Chaining algorithms for multiple genome comparison

Mohamed Ibrahim Abouelhoda, Enno Ohlebusch

Faculty of Computer Science, University of Ulm, 89069 Ulm, Germany.
E-mail: {mibrahim,eo}@informatik.uni-ulm.de

Abstract

Given n fragments from $k > 2$ genomes, Myers and Miller showed how to find an optimal global chain of colinear non-overlapping fragments in $O(n \log^k n)$ time and $O(n \log^{k-1} n)$ space. For gap costs in the L_1-metric, we reduce the time complexity of their algorithm by a factor $\frac{\log^2 n}{\log \log n}$ and the space complexity by a factor $\log n$. For the sum-of-pairs gap cost, our algorithm improves the time complexity of their algorithm by a factor $\frac{\log n}{\log \log n}$. A variant of our algorithm finds all significant local chains of colinear non-overlapping fragments. These chaining algorithms can be used in a variety of problems in comparative genomics: the computation of global alignments of complete genomes, the identification of regions of similarity (candidate regions of conserved synteny), the detection of genome rearrangements, and exon prediction.

Key words: fragment-chaining algorithms, multiple alignment, comparative genomics, range maximum query

1 Introduction

Given the continuing improvements in high-throughput genomic sequencing and the ever-expanding biological sequence databases, new advances in software tools for post-sequencing functional analysis are being demanded by the biological scientific community. Whole genome comparisons have been heralded as the next logical step toward solving genomic puzzles, such as determining coding regions, discovering regulatory signals, and deducing the mechanisms and history of genome evolution. However, before any such detailed analysis can be addressed, methods are required for comparing such large sequences. If the organisms under consideration are closely related (that is, if no or only a few genome rearrangements have occurred) or one compares regions of conserved synteny (regions in which orthologous genes occur in the same

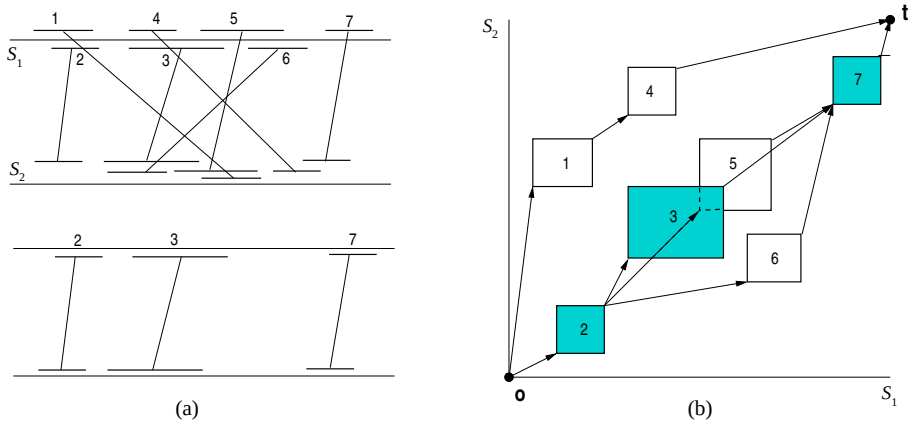

Fig. 1. Given a set of fragments (upper left figure), an optimal global chain of colinear non-overlapping fragments (lower left figure) can be computed, e.g., by computing an optimal path in the graph in (b) (in which not all edges are shown). How the graph can be constructed from the fragments is explained in Section 2.

order), then global alignments can be used for the prediction of genes and regulatory elements. This is because coding regions are relatively well preserved, while non-coding regions tend to show varying degree of conservation. Non-coding regions that do show conservation are thought important for regulating gene expression, maintaining the structural organization of the genome and possibly have other, yet unknown functions. Several comparative sequence approaches using alignments have recently been used to analyze corresponding coding and non-coding regions from different species, although mainly between human and mouse. These approaches are based on software-tools for aligning DNA-sequences [5, 7, 8, 11, 12, 18, 19, 21, 27]; see [9] for a review.

To cope with the shear volume of data, most of the software-tools use an anchor-based method that is composed of three phases:

(1) Computation of fragments (regions in the genomes that are similar).
(2) Computation of an optimal global chain of colinear non-overlapping fragments: these are the anchors that form the basis of the alignment.
(3) Alignment of the regions between the anchors.

The fragments computed in the first phase are often exact matches (maximal unique matches as in MUMmer [11, 12], maximal multiple exact matches as in MGA [18], or exact k-mers as in GLASS [5]), but one may also allow substitutions (yielding fragments as in DIALIGN [21]) or even insertions and deletions (as the BLASTZ-hits [26] that are used in MultiPipMaker [27]). Each of the fragments has a weight that can, for example, be the length of the fragment (in case of exact fragments) or its statistical significance. This article is concerned with algorithms for solving the combinatorial chaining problem of the second phase: finding an optimal (i.e., maximum weight) global chain of colinear non-overlapping fragments; see Fig. 1(a). Note that every genome

alignment tool has to solve the chaining problem somehow, but the algorithms differ from tool to tool.

A well-known solution to the chaining problem consists of finding a maximum weight path in a weighted directed acyclic graph. However, the running time of this chaining algorithm is quadratic in the number n of fragments. This can be a serious drawback if n is large. To overcome this obstacle, Zhang et al. [30] presented an algorithm that constructs an optimal chain using space division based on *kd*-trees, a data structure known from computational geometry [6]. However, a rigorous analysis of the running time of their algorithm is difficult because the construction of the chain is embedded in the *kd*-tree structure. Another chaining algorithm, devised by Myers and Miller [23], is based on the *line-sweep* paradigm, and uses *orthogonal range-searching* supported by *range trees* instead of *kd*-trees. It is the only chaining algorithm for $k > 2$ sequences that runs in sub-quadratic time $O(n \log^k n)$, but the result is a time bound higher by a logarithmic factor than what one would expect. In particular, for $k = 2$ sequences it is one log-factor slower than previous chaining algorithms [13, 22], which require only $O(n \log n)$ time. In the epilogue of their paper [23], Myers and Miller wrote: "We thought hard about trying to reduce this discrepancy but have been unable to do so, and the reasons appear to be fundamental" and "To improve upon our result appears to be a difficult open problem." In this article, we will improve their result. For gap costs in the L_1-metric, we can not only reduce the time and space complexities of Myers and Miller's algorithm by a log-factor but actually improve the time complexity by a factor $\frac{\log^2 n}{\log \log n}$. For the sum-of-pairs gap cost, our method yields a reduction by a factor $\frac{\log n}{\log \log n}$. In essence, this improvement is achieved by enhancing the ordinary range tree with (1) the combination of fractional cascading [29] and the efficient priority queues of [17, 28], which yields a more efficient search, and (2) an appropriate incorporation of gap costs, so that it is enough to determine a maximum function value over a semi-dynamic set (instead of a dynamic set). We would like to point out that our algorithm can also employ other data structures that support orthogonal range-searching. For example, if the *kd*-tree is used instead of the range tree, the algorithm takes $O((k-1)n^{2-\frac{1}{k-1}})$ time and $O(n)$ space in the worst case, for $k > 2$ and gap costs in the L_1-metric.

As already mentioned, global alignments are a valuable tool for comparing the genomes of closely related species. In case of diverged genomic sequences, however, genome rearrangements have occurred during evolution, so that a global alignment strategy is likely predestined to failure for having to align unrelated regions in an end-to-end colinear approach. In this case, either local alignments are the strategy of choice or one must first identify syntenic regions, which then can be individually aligned. However, both alternatives are faced with obstacles. Current local alignment programs suffer from a huge running

time, while the problem of automatically finding syntenic regions requires a priori knowledge of all genes and their locations in the genomes—a piece of information that is often not available. (It is beyond the scope of this paper to discuss the computational difficulties of gene prediction and the accurate determination of orthologous genes.) We will show that a local variant of our global chaining algorithm can be used to solve the problem of automatically finding regions of similarity in large genomic DNA sequences. As in the anchor-based global alignment method, one first computes fragments. In the second phase, however, instead of computing an optimal global chain of colinear non-overlapping fragments, one computes significant local chains. We call a local chain significant if its score (the sum of the weights of the fragments in the chain, where gaps between the fragments are penalized) exceeds a user-defined threshold. The computation of significant local chains can be done in the same time and space complexities as above-mentioned (for solving the global fragment-chaining problem).

