Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

OXFORD

Sequence Analysis

Co-linear chaining with overlaps and gap costs

Chirag Jain [1,*], Daniel Gibney [2] and Sharma V. Thankachan [2]

[1]Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India and
[2]Department of Computer Science, University of Central Florida, Orlando, USA.

*To whom correspondence should be addressed.

Abstract

Motivation: Co-linear chaining has proven to be a powerful technique for finding approximately optimal alignments and approximating edit distance. It is used as an intermediate step in numerous mapping tools that follow seed-and-extend strategy. Despite this popularity, subquadratic time algorithms for the case where chains support anchor overlaps and gap costs are not currently known. Moreover, a theoretical connection between co-linear chaining cost and edit distance remains unknown.
Results: We present algorithms to solve the co-linear chaining problem with anchor overlaps and gap costs in $\tilde{O}(n)$ time, where n denotes the count of anchors. We establish the first theoretical connection between co-linear chaining cost and edit distance. Specifically, we prove that for a fixed set of anchors under a carefully designed chaining cost function, the optimal 'anchored' edit distance equals the optimal co-linear chaining cost. Finally, we demonstrate experimentally that optimal co-linear chaining cost under the proposed cost function can be computed significantly faster than edit distance, and achieves high correlation with edit distance for closely as well as distantly related sequences.
Implementation: https://github.com/at-cg/ChainX
Contact: chirag@iisc.ac.in, daniel.j.gibney@gmail.com, sharma.thankachan@ucf.edu

1 Introduction

Computing an optimal alignment between two sequences is one of the most fundamental problems in computational biology. Unfortunately, conditional lower-bounds suggest that an algorithm for computing an optimal alignment, or edit distance, in strongly subquadratic time is unlikely (Backurs and Indyk (2015)). This lower-bound indicates a challenge for scaling the computation of edit distance to applications like high-throughput sequencing. Instead, heuristics are often used to obtain an approximate solution in less time and space. One such popular heuristic is chaining. This technique involves precomputing fragments between the two sequences that closely agree (in this work, exact matches called anchors), then determining which of these anchors should be kept within the alignment (See Fig. 1). Techniques along these lines are used in many tools, such as long-read mappers like Minimap2 (Li (2018)), Minigraph (Li *et al.* (2020)), Winnowmap2 (Jain *et al.* (2020)), uLTRA RNA-seq aligner (Sahlin and Makinen (2020)), IRA (Ren and Chaisson (2020)), NGMLR (Sedlazeck *et al.* (2018)), BLASR (Chaisson and Tesler (2012)) and generic alignment tools like Nucmer (Kurtz *et al.* (2004), Marçais *et al.* (2018)), CLASP (Otto *et al.* (2011)), and CoCoNUT (Abouelhoda *et al.* (2008)). We will focus on the following problem (described formally in Section 2): Given a set of n anchors, determine an optimal ordered subset (or chain) of these anchors.

There are several previous works that develop algorithms for the co-linear chaining problem (Abouelhoda and Ohlebusch (2005); Mäkinen and Sahlin (2020); Uricaru *et al.* (2011); Shibuya and Kurochkin (2003)) and even more in the context of sparse dynamic programming (Wilbur and Lipman (1983); Eppstein *et al.* (1992a,b); Myers and Miller (1995); Morgenstern (2002)). Solutions with different time complexities exist for different variations of this problem. These depend on the cost-function assigned to a chain and the types of chains permitted. Solutions include an algorithm running in time $O(n \log n \log \log n)$ for a variant of the problem where anchors used in a solution must be non-overlapping (Abouelhoda and Ohlebusch (2005)). More recently, an algorithm running in $O(n \log^2 n)$ time was given where overlaps are allowed, but gaps between anchors are not considered in the cost-function (Mäkinen and Sahlin (2020)). None of the solutions introduced thus far provide a subquadratic time algorithm for variations that use both overlap and gap costs. However, including overlaps and gaps into a cost-function is arguably a more realistic model for anchor chaining. Depending on the type of anchor, there may be no reason to assume that in an optimal alignment the anchors would be non-overlapping. At the same time, not taking into account large gaps between the anchors seems unlikely to produce optimal alignments.

