Multiple Genome Alignment: Chaining Algorithms Revisited

Mohamed Ibrahim Abouelhoda[1] and Enno Ohlebusch[2]

[1] Faculty of Technology, University of Bielefeld, P.O. Box 10 01 31, 33501 Bielefeld, Germany. mibrahim@techfak.uni-bielefeld.de
[2] Faculty of Computer Science, University of Ulm, Albert-Einstein-Allee, 89069 Ulm, Germany. eo@informatik.uni-ulm.de

Abstract. Given n fragments from $k > 2$ genomes, we will show how to find an optimal chain of colinear non-overlapping fragments in time $O(n \log^{k-2} n \log\log n)$ and space $O(n \log^{k-2} n)$. Our result solves an open problem posed by Myers and Miller because it reduces 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 $k = 2$ genomes, our algorithm takes $O(n \log n)$ time and $O(n)$ space.

1 Introduction

Given the continuing improvements in high-throughput genomic sequencing and the ever-expanding 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 analyses can be addressed, methods are required for comparing such large sequences. If the organisms under consideration are closely related, then global alignments are the strategy of choice. Although there is an immediate need for "reliable and automatic software for aligning three or more genomic sequences" [12], currently only the software tool *MGA* [10] solves the problem of aligning multiple complete genomes. This is because all previous multiple alignment algorithms were designed for comparing single protein sequences or DNA sequences containing a single gene, and are incapable of producing long alignments. In order to cope with the shear volume of data, *MGA* uses an anchor-based method that is divided into three phases: (1) computation of fragments (regions in the genomes that are similar—in *MGA* these are multiple maximal exact matches), (2) computation of an optimal 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.

This paper is concerned with algorithms for solving the combinatorial chaining problem of the second phase; see Fig. 1. Note that every genome alignment tool has to solve the chaining problem somehow, but the algorithms differ from

R. Baeza-Yates et al. (Eds.): CPM 2003, LNCS 2676, pp. 1–16, 2003.

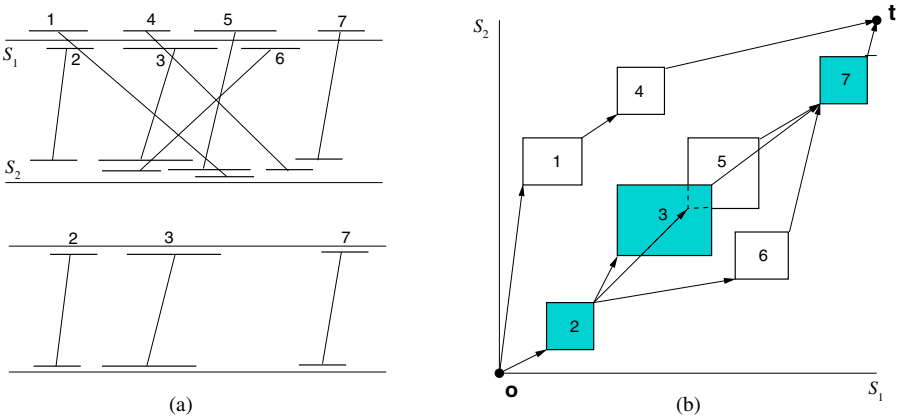

Fig. 1. Given a set of fragments (upper left figure), an optimal chain of colinear non-overlapping fragments (lower left figure) can be computed, e.g., by computing an optimal path in the graph in (b) (in which not all edges are shown).

tool to tool; see, e.g., [3,13,17]. Chaining algorithms are also useful in other bioinformatics applications such as comparing restriction maps [9] or solving the exon assembly problem which is part of eucaryotic gene prediction [6].

A well-known solution to the chaining problem consists of finding a maximum weight path in a weighted directed acyclic graph; see, e.g., [10]. 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, *MGA* currently uses a variant of an algorithm devised by Zhang et al. [20], but without taking gap costs into account. This algorithm takes advantage of the geometric nature of the chaining problem. It constructs an optimal chain using orthogonal range search based on *kd*-trees, a data structure known from computational geometry. As is typical with *kd*-tree methods, however, no rigorous analysis of the running time of the algorithm is known; cf. [15].