Under stringent thresholds, significant local chains of colinear non-overlapping fragments represent candidate regions of conserved synteny. If one aligns these individually, one gets good local alignments. We would like to point out that the automatic identification of regions of similarity is a first step toward an automatic detection of genome rearrangements. In [2], we have demonstrated that our method can be used to detect genome rearrangements such as transpositions (where a section of the genome is excised and inserted at a new position in the genome, without changing orientation) and inversions (where a section of the genome is excised, reversed in orientation, and re-inserted).

We see several advantages of our chaining algorithms:

(1) They can handle multiple genomes.
(2) They can handle any kind of fragments.
(3) In contrast to most other algorithms used in comparative genomics, they are not heuristic: They correctly solve clearly defined problems.

It is worth mentioning that chaining algorithms are also useful in other bioinformatics applications such as comparing restriction maps [22] or solving the exon assembly problem which is part of eucaryotic gene prediction [15]. A variety of applications in comparative genomics is described in [24].

This article is organized as follows. Section 2 defines the basic concepts dealt with. In Section 3, we will explain a global chaining algorithm that neglects gap costs. In Section 4, the algorithm is modified in two steps, so that it can deal with certain gap costs. Section 5 presents a local variant of the global chaining algorithm that can be used for finding regions of similarity in large genomic DNA sequences. Finally, Section 6 summarizes the main achievements of our work.

Parts of this article appeared in [1] and [2].

2 Basic concepts and definitions

For $1 \leq i \leq k$, $S_i = S_i[1 \ldots n_i]$ denotes a string of length $|S_i| = n_i$. In our application, S_i is a (long) DNA sequence. $S_i[l \ldots h]$ is the substring of S_i starting at position l and ending at position h. A *fragment* f consists of two k-tuples $beg(f) = (l_1, l_2, \ldots, l_k)$ and $end(f) = (h_1, h_2, \ldots, h_k)$ such that the strings $S_1[l_1 \ldots h_1]$, $S_2[l_2 \ldots h_2], \ldots, S_k[l_k \ldots h_k]$ are "similar". Furthermore, we speak of an *exact fragment* (or *multiple exact match*) if the substrings are identical, i.e., $S_1[l_1 \ldots h_1] = S_2[l_2 \ldots h_2] = \ldots = S_k[l_k \ldots h_k]$. An exact fragment is maximal, if the substrings can neither be simultaneously extended to the left nor to the right in all S_i.

A fragment f of k genomes can be represented by a hyper-rectangle in $\mathbb{R}^k$ with the two extreme corner points $beg(f)$ and $end(f)$, where each coordinate of the points is a non-negative integer (consequently, the words *number of genomes* and *dimension* will be used synonymously in what follows). A hyper-rectangle (called hyperrectangular domain in [25]) is the Cartesian product of intervals on distinct coordinate axes. A hyper-rectangle $[l_1 \ldots h_1] \times [l_2 \ldots h_2] \times \ldots \times [l_k \ldots h_k]$ (with $l_i < h_i$ for all $1 \leq i \leq k$) will here be denoted by $R(p, q)$, where $p = (l_1, \ldots, l_k)$ and $q = (h_1, \ldots, h_k)$.

With every fragment f, we associate a positive weight $f.weight \in \mathbb{R}$. This weight can, for example, be the length of the fragment (in case of exact fragments) or its statistical significance.

In what follows, we will often identify the point $beg(f)$ or $end(f)$ with the fragment f. This is possible because we assume that all fragments are known from the first phase of the anchored-based approach described in Section 1 (so that every point can be annotated with a tag that identifies the fragment it stems from). For example, if we speak about the score of a point $beg(f)$ or $end(f)$, we mean the score of the fragment f.

For ease of presentation, we consider the points $\mathsf{0} = (0, \ldots, 0)$ (the origin) and $\mathsf{t} = (|S_1|+1, \ldots, |S_k|+1)$ (the terminus) as fragments with weight 0. For these fragments, we define $beg(\mathsf{0}) = \perp$, $end(\mathsf{0}) = \mathsf{0}$, $beg(\mathsf{t}) = \mathsf{t}$, and $end(\mathsf{t}) = \perp$.

The coordinates of a point $p \in \mathbb{R}^k$ will be denoted by $p.x_1, p.x_2, \ldots, p.x_k$. If $k = 2$, the coordinates of p will also be written as $p.x$ and $p.y$.

Definition 2.1 We define a binary relation $\ll$ on the set of fragments by $f \ll f'$ if and only if $end(f).x_i < beg(f').x_i$ for all $1 \leq i \leq k$. If $f \ll f'$, then

we say that f *precedes* f'.

Note that $\mathsf{0} \ll f \ll \mathsf{t}$ for every fragment f with $f \neq \mathsf{0}$ and $f \neq \mathsf{t}$.

Definition 2.2 A *chain* of colinear non-overlapping fragments (or chain for short) is a sequence of fragments $f_1, f_2, \ldots, f_\ell$ such that $f_i \ll f_{i+1}$ for all $1 \leq i < \ell$. The *score* of C is

$$score(C) = \sum_{i=1}^{\ell} f_i.weight - \sum_{i=1}^{\ell-1} g(f_{i+1}, f_i)$$

where $g(f_{i+1}, f_i)$ is the cost of connecting fragment f_i to f_{i+1} in the chain. We will call this cost *gap cost*.

Definition 2.3 Given n weighted fragments and a gap cost function, the *global fragment-chaining problem* is to determine a chain of maximum score (called optimal global chain in the following) starting at the origin $\mathsf{0}$ and ending at terminus t.

A direct solution to this problem is to construct a weighted directed acyclic graph $G = (V, E)$, where the set V of vertices consists of all fragments (including $\mathsf{0}$ and t) and the set of edges E is characterized as follows: There is an edge $f \to f'$ with weight $f'.weight - g(f', f)$ if $f \ll f'$. See Fig. 1(b) for an example (where edges $f \to f'$ are omitted whenever there is a fragment $\tilde{f}$ such that $f \ll \tilde{f} \ll f'$). An optimal global chain of fragments corresponds to a path with maximum score from vertex $\mathsf{0}$ to vertex t in the graph. Because the graph is acyclic, such a path can be computed as follows. Let $f'.score$ be defined as the maximum score of all chains starting at $\mathsf{0}$ and ending at f'. $f'.score$ can be expressed by the recurrence: $\mathsf{0}.score = 0$ and

$$f'.score = f'.weight + \max\{f.score - g(f', f) : f \ll f'\} \tag{1}$$

A dynamic programming algorithm based on this recurrence takes $O(|V| + |E|)$ time provided that computing *gap costs* takes constant time. Because $|V| + |E| \in O(n^2)$, computing an optimal global chain takes quadratic time and linear space. This graph-based solution works for any number of genomes and for any kind of gap cost. As explained in Section 1, however, the time bound can be improved by considering the geometric nature of the problem. In order to present our result systematically, we first give a chaining algorithm that neglects gap costs. Then we will modify this algorithm in two steps, so that it can deal with certain gap costs.

3 The global chaining algorithm without gap costs

3.1 The basic chaining algorithm

Given a set S of points in $\mathbb{R}^k$ with associated score, a *range query* $\texttt{RQ}(p, q)$ asks for all points of S that lie in the hyper-rectangle $R(p, q)$, while a *range maximum query* $\texttt{RMQ}(p, q)$ asks for a point of maximum score in $R(p, q)$.