This work's contribution is the following:

- We provide the first algorithm running in subquadratic, $\widetilde{O}(n)$ time for chaining with overlap and gap costs[1]. Refinements based on the specific type of anchor and chain under consideration are also given. These refinements include an $O(n \log^2 n)$ time algorithm for the case where all anchors are of the same length, as is the case with k-mers.
- When n is not too large (less than the sequence lengths), we present an algorithm with $O(n \cdot OPT + n \log n)$ average-case time. This provides a simple algorithm that is efficient in practice.
- Using a carefully designed cost-function, we mathematically relate the optimal chaining cost with a generalized version of edit distance, which we call *anchored edit distance*. This is equivalent to the usual edit distance with the modification that matches performed without the support of an anchor have unit cost. A more formal definition appears in Section 2. With our cost function, we prove that over a fixed set of anchors the optimal chaining cost is equal to the anchored edit distance.
- We empirically demonstrate that computing the optimal chaining cost can be orders of magnitude faster than computing edit distance, especially in semi-global comparison mode. We also demonstrate a strong correlation between optimal chaining cost and edit distance. The observed correlation coefficients are favorable when compared to chaining heuristics implemented within commonly used sequence mappers such as Nucmer4 and Minimap2.

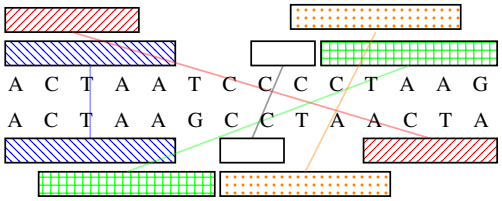


Fig. 1. Anchors representing exact matches are shown as rectangles. The co-linear chaining problem is to find an optimal ordered subset of anchors subject to some cost function.

2 Concepts and definitions

Let S_1 and S_2 be two strings of lengths $|S_1|$ and $|S_2|$ respectively. An anchor interval pair $([a..b], [c..d])$ signifies a match between $S_1[a..b]$ and $S_2[c..d]$. For an anchor I, we denote these values as $I.a$, $I.b$, $I.c$, and $I.d$. We assume that an anchor corresponds to an exact match, therefore $b-a$ equals $d-c$. Accordingly, $S_1[a+j] = S_2[c+j]$ for any $0 \leq j \leq b-a = d-c$. We say that the character match $S_1[a+j] = S_2[c+j]$ is *supported* by the anchor $([a..b], [c..d])$. Maximal exact matches (MEMs), maximal unique matches (MUMs), or k-mer matches are some of the common ways to define anchors. Maximal unique matches (Delcher *et al.* (1999)) are a subset of maximal exact matches, having the added constraint that the pattern involved occurs only once in both strings. If all intervals across all anchors have the same length (e.g., using k-mers), we say that the *fixed-length* property holds.

Our algorithm will make use of range minimum queries (RmQs). For a set of n d-dimensional points, each with an associated weight, a 'query' consists of an orthogonal d-dimensional range. The query response is the point in that range with the smallest weight. Using standard techniques, a data structure can be built in $O(n \log^{d-1} n)$ time, that can both answer queries and modify a point's weight in $O(\log^d n)$ time (de Berg *et al.* (2008)).

[1] $\widetilde{O}(\cdot)$ hides poly-logarithmic factors.