Another chaining algorithm, devised by Myers and Miller [15], falls into the category of *sparse dynamic programming* [5]. Their algorithm is based on the line-sweep paradigm, and uses orthogonal range search 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" [4]. In particular, for $k = 2$ sequences it is one log-factor slower than previous chaining algorithms [5,14], which require only $O(n \log n)$ time. In the epilog of their paper [15], 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 paper, we solve this problem. Surprisingly, we can not only reduce the time and space complexities by a log-factor but actually improve the time complexity by a factor $\frac{\log^2 n}{\log \log n}$. In essence, this improvement is achieved by (1) a combination of

fractional cascading [19] with the efficient priority queues of [18,8], which yields a more efficient search than on ordinary range trees, and (2) by incorporating gap costs into the weight of fragments, so that it is enough to determine a maximum function value over a semi-dynamic set (instead of a dynamic set). In related work, Baker and Giancarlo [1] have shown how to efficiently compute a longest common subsequence from fragments, which is a variant of our problem, but their algorithm is restricted to two sequences.

2 Basic Concepts and Definitions

For any point $p \in \mathbb{R}^k$, let $p.x_1, p.x_2, \ldots, p.x_k$ denote its coordinates. If $k = 2$, the coordinates of p will also be written as $p.x$ and $p.y$. A hyper-rectangle (called hyperrectangular domain in [16]) 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 also be denoted by $R(p, q)$, where $p = (l_1, \ldots, l_k)$ and $q = (h_1, \ldots, h_k)$ are its two extreme corner points. In the problem we consider, all points are given in advance (off-line). Therefore, it is possible to map the points into $\mathbb{N}^k$, called the *rank space*; see, e.g., [2]. Every point $(x_1, x_2, \ldots, x_k)$ is mapped to point $(r_1, r_2, \ldots, r_k)$, where r_i, $1 \leq i \leq k$, is the index (or rank) of point p in a list which is sorted in ascending order w.r.t. dimension x_i. This transformation takes $O(kn \log n)$ time and $O(n)$ space because one has to sort the points k times. Thus, we can assume that the points are already transformed to the rank space.

For $1 \leq i \leq k$, S_i denotes a string of length $|S_i|$. In our application, S_i is the DNA sequence of a genome. $S_i[l_i \ldots h_i]$ is the substring of S_i starting at position l_i and ending at position h_i. An *exact fragment* (or *multiple exact match*) 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 $S_1[l_1 \ldots h_1] = S_2[l_2 \ldots h_2] = \ldots = S_k[l_k \ldots h_k]$, i.e., the substrings are identical. It is maximal, if the substrings cannot be simultaneously extended to the left and to the right in every S_i. If mismatches are allowed in the substrings, then we speak of a *gap-free fragment*. If one further allows insertions and deletions (so that the substrings may be of unequal length), we will use the general term *fragment*. Many algorithms have been developed to efficiently compute all kinds of fragments (e.g., [10,11]), and the algorithms presented here work for arbitrary fragments.

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 non-negative. In the following, the words *number of genomes* and *dimension* will thus be used synonymously. With every fragment f, we associate a weight $f.weight \in \mathbb{R}$. This weight can, for example, be the length of the fragment (in case of gap-free fragments) or its statistical significance.

In what follows, we will often identify the point $beg(f)$ or $end(f)$ with the fragment f. 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 $\mathtt{0} = (0, \ldots, 0)$ (the origin) and $\mathtt{t} = (|S_1|, \ldots, |S_k|)$ (the terminus) as

fragments with weight 0. For these fragments, we define $beg(\mathsf{0}) = \perp$, $end(\mathsf{0}) = \mathsf{0}$, $\mathsf{0}.score = 0$, $beg(\mathsf{t}) = \mathsf{t}$, and $end(\mathsf{t}) = \perp$.