Lemma 3.1 *Suppose that the gap cost function is the constant function* 0*. If* $\texttt{RMQ}(\texttt{0}, beg(f') - \vec{1})$ *returns the end point of fragment* f *(where* $\vec{1}$ *denotes the vector* $(1, \ldots, 1)$*),then* $f'.score = f'.weight + f.score$*.*

Proof This follows immediately from recurrence (1). □

Suppose that the start and end points of the fragments are sorted w.r.t. their x_1 coordinate. Then, processing the points in ascending order of their x_1 coordinate simulates a line (plane or hyper-plane in higher dimensions) that sweeps the points w.r.t. their x_1 coordinate. Here, we will use this so-called *line-sweep* technique to construct an optimal chain. If a point has already been scanned by the sweeping line, it is said to be *active*; otherwise it is said to be *inactive*. During the sweeping process, the x_1 coordinates of the active points are smaller than the x_1 coordinate of the currently scanned point s. According to Lemma 3.1, if s is the start point of fragment f', then an optimal chain ending at f' can be found by an RMQ over the set of active end points of fragments. Since $p.x_1 < s.x_1$ for every active end point p, the RMQ need not take the first coordinate into account. In other words, the RMQ is confined to the range $R(\texttt{0}, (s.x_2, \ldots, s.x_k) - \vec{1})$ in $\mathbb{R}^{k-1}$, so that the dimension of the problem is reduced by one. To manipulate the point set during the sweeping process, we need a semi-dynamic data structure D that stores the end points of fragments and efficiently supports the following two operations: (1) activation and (2) RMQ over the set of active points. Algorithm 1 is based on such a data structure D, which will be defined later. In the algorithm, $f'.prec$ denotes a field that stores the preceding fragment of f' in a chain. It is an immediate consequence of Lemma 3.1 that Algorithm 1 finds an optimal chain. The complexity of the algorithm depends of course on how the data structure D is implemented. In the following subsection, we will outline an implementation of D that supports RMQs with activation in time $O(n \log^{d-1} n \log\log n)$ and space $O(n \log^{d-1} n)$ for n points, where d is the dimension. Because in our chaining problem $d = k-1$, finding an optimal chain by Algorithm 1 takes $O(n \log^{k-2} n \log\log n)$ time and $O(n \log^{k-2} n)$ space.

Algorithm 1. (k-dimensional chaining of n fragments)
Sort all start and end points of the n fragments in ascending order w.r.t. their x_1 coordinate and store them in the array points; because we include the end point of the origin and the start point of the terminus, there are $2n+2$ points. Store all end points of the fragments (ignoring their x_1 coordinate) as inactive in the $(k-1)$-dimensional data structure D.
for $i := 1$ **to** $2n+2$
  if points$[i]$ is the start point of fragment f' **then**
    $q :=$ RMQ$(0, (\mathsf{points}[i].x_2, \ldots, \mathsf{points}[i].x_k) - \vec{1})$
    determine the fragment f with $end(f) = q$
    $f'.prec := f$
    $f'.score := f'.weight + f.score$
  else /⋆ points$[i]$ is end point of a fragment ⋆/
    activate $(\mathsf{points}[i].x_2, \ldots, \mathsf{points}[i].x_k)$ in D

3.2 Answering RMQs with activation efficiently

Our orthogonal range-searching data structure is based on *range trees*, which are well-known in computational geometry. Given a set S of n d-dimensional points, its range tree can be built as follows (see, e.g., [4, 25]). For $d = 1$, the range tree of S is a minimum-height binary search tree or an array storing S in sorted order. For $d > 1$, the range tree of S is a minimum-height binary search tree T with n leaves, whose ith leftmost leaf stores the point in S with the ith smallest x_1-coordinate. To each interior node v of T, we associate a canonical subset $C_v \subseteq S$ containing the points stored at the leaves of the subtree rooted at v. For each v, let l_v (resp. h_v) be the smallest (resp. largest) x_1 coordinate of any point in C_v and let $C_v^* = \{(p.x_2, \ldots, p.x_d) \in \mathbb{R}^{d-1} \mid (p.x_1, p.x_2, \ldots, p.x_d) \in C_v\}$. The interior node v stores l_v, h_v, and a $(d-1)$-dimensional range tree constructed on C_v^*. For any fixed dimension d, the data structure can be built in $O(n \log^{d-1} n)$ time and space. A range query RQ(p, q) for the hyper-rectangle $R(p, q) = [l_1 \ldots h_1] \times [l_2 \ldots h_2] \times \ldots \times [l_d \ldots h_d]$ can be answered as follows. If $d = 1$, the query can be answered in $O(\log n)$ time by a binary search. For $d > 1$, we traverse the range tree starting at the root. Suppose node v is visited in the traversal. If v is a leaf, then we report its corresponding point if it lies inside $R(p, q)$. If v is an interior node, and the interval $[l_v, h_v]$ does not intersect $[l_1, h_1]$, there is nothing to do. If $[l_v, h_v] \subseteq [l_1, h_1]$, we recursively search in the $(d-1)$-dimensional range tree stored at v with the hyper-rectangle $[l_2 \ldots h_2] \times \ldots \times [l_d \ldots h_d]$. Otherwise, we recursively visit both children of v. This procedure takes $O(\log^d n + z)$ time, where z is the number of points in the hyper-rectangle $R(p, q)$.

The technique of *fractional cascading* [29] saves one log-factor in answering range queries (in the same construction time and using the same space as the original range tree). Here, we will recall this technique for range queries of

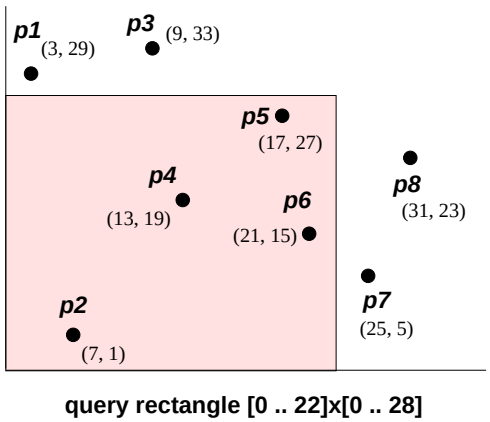

Fig. 2. A set of points and a query rectangle.

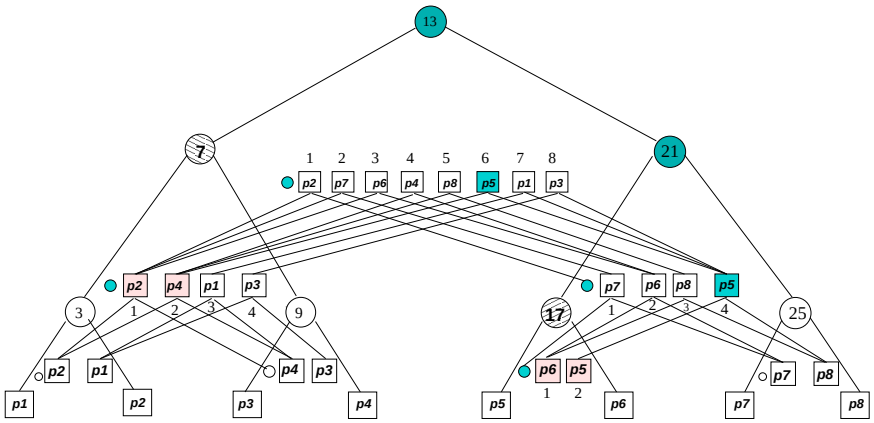

Fig. 3. Range tree with fractional cascading for the points in Fig. 2. The colored nodes are visited for answering the range query of Fig. 2. Hatched nodes are the canonical nodes. The small circles refer to NULL pointers. In this example, $p_{h_1} = p_6$ and $p_{h_2} = p_5$. The colored elements of the *y-arrays* of the canonical nodes are the points in the query rectangle of Fig. 2. The value $h_{v.L}$ in every internal node v is the x coordinate that separates the points in its left subtree from those occurring in its right subtree.

the form $\texttt{RQ}(0, q)$ because we want to modify it to answer range maximum queries of the form $\texttt{RMQ}(0, q)$ efficiently. For ease of presentation, we consider the case $d = 2$. In this case, the range tree is a binary search tree (called *x-tree*) of arrays (called *y-arrays*). Let v be a node in the *x-tree* and let $v.L$ and $v.R$ be its left and right children. The *y-array* A_v of v contains all the points in C_v sorted in ascending order w.r.t. their y coordinate. Every element $p \in A_v$ has two downstream pointers: The left pointer *Lptr* and the right pointer *Rptr*. The left pointer *Lptr* points to the largest (i.e., rightmost) element q_1 in $A_{v.L}$ such that $q_1 \leq p$ (*Lptr* is a NULL pointer if such an element does not exist). In an implementation, *Lptr* is the index with $A_{v.L}[Lptr] = q_1$. Analogously, the right pointer *Rptr* points to the largest element q_2 of $A_{v.R}$ such that $q_2 \leq p$. Fig. 3 shows an example of this structure. Locating all the points in a rectangle $R(0, (h_1, h_2))$ is done in two stages. In the first stage,

