requirements increase to $O(kn^2)$.

5 Collection of δ-locations of cluster filters (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 k-tuples $(S_1[i_1, j_1], \ldots, S_k[i_k, j_k])$ with pairwise intersecting character sets that satisfy

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

Maximal δ-locations of C' can be found efficiently by a modified version of Algorithm 1 that iterates through a location of C' and generates uniquely all maximal δ-locations. Details of the algorithm are left to the reader.

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 $S_l = (abcx_l)^n$, $1 \leq l \leq k$ and $x_i \neq x_j$ for $i \neq j$, there are $O(n^k)$ k-tuples. However, for gene sequences where $|\Sigma|$ is in $\Theta(n)$ our experience shows that this approach is feasible for reasonable values of δ.

6 Computation of median gene clusters from k-tuples (Step 3)

The computation of the median of a k-tuple consists of a simple majority vote of the characters occurring as its elements, i.e. a gene occurring in at least half of the tuple elements becomes an element of the median. The median of each k-tuple is checked whether it fulfills inequality (1) and in case it fulfills the distance constraint it is reported as a median gene cluster.

Note that there can be several medians that have to be tested: If k is even, there may be ties when some character occurs in exactly $k/2$ of the elements. However, since each tie adds $k/2$ to

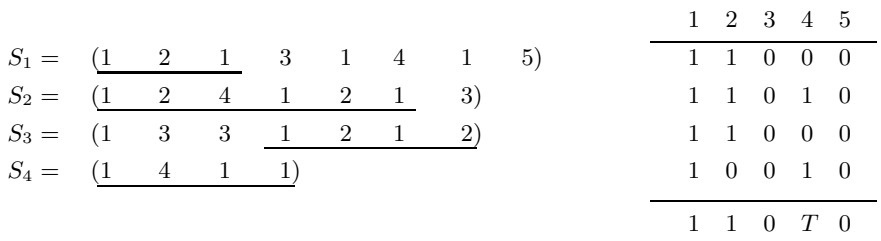

Fig. 4. For $\delta = 3$ and the cluster filter $C' = \{1, 2\}$, one of its 3-locations in each of the sequences is given by the underlined substrings. The tie of the fourth character (denoted by T) yields the two medians $\{1, 2\}$ and $\{1, 2, 4\}$.

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 Fig. 4 illustrates.

Moreover it can happen that the same character set is generated more than once either by duplicate k-tuples 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. In our implementation we filter away such multiple occurrences.

7 Experimental results

In an initial test we compared the performance of our two algorithms 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 [9] 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 1070 median gene clusters in this dataset, among them the ten operons studied in [9]. These findings show that our method finds a superset of the gene clusters detected by the `HomologyTeams` software. A biological evaluation of the additional gene clusters is currently in progress.

7.2 Comparison to ILP approach

We downloaded the genome datasets from `http://gi.cebitec.uni-bielefeld.de/comet` used in [13]. The dataset consists of the annotated genomes of *C. glutamicum* and *M. tuberculosis* where labeling of genes according to gene family membership already took place.

Our program found the gene cluster reported in [13] in 17 seconds using appropriate values for the parameters δ and s while the ILP using CPLEX 9.03 took more than one hour on a superior processor. 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 [13] 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 hrs and 4 min.

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 three bacterial genomes: *Bacillus subtilis*, *Buchnera aphidicola* and *Escherichia coli* with different combinations of s and δ. Results are shown in Table 1.

	s=10, δ=0	s=20, δ=5	s=25, δ=10	s=30, δ=15	s = 35, δ=25
distinct medians	91	152	101	21	1
computation time in sec.	4.7	7.2	12.0	26.1	186.2

Table 1. The number of distinct median gene clusters found for different combinations of s and δ in three bacterial genomes and the corresponding computation times.

Many of the gene clusters found in this experiment belong to well-conserved ribosomal protein operons. In order to find other gene clusters, genes associated with ribosomal proteins were masked in the genomes for an additional test. Distance thresholds needed to be chosen larger for a fixed s in this setting in order to find gene clusters. For example, with $s = 13$ and $\delta = 10$, we found five distinct gene clusters, among them the following gene cluster involved in flagellar biosynthesis:

Fig. 5. A gene cluster involved in flagellar biosythesis, detected by our method with parameters set to $\delta = 10$ and $s = 13$.

8 Conclusion

In this paper 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* [3, 9] in two ways: The problem of low global cluster density reported in [12] 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 ILP 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. A broader analysis of the influence of s and δ on sensitivity, specificity, and running time of our method is currently in progress. 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.

In the future, we want to extend our method to detect median gene clusters that occur only in a subset of the input genomes. We also want to provide a statistical analysis of the detected clusters to rank the reported clusters according to their significance.

References

1. Amihood Amir, Leszek Gasieniec, and Riva Shalom. Improved approximate common interval. *Inf. Process. Lett.*, 103(4):142–149, 2007.
2. A. Bergeron, C. Chauve, F. de Mongolfier, and M. Raffinot. Computing common intervals of k permutations, with applications to modular decomposition of graphs. In *Proceedings of ESA 2005*, volume 3669 of *LNCS*, pages 779–790, 2005.
3. 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.
4. C. Chauve, Y. Diekmann, S. Heber, J. Mixtacki, S. Rahmann, and J. Stoye. On common intervals with errors. Report 2006-02, Technische Fakultät der Universität Bielefeld, Abteilung Informationstechnik, 2006.
5. T. Dandekar, B. Snel, M. Huynen, and P. Bork. Conservation of gene order: a fingerprint of proteins that physically interact. *Trends Biochem Sci*, 23(9):324–8, 1998.
6. G. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. *J. Discr. Alg.*, 5:330–340, 2007.
7. M. Frances and A. Litman. On covering problems of codes. *Theor. Comput. Sci.*, 30:133–119, 1997.
8. J. Gramm, R. Niedermeier, and P. Rossmanith. Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, 37:25–42, 2003.
9. X. He and M. H. Goldwasser. Identifying conserved gene clusters in the presence of homology families. *J. Comp. Biol.*, 12:638–656, 2005.
10. S. Heber and J. Stoye. Algorithms for finding gene clusters. In *Proceedings of WABI 2001*, volume 2149 of *LNCS*, pages 252–263, 2001.
11. S. Heber and J. Stoye. Finding all common intervals of k permutations. In *Proceedings of CPM 2001*, volume 2089 of *LNCS*, pages 207–218, 2001.
12. R. Hoberman and D. Durand. The incompatible desiderata of gene cluster properties. In *Proceedings of RCG 2005*, volume 3678 of *LNBI*, pages 73–87, 2005.
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*, 26:290–309, 2000.

Appendix: Alternatives

Some variations of the model described in the main part of this paper are discussed in the following.

Transformation set distance

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

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

This distance measure is called the *transformation set distance* between C and C' and is also 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.

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:

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

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 [7]. There exist fixed-parameter algorithms that run in polynomial time for a fixed distance [8], but these are of limited use for this application.