Definition 1. *We define the 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'. We further define $\mathsf{0} \ll f \ll \mathsf{t}$ for every fragment f with $f \neq \mathsf{0}$ and $f \neq \mathsf{t}$.*

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

Given a set of n fragments and a *gap cost* function g, the *fragment-chaining problem* is to determine a chain of maximum score (called optimal 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). An optimal chain of fragments, starting at the origin $\mathsf{0}$ and ending at terminus t, 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 that start at $\mathsf{0}$ and end 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 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 Chaining Algorithm without Gap Cost

3.1 The Chaining Algorithm

Because our algorithm is based on orthogonal range search for maximum, we have to recall two notions. Given a set S of points in $\mathbb{R}^k$ with associated score, a *range query* (`RQ`) asks for all the points of S that lie in a hyper-rectangle $R(p, q)$, while a *range maximum query* (`RMQ`) asks for a point of maximum score in $R(p, q)$. In the following, `RMQ` will also denote a procedure that takes two points p and q as input and returns a point of maximum score in the hyper-rectangle $R(p, q)$.

Lemma 3. *Suppose that the gap cost function g is the constant function 0. If* RMQ$(0, beg(f'))$ *returns the end point of fragment f, then $f'.score = f'.weight + f.score$.*

Proof. This follows immediately from recurrence (1).

We will further use the line-sweep paradigm to construct an optimal chain. 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. 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, if s is the start point of fragment f', then an optimal chain ending at f' can be found by a 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 $R(0, (s.x_2, \ldots, s.x_k))$, 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. The following algorithm is based on such a data structure D, which will be defined later.

Algorithm 4 *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$(0, (\text{points}[i].x_2, \ldots, \text{points}[i].x_k))$
    determine the fragment f with $end(f) = q$
    $f'.prec := f$
    $f'.score := f'.weight + f.score$
  else $\backslash\star$ points$[i]$ *is end point of a fragment* $\star\backslash$
    activate $(\text{points}[i].x_2, \ldots, \text{points}[i].x_k)$ *in D*

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 that Algorithm 4 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 RMQ with activation in time $O(n \log^{d-1} n \log\log n)$ and space $O(n \log^{d-1} n)$, where d is the dimension and n is the number of points. Because in our chaining problem $d = k-1$, finding an optimal chain by Algorithm 4 takes $O(n \log^{k-2} n \log\log n)$ time and $O(n \log^{k-2} n)$ space.

3.2 Answering RMQ with Activation Efficiently

In the following, we assume the reader to be familiar with *range trees*. An introduction to this well-known data structure can, for example, be found in [16, pp. 83-88]. Given a set S of n d-dimensional points, its range tree can be built in $O(n \log^{d-1} n)$ time and space and it supports range queries $\texttt{RQ}(p, q)$ in $O(\log^d n + z)$ time, where z is the number of points in the hyper-rectangle $R(p, q)$. The technique of *fractional cascading* [19] saves one log-factor in answering range queries.[1] We briefly describe this technique because we want to modify it to answer $\texttt{RMQ}(0, q)$ with activation 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 binary search trees (called *y-trees*). In fractional cascading, the *y-trees* are replaced with arrays (called *y-arrays*) as follows. Let $v.L$ and $v.R$ be the left and right child nodes of a node $v \in$ *x-tree* and let A_v denote the *y-array* of v. That is, A_v contains all the points in the leaf list of 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 an element q_1 of $A_{v.L}$, where q_1 is either p itself or the rightmost element in A_v that precedes p and also occurs in $A_{v.L}$. In an implementation, *Lptr* is the index with $A_{v.L}[Lptr] = q_1$. Analogously, the right pointer *Rptr* points to an element q_2 of $A_{v.R}$, where q_2 is either p itself or the rightmost element in A_v that precedes p and also occurs in $A_{v.R}$. Fig. 2 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 $p_{h_2}.y \in [0 \ldots h_2]$. Then, in the second stage, the *x-tree* is traversed (while keeping track of the downstream pointers) to locate the rightmost leave p_{h_1} such that $p_{h_1}.x \in [0 \ldots h_1]$. During the traversal of the *x-tree*, we identify a set of nodes which we call *maximum splitting nodes*. A maximum splitting node is either a node on the path from the root to p_{h_1} such that the points in its leaf list are within $[0 \ldots h_1]$ or it is a left child of a node on the path satisfying the same condition. The set of maximum splitting nodes is the smallest set of nodes $v_1, \ldots, v_\ell \in$ *x-tree* such that $\uplus_{j=1}^{\ell} A_{v_j} = \texttt{RQ}(0, (h_1, \infty))$.[2] In other words, $P := \uplus_{j=1}^{\ell} A_{v_j}$ contains every point $p \in S$ such that $p.x \in [0 \ldots h_1]$. However, not every point $p \in P$ satisfies $p.y \in [0 \ldots h_2]$. 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 maximum splitting 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 $R(0, (h_1, h_2))$, whereas all other elements of A_{v_j} lie in $R(0, (h_1, h_2))$. For this reason, we will call the element e_j the *splitting* element. It is easy to see that the number of maximum splitting nodes is $O(\log n)$. Moreover, we can find all of them and the splitting elements of their *y-arrays* in $O(\log n)$ time; cf. [19].