a binary search is performed over the *y-array* of the root node of the *x-tree* to locate the rightmost point p_{h_2} such that $p_{h_2}.y \in [0 \dots h_2]$. Then, in the second stage, the *x-tree* is traversed (while keeping track of the downstream pointers) to locate the rightmost leaf p_{h_1} such that $p_{h_1}.x \in [0 \dots h_1]$. During the traversal of the *x-tree*, we identify a set of nodes which we call *canonical nodes* (w.r.t. the given range query). The set of canonical nodes is the smallest set of nodes $v_1, \dots, v_\ell \in$ *x-tree* such that $\uplus_{j=1}^{\ell} C_{v_j} = \texttt{RQ}(\texttt{0}, (h_1, \infty))$. ($\uplus$ denotes disjoint union.) In other words, $P := \uplus_{j=1}^{\ell} A_{v_j} = \uplus_{j=1}^{\ell} C_{v_j}$ contains every point $p \in S$ such that $p.x \in [0 \dots h_1]$. However, not every point $p \in P$ satisfies $p.y \in [0 \dots h_2]$. Here, the downstream pointers come into play. As already mentioned, the downstream pointers are followed while traversing the *x-tree*, and to follow one pointer takes constant time. If we encounter a canonical node v_j, then the element e_j, to which the last downstream pointer points, partitions the list A_{v_j} as follows: Every e that is strictly to the right of e_j is not in $R(\texttt{0}, (h_1, h_2))$, whereas all other elements of A_{v_j} lie in $R(\texttt{0}, (h_1, h_2))$. For this reason, we will call the element e_j the *split* element. It is easy to see that the number of canonical nodes is $O(\log n)$. Moreover, we can find all of them and the split elements of their *y-arrays* in $O(\log n)$ time; cf. [29]. Therefore, the range tree with fractional cascading supports 2-dimensional range queries in $O(\log n + z)$ time, where z is the number of points in the rectangle $R(\texttt{0}, q)$. For dimension $d > 2$, it takes time $O(\log^{d-1} n + z)$.

In order to answer RMQs with activation, we will further enhance every *y-array* that occurs in the fractional cascading data structure with a priority queue as described in [17, 28]. Each of these queues is (implicitly) constructed over the *rank space*[1] of the points in the *y-array*. The rank space of the points in the *y-array* consists of points in the range $[0 \dots m]$, where m is the size of the *y-array*, and the rank of a point is its index in the *y-array* because the *y-array* is sorted w.r.t. the y dimension.

The priority queue supports the operations $insert(r)$, $delete(r)$, $predecessor(r)$ (gives the largest element $\leq r$), and $successor(r)$ (gives the smallest element $> r$) in time $O(\log \log m)$, where r is an integer in the range $[0 \dots m]$. Algorithm 2 shows how to activate a point q in the 2-dimensional range tree and Algorithm 3 shows how to answers an $\texttt{RMQ}(0, q)$.

Note that in the outer while-loop of Algorithm 2, the following invariant is maintained: If $0 \leq i_1 < i_2 < \ldots < i_\ell \leq m$ are the entries in the priority queue attached to A_v, then $A_v[i_1].score \leq A_v[i_2].score \leq \ldots \leq A_v[i_\ell].score$.

Algorithm 3 gives pseudo-code for answering $\texttt{RMQ}(\texttt{0}, q)$, but we would like to first describe the idea on a higher level. In essence, we locate all canonical nodes $v_1, \dots, v_\ell$ in D for the hyper-rectangle $R(\texttt{0}, q)$. For any v_j, $1 \leq j \leq \ell$, let the

[1] See, e.g., [10, 14] for more details on the rank space.

Algorithm 2. (Activation of a point q in the data structure D)

$v :=$ root node of the *x-tree*
find the rank (*index*) r of q in A_v by a binary search
while $(v \neq \perp)$
  if $(A_v[r].score > A_v[predecessor(r)].score)$ **then**
    insert(r) into the priority queue attached to A_v
    while$(A_v[r].score > A_v[successor(r)].score)$
      delete$(successor(r))$ from the priority queue attached to A_v
  if $(A_v[r] = A_{v.L}[A_v[r].Lptr])$ **then**
    $r := A_v[r].Lptr$
    $v := v.L$
  else
    $r := A_v[r].Rptr$
    $v := v.R$

Algorithm 3. ($\texttt{RMQ}(\texttt{0}, q)$ in the data structure D)

$v :=$ root node of the *x-tree*
$max\_score := -\infty$
$max\_point := \perp$
find the rank (*index*) r of the rightmost point p with $p.y \in [0 \ldots q.y]$ in A_v
while $(v \neq \perp)$
  if $(h_v.x \leq q.x)$ **then** /$\star$ v is a canonical node $\star$/
    $tmp := predecessor(r)$ in the priority queue of A_v
    $max\_score := \max\{max\_score, A_v[tmp].score\}$
    if $(max\_score = tmp.score)$ **then** $max\_point := A_v[tmp]$
  else if $(h_{v.L}.x \leq q.x)$ **then** /$\star$ $v.L$ is a canonical node $\star$/
    $tmp := predecessor(A_v[r].Lptr)$ in the priority queue of $A_{v.L}$
    $max\_score := \max\{max\_score, A_{v.L}[tmp].score\}$
    if $(max\_score = tmp.score)$ **then** $max\_point := A_{v.L}[tmp]$
    $r := A_v[r].Rptr$
    $v := v.R$
  else
    $r := A_v[r].Lptr$
    $v := v.L$

r_jth element be the split element in A_{v_j}. We have seen that $\uplus_{j=1}^{\ell} A_{v_j}$ contains every point $p \in S$ such that $p.x \in [0 \ldots q.x]$. Now if r_j is the index of the split element of A_{v_j}, then all points $A_{v_j}[i]$ with $i \leq r_j$ are in $R(\texttt{0}, q)$, whereas all other elements $A_{v_j}[i]$ with $i > r_j$ are not in $R(\texttt{0}, q)$. Since Algorithm 2 maintains the above-mentioned invariant, the element with highest score in the priority queue of A_{v_j} that lies in $R(\texttt{0}, q)$ is $q_j = predecessor(r_j)$ (if r_j is in the priority queue of A_{v_j}, then $q_j = r_j$ because $predecessor(r_j)$ gives the largest element $\leq r_j$). We then compute $max\_score := \max\{A_{v_j}[q_j].score \mid 1 \leq j \leq \ell\}$ and return $max\_point = A_{v_i}[q_i]$, where $A_{v_i}[q_i].score = max\_score$.

Because the number of canonical nodes is $O(\log n)$ and any of the priority queue operations takes $O(\log \log n)$ time, answering a 2-dimensional range maximum query takes $O(\log n \log \log n)$ time. Since every point occurs in at most $\log n$ priority queues, there are at most $n \log n$ delete operations. Hence the total time complexity of activating n points is $O(n \log n \log \log n)$.

Theorem 3.2 *Given $k > 2$ genomes and n fragments, an optimal global chain (without gap costs) can be found in $O(n \log^{k-2} n \log \log n)$ time and $O(n \log^{k-2} n)$ space.*

Proof In Algorithm 1, the points are first sorted w.r.t. their first dimension and the `RMQ` with activation is required only for $d = k - 1$ dimensions. For $d \geq 2$ dimensions, the preceding data structure is implemented for the last two dimensions of the range tree, which yields a data structure D that requires $O(n \log^{d-1} n)$ space and $O(n \log^{d-1} n \log \log n)$ time for n `RMQ`s and n activation operations. Consequently, one can find an optimal chain in $O(n \log^{k-2} n \log \log n)$ time and $O(n \log^{k-2} n)$ space. □

In case $k = 2$, the data structure D is simply a priority queue (over the rank space of all points). Therefore, if the points are already sorted, then the algorithm takes $O(n \log \log n)$ time. Otherwise, the sorting procedure in Algorithm 1 dominates the overall time complexity of Algorithm 1 because it requires $O(n \log n)$ time.

4 Incorporating gap costs

In the previous section, fragments were chained without penalizing the gaps in between them. In this section, we modify the algorithm, so that it can take gap costs into account.

4.1 Gap costs in the L_1 metric

We first handle the case in which the cost for the gap between two fragments is the distance between the end and start point of the two fragments in the L_1 metric. For two points $p, q \in \mathbb{R}^k$, this distance is defined by

$$d_1(p, q) = \sum_{i=1}^{k} |p.x_i - q.x_i|$$

```
ACCXXXX___AGG          ACCXXXXAGG
ACC____YYYAGG          ACCYYY_AGG
```

Fig. 4. Alignments based on the fragments ACC and AGG w.r.t. gap cost g_1 (left) and the sum-of-pairs gap cost with $\lambda > \frac{1}{2}\epsilon$ (right), where X and Y are anonymous characters.

and for two fragments $f \ll f'$ we define $g_1(f', f) = d_1(beg(f'),end(f))$. If an alignment of two sequences S_1 and S_2 shall be based on fragments and one uses this gap cost, then the characters between the two fragments are *deleted/inserted*; see left side of Fig. 4.

The problem with gap costs in our approach is that an RMQ does not take the cost $g(f', f)$ from recurrence (1) into account, and if we would explicitly compute $g(f', f)$ for every pair of fragments with $f \ll f'$, then this would yield a quadratic time algorithm. Thus, it is necessary to express the gap costs implicitly in terms of weight information attached to the points. We achieve this by using the *geometric cost* of a fragment f, which we define in terms of the terminus point $\mathtt{t}$ as $gc(f) = d_1(\mathtt{t}, end(f))$.

Lemma 4.1 *Let f, $\tilde{f}$, and f' be fragments such that $f \ll f'$ and $\tilde{f} \ll f'$. Then the inequality $\tilde{f}.score - g_1(f', \tilde{f}) > f.score - g_1(f', f)$ holds true if and only if the inequality $\tilde{f}.score - gc(\tilde{f}) > f.score - gc(f)$ holds.*

Proof

$$
\begin{aligned}
& \tilde{f}.score - g_1(f', \tilde{f}) > f.score - g_1(f', f) \\
\Leftrightarrow\ & \tilde{f}.score - \textstyle\sum_{i=1}^{k} (beg(f').x_i - end(\tilde{f}).x_i) > \\
& f.score - \textstyle\sum_{i=1}^{k} (beg(f').x_i - end(f).x_i) \\
\Leftrightarrow\ & \tilde{f}.score - \textstyle\sum_{i=1}^{k} (\mathtt{t}.x_i - end(\tilde{f}).x_i) > \\
& f.score - \textstyle\sum_{i=1}^{k} (\mathtt{t}.x_i - end(f).x_i) \\
\Leftrightarrow\ & \tilde{f}.score - gc(\tilde{f}) > f.score - gc(f)
\end{aligned}
$$

The second equivalence follows from adding $\sum_{i=1}^{k} beg(f').x_i$ to and subtracting $\sum_{i=1}^{k} \mathtt{t}.x_i$ from both sides of the inequality. Fig. 5 illustrates the lemma for $k = 2$. □

Because $\mathtt{t}$ is fixed, the value $gc(f)$ is known in advance for every fragment f. Therefore, Algorithm 1 needs only two slight modifications to take gap costs into account. First, we replace the statement $f'.score := f'.weight + f.score$

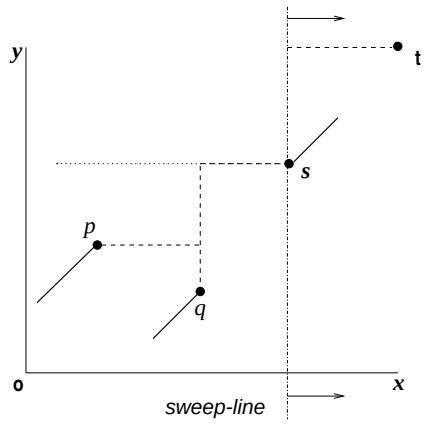

Fig. 5. Points p and q are active end points of the fragments f and $\tilde{f}$. The start point s of fragment f' is currently scanned by the sweeping line and $\mathtt{t}$ is the terminus point.

with

$$f'.score := f'.weight + f.score - g_1(f', f)$$

Second, if $\mathsf{points}[i]$ is the end point of f', then it will be activated with $f'.priority := f'.score - gc(f')$. Thus, an RMQ will return a point of maximum priority instead of a point of maximum score. The next lemma implies the correctness of the modified algorithm.

Lemma 4.2 *If the range maximum query* $\mathtt{RMQ}(\mathsf{0}, beg(f') - \vec{1})$ *returns the end point of fragment* $\tilde{f}$*, then we have* $\tilde{f}.score - g_1(f', \tilde{f}) = \max\{f.score - g_1(f', f) : f \ll f'\}$.

Proof $\mathtt{RMQ}(\mathsf{0}, beg(f') - \vec{1})$ returns the end point of fragment $\tilde{f}$ such that $\tilde{f}.priority = \max\{f.priority : f \ll f'\}$. Since $f.priority = f.score - gc(f)$ for every fragment f, it is an immediate consequence of Lemma 4.1 that $\tilde{f}.score - g_1(f', \tilde{f}) = \max\{f.score - g_1(f', f) : f \ll f'\}$. □

4.2 The sum-of-pairs gap cost

In this section we consider the gap cost associated with the "sum-of-pairs" model, which was introduced by Myers and Miller [23]. For clarity of presentation, we first treat the case $k = 2$ because the general case $k > 2$ is rather involved.

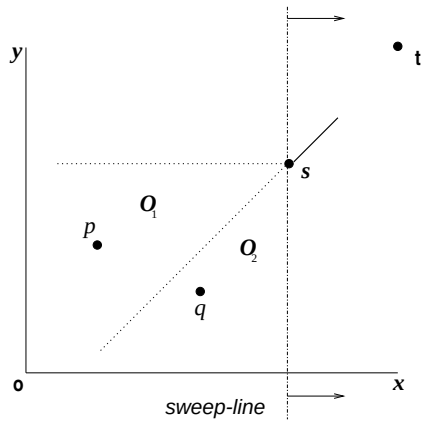

Fig. 6. The first quadrant of point s is divided into two octants.

4.2.1 The case $k = 2$:

For two points $p, q \in \mathbb{R}^2$, we write $\Delta_{x_i}(p, q) = |p.x_i - q.x_i|$, where $i \in \{1, 2\}$. We will sometimes simply write Δ_{x_1} and Δ_{x_2} if their arguments can be inferred from the context. The sum-of-pairs distance of two points $p, q \in \mathbb{R}^2$ depends on the parameters ϵ and λ and was defined in [23] as follows:

$$d(p,q) = \begin{cases} \epsilon\Delta_{x_2} + \lambda(\Delta_{x_1} - \Delta_{x_2}) \text{ if } \Delta_{x_1} \geq \Delta_{x_2} \\ \epsilon\Delta_{x_1} + \lambda(\Delta_{x_2} - \Delta_{x_1}) \text{ if } \Delta_{x_2} \geq \Delta_{x_1} \end{cases}$$

We rearrange these terms and derive the following equivalent definition:

$$d(p,q) = \begin{cases} \lambda\Delta_{x_1} + (\epsilon - \lambda)\Delta_{x_2} \text{ if } \Delta_{x_1} \geq \Delta_{x_2} \\ (\epsilon - \lambda)\Delta_{x_1} + \lambda\Delta_{x_2} \text{ if } \Delta_{x_2} \geq \Delta_{x_1} \end{cases}$$

For two fragments f and f' with $f \ll f'$, we define $g(f', f) = d(beg(f'), end(f))$. Intuitively, $\lambda > 0$ is the cost of aligning an anonymous character with a gap position in the other sequence, while $\epsilon > 0$ is the cost of aligning two anonymous characters. For $\lambda = 1$ and $\epsilon = 2$, this gap cost coincides with the g_1 gap cost, whereas for $\lambda = 1$ and $\epsilon = 1$, this gap cost corresponds to the L_∞ metric. (The gap cost of connecting two fragments $f \ll f'$ in the L_∞ metric is defined by $g_\infty(f', f) = d_\infty(beg(f'),end(f))$, where $d_\infty(p, q) = \max_{i \in [1..k]} |p.x_i - q.x_i|$ for $p, q \in \mathbb{R}^k$.) Following [23, 30], we demand that $\lambda > \frac{1}{2}\epsilon$ because otherwise it would always be best to connect fragments entirely by gaps as in the L_1 metric. So if an alignment of two sequences S_1 and S_2 shall be based on fragments and one uses the sum-of-pairs gap cost with $\lambda > \frac{1}{2}\epsilon$, then the characters between the two fragments are *replaced* as long as possible and the remaining characters are *deleted* or *inserted*; see right side of Fig. 4.