[1] In the same construction time and using the same space as the original range tree.
[2] $\uplus$ denotes disjoint union.

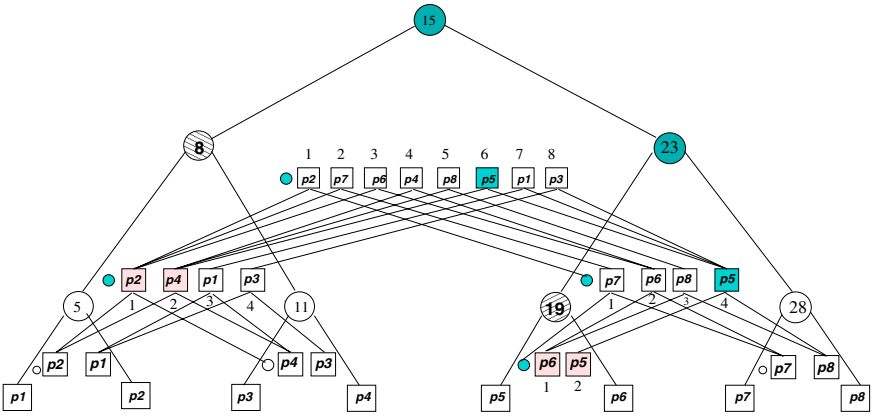

Fig. 2. Fractional cascading: colored nodes are the visited ones. Hatched nodes are the maximum splitting nodes. The small circles refer to NULL pointers. In this example, $p_{h_1} = p_6$ and $p_{h_2} = p_5$. The colored elements of the *y-arrays* of the maximum splitting nodes are the points in the query rectangle, which is shown in Fig. 3. The numerical value in every internal node is the x coordinate that separates the points in its left subtree from those occurring in its right subtree.

Therefore, the range tree with fractional cascading supports 2-dimensional range queries in $O(\log n + z)$ time. For dimension $d > 2$, it takes time $O(\log^{d-1} n + z)$.

In order to answer $\texttt{RMQ}(0, q)$ with activation efficiently, we will further enhance every *y-array* that occurs in the fractional cascading data structure with a priority queue as described in [18,8]. Each of these queues is (implicitly) constructed over the rank space of the points in the *y-array* (note that the *y-arrays* are sorted w.r.t. the y dimension). The rank space of the points in the *y-array* consists of points in the range $[0 \ldots m]$, where m is the size of the *y-array*. The priority queue supports the operations $insert(r)$, $delete(r)$, $predecessor(r)$

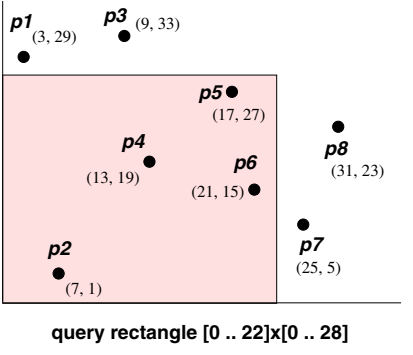

Fig. 3. Query rectangle for the example of Fig. 2.