In order to compute the score of a fragment f' with $beg(f') = s$, the following definitions are useful. The *first quadrant* of a point $s \in \mathbb{R}^2$ consists of all points $p \in \mathbb{R}^2$ with $p.x_1 \leq s.x_1$ and $p.x_2 \leq s.x_2$. We divide the first quadrant of s into regions O_1 and O_2 by the straight line $x_2 = x_1 + (s.x_2 - s.x_1)$. O_1, called the *first octant* of s, consists of all points p in the first quadrant of s satisfying $\Delta_{x_1} \geq \Delta_{x_2}$ (i.e., $s.x_1 - p.x_1 \geq s.x_2 - p.x_2$), these are the points lying above or on the straight line $x_2 = x_1 + (s.x_2 - s.x_1)$; see Fig. 6. The *second octant* O_2 consists of all points q satisfying $\Delta_{x_2} \geq \Delta_{x_1}$ (i.e., $s.x_2 - q.x_2 \geq s.x_1 - q.x_1$), these are the points lying below or on the straight line $x_2 = x_1 + (s.x_2 - s.x_1)$. Then $f'.score = f'.weight + \max\{v_1, v_2\}$, where $v_i = \max\{f.score - g(f', f) : f \ll f'$ and $end(f)$ lies in octant $O_i\}$, for $i \in \{1, 2\}$.

However, our chaining algorithms rely on RMQs, and these work only for orthogonal regions, not for octants. For this reason, we will make use of the *octant-to-quadrant* transformations of Guibas and Stolfi [16]. The transformation $T_1 : (x_1, x_2) \mapsto (x_1 - x_2, x_2)$ maps the first octant to a quadrant. More precisely, point $T_1(p)$ is in the first quadrant of $T_1(s)$ if and only if p is in the first octant of point s.[2] Similarly, for the transformation $T_2 : (x_1, x_2) \mapsto (x_1, x_2 - x_1)$, point q is in the second octant of point s if and only if $T_2(q)$ is in the first quadrant of $T_2(s)$. By means of these transformations, we can apply the same techniques as in the previous sections. We just have to define the geometric cost properly. The following lemma shows how to choose the geometric cost gc_1 for points in the first octant O_1. An analogous lemma holds for points in the second octant.

Lemma 4.3 *Let f, $\tilde{f}$, and f' be fragments such that $f \ll f'$ and $\tilde{f} \ll f'$. If $end(f)$ and $end(\tilde{f})$ lie in the first octant of $beg(f')$, then $\tilde{f}.score - g(f', \tilde{f}) > f.score - g(f', f)$ if and only if $\tilde{f}.score - gc_1(\tilde{f}) > f.score - gc_1(f)$, where $gc_1(\hat{f}) = \lambda\Delta_{x_1}(\mathtt{t}, end(\hat{f})) + (\epsilon - \lambda)\Delta_{x_2}(\mathtt{t}, end(\hat{f}))$ for any fragment $\hat{f}$.*

Proof Similar to the proof of Lemma 4.1. □

In Section 4.1 there was only one geometric cost gc, but here we have to take two different geometric costs gc_1 and gc_2 into account. To cope with this problem, we need two data structures D_1 and D_2, where D_i stores the set of points

$$\{T_i(end(f)) \mid f \text{ is a fragment}\}$$

[2] Observe that the transformation may yield points with negative coordinates, but it is easy to overcome this obstacle by an additional transformation (a translation). Hence we will skip this minor problem.

If we encounter the end point of fragment f' in Algorithm 1, then we activate point $T_1(end(f'))$ in D_1 with priority $f'.score - gc_1(f')$ and point $T_2(end(f'))$ in D_2 with priority $f'.score - gc_2(f')$. If we encounter the start point of fragment f', then we launch two range maximum queries, namely $\texttt{RMQ}(\texttt{0}, T_1(beg(f') - \vec{1}))$ in D_1 and $\texttt{RMQ}(\texttt{0}, T_2(beg(f') - \vec{1}))$ in D_2. If the first RMQ returns $T_1(end(f_1))$ and the second returns $T_2(end(f_2))$, then f_i is a fragment of highest priority in D_i, $1 \leq i \leq 2$, such that $T_i(end(f_i)) \ll T_i(beg(f'))$. Because a point p is in the octant O_i of point $beg(f')$ if and only if $T_i(p)$ is in the first quadrant of $T_i(beg(f'))$, it follows that f_i is a fragment such that its priority $f_i.score - gc_i(f_i)$ is maximal in octant O_i. Therefore, according to Lemma 4.3, the value $v_i = f_i.score - g(f', f_i)$ is maximal in octant O_i. Hence, if $v_1 > v_2$, then we set $f'.prec = f_1$ and $f'.score := f'.weight + v_1$. Otherwise, we set $f'.prec = f_2$ and $f'.score := f'.weight + v_2$.

For the sum-of-pairs gap cost, the two-dimensional chaining algorithm runs in $O(n \log n \log \log n)$ time and $O(n \log n)$ space because of the two-dimensional RMQ*s* required for the transformed points. This is in sharp contrast to gap costs in the L_1-metric, where we merely need one-dimensional RMQ*s*.

4.2.2 The case $k > 2$:

In this case, the sum-of-pairs gap cost is defined for fragments $f \ll f'$ by

$$g_{sop}(f', f) = \sum_{0 \leq i < j \leq k} g(f'_{i,j}, f_{i,j})$$

where $f'_{i,j}$ and $f_{i,j}$ are the two-dimensional fragments consisting of the ith and jth component of f' and f, respectively. For example, in case of $k = 3$, let $s = beg(f')$ and $p = end(f)$ and assume that $\Delta_{x_1}(s,p) \geq \Delta_{x_2}(s,p) \geq \Delta_{x_3}(s,p)$. In this case, we have $g_{sop}(f', f) = 2\lambda\Delta_{x_1} + \epsilon\Delta_{x_2} + (\epsilon - \lambda)2\Delta_{x_3}$ because $g(f'_{1,2}, f_{1,2}) = \lambda\Delta_{x_1} + (\epsilon - \lambda)\Delta_{x_2}$, $g(f'_{1,3}, f_{1,3}) = \lambda\Delta_{x_1} + (\epsilon - \lambda)\Delta_{x_3}$, and $g(f'_{2,3}, f_{2,3}) = \lambda\Delta_{x_2} + (\epsilon - \lambda)\Delta_{x_3}$. By contrast, if $\Delta_{x_1} \geq \Delta_{x_3} \geq \Delta_{x_2}$, then the equality $g_{sop}(f', f) = 2\lambda\Delta_{x_1} + (\epsilon - \lambda)2\Delta_{x_2} + \epsilon\Delta_{x_3}$ holds.

In general, each of the $k!$ permutations π of $1, \ldots, k$ yields a hyper-region R_π defined by $\Delta_{x_{\pi(1)}} \geq \Delta_{x_{\pi(2)}} \geq \ldots \geq \Delta_{x_{\pi(k)}}$ in which a specific formula for $g_{sop}(f', f)$ holds. That is, in order to obtain the score of a fragment f', we must compute $f'.score = f'.weight + \max\{v_\pi \mid \pi \text{ is a permutation of } 1, \ldots, k\}$, where

$$v_\pi = \max\{f.score - g_{sop}(f', f) : f \ll f' \text{ and } end(f) \text{ lies in } R_\pi\}$$

Because our RMQ-based approach requires orthogonal regions, each of these hyper-regions R_π of s must be transformed into the *first hyper-corner* of some

point $\tilde{s}$. The first hyper-corner of a point $\tilde{s} \in \mathbb{R}^k$ is the k-dimensional analogue to the first quadrant of a point in $\mathbb{R}^2$. It consists of all points $p \in \mathbb{R}^k$ with $p.x_i \leq \tilde{s}.x_i$ for all $1 \leq i \leq k$ (note that there are 2^k hyper-corners). We describe the generalization of the *octant-to-quadrant* transformations for the case $k = 3$. The extension to the case $k > 3$ is obvious. There are 3! hyper-regions, hence 6 transformations:

$$\begin{aligned}
&\Delta_{x_1} \geq \Delta_{x_2} \geq \Delta_{x_3} : T_1(x_1, x_2, x_3) = (x_1 - x_2, x_2 - x_3, x_3) \\
&\Delta_{x_1} \geq \Delta_{x_3} \geq \Delta_{x_2} : T_2(x_1, x_2, x_3) = (x_1 - x_3, x_2, x_3 - x_2) \\
&\Delta_{x_2} \geq \Delta_{x_1} \geq \Delta_{x_3} : T_3(x_1, x_2, x_3) = (x_1 - x_3, x_2 - x_1, x_3) \\
&\Delta_{x_2} \geq \Delta_{x_3} \geq \Delta_{x_1} : T_4(x_1, x_2, x_3) = (x_1, x_2 - x_3, x_3 - x_1) \\
&\Delta_{x_3} \geq \Delta_{x_1} \geq \Delta_{x_2} : T_5(x_1, x_2, x_3) = (x_1 - x_2, x_2, x_3 - x_1) \\
&\Delta_{x_3} \geq \Delta_{x_2} \geq \Delta_{x_1} : T_6(x_1, x_2, x_3) = (x_1, x_2 - x_1, x_3 - x_2)
\end{aligned}$$

In what follows, we will focus on the particular case where π is the identity permutation. The hyper-region corresponding to the identity permutation will be denoted by R_1 and its transformation by T_1. The other permutations are numbered in an arbitrary order and are handled similarly.

Lemma 4.4 *Point $p \in \mathbb{R}^k$ is in hyper-region R_1 of point s if and only if $T_1(p)$ is in the first hyper-corner of $T_1(s)$, where $T_1(x_1, x_2, \ldots, x_k) = (x_1 - x_2, x_2 - x_3, \ldots, x_{k-1} - x_k, x_k)$.*

Proof $T_1(p)$ is in the first hyper-corner of $T_1(s)$

$$\begin{aligned}
&\Leftrightarrow && T_1(s).x_i \geq T_1(p).x_i && \text{for all } 1 \leq i \leq k \\
&\Leftrightarrow && s.x_i - s.x_{i+1} \geq p.x_i - p.x_{i+1} \text{ and } s.x_k \geq p.x_k && \text{for all } 1 \leq i < k \\
&\Leftrightarrow && (s.x_1 - p.x_1) \geq (s.x_2 - p.x_2) \geq \ldots \geq (s.x_k - p.x_k) \\
&\Leftrightarrow && \Delta_{x_1}(s, p) \geq \Delta_{x_2}(s, p) \geq \ldots \geq \Delta_{x_k}(s, p)
\end{aligned}$$

The last statement holds if and only if p is in hyper-region R_1 of s. □

For each hyper-region R_j, we compute the corresponding geometric cost $gc_j(f)$ of every fragment f. Note that for every index j a k-dimensional analogue to Lemma 4.3 holds. Furthermore, for each transformation T_j, we keep a data structure D_j that stores the transformed end points $T_j(end(f))$ of all fragments f. Algorithm 4 generalizes the 2-dimensional chaining algorithm described above to k dimensions. For every start point $beg(f')$ of a fragment f', Algorithm 4 searches for a fragment f in the first hyper-corner of $beg(f')$ such that $f.score - g_{sop}(f', f)$ is maximal. This entails $k!$ `RMQ`s because the

Algorithm 4. (k-dim. chaining of n fragments w.r.t. the sum-of-pairs gap cost)
Sort all start and end points of the n fragments in ascending order w.r.t. their x_1 coordinate and store them in the array points; because we include the end point of the origin and the start point of the terminus, there are $2n+2$ points.
for $j := 1$ **to** $k!$
  apply transformation T_j to the end points of the fragments and store the resulting points as inactive in the k-dimensional data structure D_j
for $i := 1$ **to** $2n+2$
  if points$[i]$ is the start point of fragment f' **then**
    $maxRMQ := -\infty$
    for $j := 1$ **to** $k!$
      $q :=$ RMQ$(0, T_j(\text{points}[i] - \vec{1}))$ in D_j
      determine the fragment f_q with $T_j(end(f_q)) = q$
      $maxRMQ := \max\{maxRMQ, f_q.score - g_{sop}(f', f_q)\}$
      if $f_q.score - g_{sop}(f', f_q) = maxRMQ$ **then** $f := f_q$
    $f'.prec := f$
    $f'.score := f'.weight + maxRMQ$
  else /⋆ points$[i]$ is end point of a fragment f' ⋆/
    for $j := 1$ **to** $k!$
      activate $T_j(\text{points}[i])$ in D_j with priority $f'.score - gc_j(f')$

first hyper-corner is divided into $k!$ hyper-regions. Analogously, for every end point $end(f')$ of a fragment f', Algorithm 4 performs $k!$ activation operations. Therefore, the total time complexity of Algorithm 4 is $O(k!\, n \log^{k-1} n \log\log n)$ and its space requirement is $O(k!\, n \log^{k-1} n)$. This result improves the running time of Myers and Miller's algorithm [23] by a factor $\frac{\log n}{\log\log n}$.

5 The local chaining algorithm

In the previous sections, we have tackled the global chaining problem, which asks for an optimal chain starting at the origin 0 and ending at terminus t. However, in many applications (such as searching for local similarities in genomic sequences) one is interested in chains that can start and end with arbitrary fragments. If we remove the restriction that a chain must start at the origin and end at the terminus, we get the local chaining problem; see Fig. 7.

Definition 5.1 Given n weighted fragments and a gap cost function g, the *local fragment-chaining problem* is to determine a chain of maximum score ≥ 0. Such a chain will be called *optimal local chain*.

Note that if g is the constant function 0, then an optimal local chain must also be an optimal global chain, and vice versa. Our solution to the local chaining

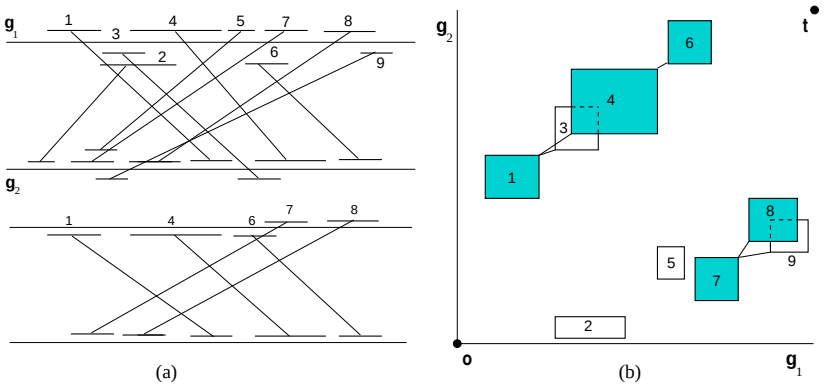

Fig. 7. Computation of local chains of colinear non-overlapping fragments. The optimal local chain is composed of the fragments 1, 4, and 6. Another significant local chain consists of the fragments 7 and 8.

problem is a variant of the global chaining algorithm. For ease of presentation, we will use gap costs corresponding to the L_1 metric (see Section 4.1), but the approach also works with the sum-of-pairs gap cost (see Section 4.2).

Definition 5.2 Let

$$f'.score = \max\{score(C) : C \text{ is a chain ending with } f'\}$$

A chain C ending with f' and satisfying $f'.score = score(C)$ will be called *optimal chain ending with* f'.

Lemma 5.3 *The following equality holds:*

$$f'.score = f'.weight + \max\{0, f.score - g_1(f', f) : f \ll f'\} \tag{2}$$

Proof Let $C' = f_1, f_2, \ldots, f_\ell, f'$ be an optimal chain ending with f', that is, $score(C') = f'.score$. Because the chain that solely consists of fragment f' has score $f'.weight \geq 0$, we must have $score(C') \geq f'.weight$. If $score(C') = f'.weight$, then $f.score - g_1(f', f) \leq 0$ for every fragment f that precedes f', because otherwise it would follow $score(C') > f'.weight$. Hence equality (2) holds in this case. So suppose $score(C') > f'.weight$. Clearly, $score(C') = f'.weight + score(C) - g_1(f', f_\ell)$, where $C = f_1, f_2, \ldots, f_\ell$. It is not difficult to see that C must be an optimal chain that is ending with f_ℓ because otherwise C' would not be optimal. Therefore, $score(C') = f'.weight + f_\ell.score - g_1(f', f_\ell)$. If there were a fragment f that precedes f' such that $f.score - g_1(f', f) > f_\ell.score - g_1(f', f_\ell)$, then it would follow that C' is not optimal. We conclude that equality (2) holds. □

With the help of Lemma 5.3, we obtain Algorithm 5. It is not difficult to

Algorithm 5. (Finding an optimal local chain)
for every fragment f' **do begin**
determine $\hat{f}$ with $\hat{f}.score - g_1(f', \hat{f}) = \max\{f.score - g_1(f', f) \mid f \ll f'\}$
$max := \max\{0, \hat{f}.score - g_1(f', \hat{f})\}$
if $max > 0$ **then** $f'.prec := \hat{f}$ **else** $f'.prec := NULL$
$f'.score := f'.weight + max$
end
determine a fragment $\tilde{f}$ such that $\tilde{f}.score = \max\{f.score \mid f \text{ is a fragment}\}$
report an optimal local chain by following the pointers $\tilde{f}.prec$ until a fragment f with $f.prec = NULL$ is reached

verify that we can implement this algorithm by means of the techniques of the previous sections such that it solves the local fragment-chaining problem in the same time and space complexities as our solution to the global fragment-chaining problem.