(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 $[0 \ldots m]$. Algorithm 5 shows how to activate a point in the range tree and Algorithm 6 answers a `RMQ`.

Algorithm 5 *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 $(v \neq \bot)$
  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**
    $r := A_v[r].Rptr$
    $v := v.L$
  else
    $r := A_v[r].Rptr$
    $v := v.R$

Note that in the outer while-loop of Algorithm 5, 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 $A_v[i_1].score < A_v[i_2].score < \ldots < A_v[i_\ell].score$.

Algorithm 6 *`RMQ`$(0, q)$ in the data structure D*

$v :=$ *root node of the x-tree*
$max\_score := -\infty$
$max\_point := \bot$
find the rank (index) r of the rightmost point p with $p.y \in [0 \ldots q.y]$ in A_v
while $(v \neq \bot)$
  if $(v.xmax \leq q.x)$ **then** $\backslash\star$ *v is a maximum splitting node* $\star\backslash$
    $tmp :=$ *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 $(v.xkey \leq q.x)$ **then** $\backslash\star$ *$v.L$ is a maximum splitting node* $\star\backslash$
    $tmp :=$ *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
    $r := A_v[r].Lptr$
    $v := v.L$

In Algorithm 6, we assume that every node v has a field $v.xmax$ such that $v.xmax = \max\{p.x \mid p \in A_v\}$. Furthermore, $v.xkey$ is an x-coordinate (computed during the construction of D) that separates the points occurring in $A_{v.L}$

(or equivalently, in the leaf list of $v.L$) from those occurring in $A_{v.R}$ (or equivalently, in the leaf list of $v.R$). Algorithm 6 gives pseudocode for answering $\texttt{RMQ}(0, q)$, but we would also like to describe the algorithm on a higher level. In essence, Algorithm 6 locates all maximal splitting nodes $v_1, \ldots, v_\ell$ in D for the hyper-rectangle $R(0, q)$. For any v_j, $1 \leq j \leq \ell$, let the r_jth element be the splitting 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 splitting element of A_{v_j}, then all points $A_{v_j}[i]$ with $i \leq r_j$ are in $R(0, q)$, whereas all other elements $A_{v_j}[i]$ with $i > r_j$ are not in $R(0, q)$. Since Algorithm 5 maintains the above-mentioned invariant, the element with highest score in the priority queue of A_{v_j} that lies in $R(0, q)$ is $q_j = \mathit{predecessor}(r_j)$ (if r_j is in the priority queue of A_{v_j}, then $q_j = r_j$ because $\mathit{predecessor}(r_j)$ gives the largest element $\leq r_j$). Algorithm 6 then computes $\mathit{max\_score} := \max\{A_{v_j}[q_j].\mathit{score} \mid 1 \leq j \leq \ell\}$ and returns $\mathit{max\_point} = A_{v_i}[q_i]$, where $A_{v_i}[q_i].\mathit{score} = \mathit{max\_score}$.

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

Theorem 7. *Given $k > 2$ genomes and n fragments, an optimal 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 4, 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 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. But the transformation to the rank space and the sorting procedure in Algorithm 4 require $O(n \log n)$ time, and thus dominate the overall time complexity of Algorithm 4. To sum up, Algorithm 4 takes $O(n \log n)$ time and $O(n)$ space for $k = 2$.

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

```
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 g_∞ (right), where X and Y are anonymous characters.

metric. For two points $p, q \in \mathbb{R}^k$, this distance is defined by

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

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 a 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 8. *Let f, $\tilde{f}$, and f' be fragments such that $f \ll f'$ and $\tilde{f} \ll f'$. Then we have $\tilde{f}.score - g_1(f', \tilde{f}) > f.score - g_1(f', f)$ if and only if the inequality $\tilde{f}.score - gc(\tilde{f}) > f.score - gc(f)$ holds.*

Proof.

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

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 4 needs only two slight modifications to take gap costs into account. In order to apply Lemma 8, we set $\mathtt{0}$.score $= -g_1(\mathtt{t}, \mathtt{0})$. Moreover, in Algorithm 4 we replace the statement $f'.score := f'.weight + f.score$ with

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

We subtract $gc(f')$ in view of Lemma 8. Furthermore, we have to add $gc(f)$ to compensate for the subtraction of this value when the score of fragment f was

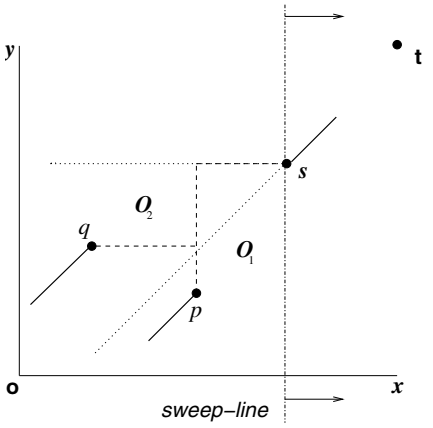

Fig. 5. Points p and q are active end points of the fragments f and $\tilde{f}$. The start point s of fragment f' is currently scanned by the sweeping line and t is the terminus point.

computed. This modified algorithm maintains the following invariant: If the end point of a fragment f is active, then $f.score$ stores the maximum score of all chains (with g_1 gap costs taken into account) that start at 0 and end at f minus the geometric cost of f.

4.2 The Sum-of-Pair Gap Cost

For clarity of presentation, we first treat the case $k = 2$ because the general case $k > 2$ is rather involved.

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-pair distance of two points $p, q \in \mathbb{R}^2$ depends on the parameters ϵ and λ and was defined by Myers and Miller [15] as follows:

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

However, 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 [15,20], 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-pair 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 below or on the straight line $x_2 = x_1 + (s.x_2 - s.x_1)$; see Fig. 5. 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 above 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 `RMQ`, 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 [7]. 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 p is in the first octant of point s if and only if $T_1(p)$ is in the first quadrant of $T_1(s)$.[3] 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 one has 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 9. *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 8.

In Section 4.1, we dealt with the geometric cost gc by modifying the field $f.score$. This is not possible here because 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

$$\{T_i(end(f).x_2, \ldots, end(f).x_k) \mid f \text{ is a fragment}\}$$

[3] 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 4, then we activate point $T_1(end(f').x_2, \ldots, end(f').x_k)$ in D_1 with priority $f'.score - gc_1(f')$ and point $T_2(end(f').x_2, \ldots, end(f').x_k)$ in D_2 with priority $f'.score - gc_2(f')$. If we encounter the start point of fragment f', then we launch two RMQ, namely $\texttt{RMQ}(0, T_1(beg(f').x_2, \ldots, beg(f').x_k))$ in the data structure D_1 and analogously $\texttt{RMQ}(0, T_2(beg(f').x_2, \ldots, beg(f').x_k))$ 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 such that $T_i(end(f_i).x_2, \ldots, end(f_i).x_k) < T_i(beg(f').x_2, \ldots, beg(f').x_k)$, where $1 \leq i \leq 2$. 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 9, 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 $f'.score := f'.weight + v_2$.

The case $k > 2$**:** In this case, the sum-of-pair 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

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

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 10. *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 $T_1(s)$

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

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 9 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 11 generalizes the 2-dimensional chaining algorithm described above to k dimensions.