We stress that Algorithm 5 can easily be modified, so that it can report all chains whose score exceeds some threshold T (in Algorithm 5, instead of determining a fragment $\tilde{f}$ of maximum score, one determines all fragments whose score exceeds T). Such chains will be called *significant local chains*; see Fig. 7. As outlined in Section 1, under stringent thresholds, significant local chains of colinear non-overlapping fragments represent candidate regions of conserved synteny. Furthermore, the automatic identification of regions of similarity is a first step toward an automatic detection of genome rearrangements. The interested reader is referred to [2], where we have provided evidence that our local chaining method can be used to detect genome rearrangements such as transpositions and inversions.

6 Conclusions

In this article, we have presented line-sweep algorithms that solve both the global and the local fragment-chaining problem of multiple genomes. For $k > 2$ genomes, our algorithms take

- $O(n \log^{k-2} n \log\log n)$ time and $O(n \log^{k-2} n)$ space without gap costs,
- $O(n \log^{k-2} n \log\log n)$ time and $O(n \log^{k-2} n)$ space for gap costs in the L_1 metric,
- $O(k!\ n \log^{k-1} n \log\log n)$ time and $O(k!\ n \log^{k-1} n)$ space for the sum-of-pairs gap cost and for gap costs in the L_∞ metric.

For $k = 2$, they take $O(n \log n)$ time and $O(n)$ space for gap costs in the L_1 metric. If the fragments are already sorted, then the time complexity reduces

to $O(n \log \log n)$. For the sum-of-pairs gap cost and for gap costs in the L_∞ metric, our algorithms take $O(n \log n \log \log n)$ time and $O(n \log n)$ space.

We stress that our algorithm can employ any other data structure that supports orthogonal range-searching. For example, if the *kd*-tree is used instead of the range tree, the algorithms take $O((k-1)n^{2-\frac{1}{k-1}})$ time and $O(n)$ space for gap costs in the L_1 metric. (It has been shown in [20] that answering *one* d-dimensional range query with the *kd*-tree takes $O(dn^{1-\frac{1}{d}})$ time in the worst case.) Moreover, for small k, a collection of programming tricks can speed up the running time in practice; for more details we refer the reader to [6].

Among other things, the software tool CHAINER [3] contains an implementation of the chaining algorithms presented here. Currently, this implementation uses gap costs in the L_1 metric and is based on *kd*-trees. It remains future work to experimentally compare this *kd*-tree version with an implementation based on range trees.

Acknowledgment

The authors were supported by DFG-grant Oh 54/4-1. Thanks go to an anonymous reviewer whose suggestions helped to improve the presentation of this article.

References

[1] M.I. Abouelhoda and E. Ohlebusch. Multiple genome alignment: Chaining algorithms revisited. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, LNCS 2676, pages 1–16. Springer-Verlag, 2003.

[2] M.I. Abouelhoda and E. Ohlebusch. A local chaining algorithm and its applications in comparative genomics. In *Proc. 3rd Workshop on Algorithms in Bioinformatics*, LNBI 2812, pages 1–16. Springer-Verlag, 2003.

[3] M.I. Abouelhoda and E. Ohlebusch. CHAINER: Software for comparing genomes. In *12th International Conference on Intelligent Systems for Molecular Biology/3rd European Conference on Computational Biology*, to appear.

[4] P. Agarwal. Range searching. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 31, pages 575–603. CRC Press, 1997.

[5] S. Batzoglou, L. Pachter, J.P. Mesirov, B. Berger, and E.S. Lander. Human and mouse gene structure: Comparative analysis and application to exon prediction. *Genome Research*, 10:950–958, 2001.

[6] J.L. Bently. K-d trees for semidynamic point sets. In *6th Annual ACM Symposium on Computational Geometry*, pages 187–197. ACM, 1990.

[7] N. Bray and L. Pachter. MAVID multiple alignment server. *Nucleic Acids Research*, 31:3525–3526, 2003.

[8] M. Brudno, C.B. Do, G.M. Cooper, M.F. Kim, E. Davydov, E.D. Green, A. Sidow, and S. Batzoglou. LAGAN and Multi-LAGAN: Efficient tools for large-scale multiple alignment of genomic DNA. *Genome Research*, 13(4):721–731, 2003.

[9] P. Chain, S. Kurtz, E. Ohlebusch, and T. Slezak. An applications-focused review of comparative genomics tools: Capabilities, limitations and future challenges. *Briefings in Bioinformatics*, 4(2):105-123, 2003.

[10] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[11] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of whole genomes. *Nucleic Acids Res.*, 27:2369–2376, 1999.

[12] A.L. Delcher, A. Phillippy, J. Carlton, and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res.*, 30(11):2478-2483, 2002.

[13] D. Eppstein, Z. Galil, R. Giancarlo, and G.F. Italiano. Sparse dynamic programming. I:Linear cost functions; II:Convex and concave cost functions. *Journal of the ACM*, 39:519–567, 1992.

[14] H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th Annual ACM Symposium on Theory of Computing*, pages 135–143. ACM, 1984.

[15] M.S. Gelfand, A.A. Mironov, and P.A. Pevzner. Gene recognition via spliced sequence alignment. *Proc. Nat. Acad. Sci.*, 93:9061–9066, 1996.

[16] L.J. Guibas and J. Stolfi. On computing all north-east nearest neighbors in the L_1 metric. *Information Processing Letters*, 17(4):219–223, 1983.

[17] D.B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Mathematical Systems Theory*, 15:295–309, 1982.

[18] M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18:S312–S320, 2002.

[19] W.J. Kent and A.M. Zahler. Conservation, regulation, synteny, and introns in a large-scale C.briggsae-C.elegans genomic alignment. *Genome Research*, 10:1115–1125, 2000.

[20] D.T. Lee and C.K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9:23–29, 1977.

[21] B. Morgenstern. A space-efficient algorithm for aligning large genomic sequences. *Bioinformatics* 16:948-949, 2000.

[22] E.W. Myers and X. Huang. An $O(n^2 \log n)$ restriction map comparison and search algorithm. *Bulletin of Mathematical Biology*, 54(4):599–618, 1992.

[23] E.W. Myers and W. Miller. Chaining multiple-alignment fragments in sub-quadratic time. *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 38–47. ACM, 1995.

[24] E. Ohlebusch and M.I. Abouelhoda. Chaining algorithms and applications in comparative genomics. In S. Aluru, editor, *Handbook of Computational Molecular Biology*, chapter 20. CRC Press, to appear.

[25] F.P. Preparata and M.I. Shamos. *Computational geometry: An introduction*. Springer-Verlag, New York, 1985.

[26] S. Schwartz, J.K. Kent, A. Smit, Z. Zhang, R. Baertsch, R. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with BLASTZ. *Genome Research*, 13:103–107, 2003.

[27] S. Schwartz, L. Elnitski, M. Li, M. Weirauch, C. Riemer, A. Smit, NISC Comparative Sequencing Program, E. D. Green, R. C. Hardison, and W. Miller. MultiPipMaker and supporting tools: Alignments and analysis of multiple genomic DNA sequences. *Nucleic Acids Research*, 31(13):3518–3524, 2003.

[28] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[29] D.E. Willard. New data structures for orthogonal range queries. *SIAM Journal of Computing*, 14:232–253, 1985.

[30] Z. Zhang, B. Raghavachari, R. Hardison, and W. Miller. Chaining multiple-alignment blocks. *Journal of Computational Biology*, 1:217–226, 1994.