Algorithm 11 *k-dim. chaining of n fragments w.r.t. the sum-of-pair 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 (ignoring their x_1 coordinate) as inactive in the $(k-1)$-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$(0, T_j(\text{points}[i].x_2, \ldots, \text{points}[i].x_k))$ *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 $\backslash\star$ points$[i]$ *is end point of a fragment f'* $\star\backslash$
    for $j := 1$ **to** $k!$
      activate $T_j(\text{points}[i].x_2, \ldots, \text{points}[i].x_k))$ in D_j with priority $f'.score - gc_j(f')$

For every start point $beg(f')$ of a fragment f', Algorithm 11 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` because the first hyper-corner is divided into $k!$ hyper-regions. Analogously, for every end point $end(f')$ of a fragment f', Algorithm 11 performs $k!$ activation operations. Therefore, the total time complexity of Algorithm 11 is $O(k!\, n \log^{k-2} n \log\log n)$ and its space requirement is $O(k!\, n \log^{k-2} n)$. This result improves the running time of Myers and Miller's algorithm [15] by a factor $\frac{\log^2 n}{\log\log n}$ and the space requirement by one log-factor.

5 Conclusions

In this paper, we have presented a line-sweep algorithm that solves the fragment chaining problem of multiple genomes. For $k > 2$ genomes, our algorithm takes

- $O(n \log^{k-2} n \log\log n)$ 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,
- $O(k!\, n \log^{k-2} n \log\log n)$ time and $O(k!\, n \log^{k-2} n)$ space for the sum-of-pair gap cost.

For $k = 2$, it takes $O(n \log n)$ time and $O(n)$ space for any of the above-mentioned gap costs.

This solves the open problem of reducing the time complexity of Myers and Miller's [15] chaining algorithm. Specifically, our algorithm reduces the time complexity of their algorithm by a factor $O(\frac{\log^2}{\log\log n})$ and the space complexity by a log factor. Myers and Miller did not provide an implementation of their chaining algorithm, but we are currently implementing ours. To find the chaining algorithm that performs best in practice, we are planning to conduct experiments that compare the running times of various chaining algorithms, including the algorithms that are based on *kd*-trees.

It is worth-mentioning that the longest common subsequence (LCS) from fragments problem can also be solved within our framework. This generalizes the algorithm of [1] to more than two sequences.

References

1. B.S. Baker and R. Giancarlo. Longest common subsequence from fragments via sparse dynamic programming. In *Proc. 6th European Symposium on Algorithms*, LNCS 1461, pp. 79–90, 1998.
2. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.
3. 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.
4. D. Eppstein. `http://www.ics.uci.edu/~eppstein/pubs/p-sparsedp.html`.

5. D. Eppstein, R. Giancarlo, Z. Galil, 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.
6. M.S. Gelfand, A.A. Mironov, and P.A. Pevzner. Gene recognition via spliced sequence alignment. *Proc. Nat. Acad. Sci.*, 93:9061–9066, 1996.
7. 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.
8. 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.
9. D. Joseph, J. Meidanis, and P. Tiwari. Determining DNA sequence similarity using maximum independent set algorithms for interval graphs. *Proc. 3rd Scandinavian Workshop on Algorithm Theory*, LNCS 621, pp. 326–337, 1992.
10. M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18:S312–S320, 2002.
11. M.-Y. Leung, B.E. Blaisdell, C. Burge, and S. Karlin. An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *Journal of Molecular Biology*, 221:1367–1378, 1991.
12. W. Miller. Comparison of genomic DNA sequences: Solved and unsolved problems. *Bioinformatics*, 17:391–397, 2001.
13. B. Morgenstern. A space-efficient algorithm for aligning large genomic sequences. *Bioinformatics* 16:948–949, 2000.
14. 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.
15. E.W. Myers and W. Miller. Chaining multiple-alignment fragments in sub-quadratic time. *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, pp. 38–47, 1995.
16. F.P. Preparata and M.I. Shamos. *Computational geometry: An introduction.* Springer-Verlag, New York, 1985.
17. S. Schwartz, Z. Zhang, K.A. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and W. Miller. PipMaker—A web server for aligning two genomic DNA sequences., *Genome Research*, 4(10):577–586, 2000.
18. 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.
19. D.E. Willard. New data structures for orthogonal range queries. *SIAM Journal of Computing*, 14:232–253, 1985.
20. Z. Zhang, B. Raghavachari, R. Hardison, and W. Miller. Chaining multiple-alignment blocks. *Journal of Computational Biology*, 1:51–64, 1994.