2.1 Co-linear chaining problem with overlap and gap costs

Given a set of n anchors $\mathcal{A}$ for strings S_1 and S_2, we assume that $\mathcal{A}$ already contains two *end-point* anchors $\mathcal{A}_{left} = ([0,0],[0,0])$ and $\mathcal{A}_{right} = ([|S_1|+1, |S_1|+1], [|S_2|+1, |S_2|+1])$. We define the strict precedence relationship $\prec$ between two anchors $I' := \mathcal{A}[j]$ and $I := \mathcal{A}[i]$ as $I' \prec I$ if and only if $I'.a \leq I.a$, $I'.b \leq I.b$, $I'.c \leq I.c$, $I'.d \leq I.d$, and strict inequality holds for at least one of the four inequalities. In other words, the interval belonging to I' for S_1 (resp. S_2) should start before or at the starting position of the interval belonging to I for S_1 (resp. S_2) and should not extend past it. We also define the weak precedence relation $\prec_w$ as $I' \prec_w I$ if and only if $I'.a \leq I.a$, $I'.c \leq I.c$ and strict inequality holds for at least one of the two inequalities, i.e., intervals belonging to I' should start before or at the starting position of intervals belonging to I, but now intervals belonging to I' can be extended past the intervals belonging to I. The aim of the problem is to find a totally ordered subset (a chain) of $\mathcal{A}$ that achieves the minimum cost under the cost function presented next. We will specify whether we mean a chain under strict precedence or a chain under weak precedence where necessary.

Cost function. For $I' \prec I$, the function $connect(I', I)$ is designed to indicate the cost of connecting anchor I' to anchor I in an alignment. The chaining problem asks for a chain of $m \leq n$ anchors, $\mathcal{A}'[1]$, $\mathcal{A}'[2]$, ..., $\mathcal{A}'[m]$, such that the following properties hold: (i) $\mathcal{A}'[1] = \mathcal{A}_{left}$, (ii) $\mathcal{A}'[m] = \mathcal{A}_{right}$, (iii) $\mathcal{A}'[1] \prec \mathcal{A}'[2] \prec \ldots \prec \mathcal{A}'[m]$, and (iv) the cost $\sum_{i=1}^{m-1} connect(\mathcal{A}'[i], \mathcal{A}'[i+1])$ is minimized.

We next define the function $connect$. In Section 3.2, we will see that this definition is well motivated by the relationship with anchored edit distance. For a pair of anchors I', I such that $I' \prec I$:

- The gap in string S_1 between anchors I' and I is $g_1 = \max(0, I.a - I'.b - 1)$. Similarly, the gap between the anchors in string S_2 is $g_2 = \max(0, I.c - I'.d - 1)$. We define the gap cost as $g(I', I) = \max(g_1, g_2)$.
- The overlap o_1 is defined such that $I'.b - o_1$ reflects the non-overlapping prefix length of anchor I' in string S_1. Specifically, $o_1 = \max(0, I'.b - I.a + 1)$. Similarly, define $o_2 = \max(0, I'.d - I.c + 1)$. We define the overlap cost as $o(I', I) = |o_1 - o_2|$.
- Lastly, define $connect(I', I) = g(I', I) + o(I', I)$.

The same definitions are used for weak precedence, only using $\prec_w$ in the place of $\prec$.

Regardless of the definition of $connect$, the above problem can be trivially solved in $O(n^2)$ time and $O(n)$ space. First sort the anchors by the component $\mathcal{A}[\cdot].a$ and let $\mathcal{A}'$ be the sorted array. The chaining problem then has a direct dynamic programming solution by filling an n-sized array C from left-to-right, such that $C[i]$ reflects the cost of an optimal chain that ends at anchor $\mathcal{A}'[i]$. The value $C[i]$ is computed using the recursion: $C[i] = \min_{\mathcal{A}'[k] \prec \mathcal{A}'[i]} \left(C[k] + connect(\mathcal{A}'[k], \mathcal{A}'[i]) \right)$ where the base case associated with anchor $\mathcal{A}_{left}$ is $C[1] = 0$. Array C is computed using two nested loops, resulting in a time complexity of $O(n^2)$. The optimal chaining cost will be stored in $C[n]$. We will provide an $O(n \log^4 n)$ time algorithm for this problem for the case where the $connect$ function is the one stated above.

2.2 Anchored edit distance

The edit distance problem is to identify the minimum number of operations (substitutions, insertions, or deletions) that must be applied to string S_2 to transform it to S_1. Edit operations can be equivalently represented as an alignment (a.k.a. edit transcript) that specifies the associated matches, mismatches and gaps while placing one string on top of another. The *anchored edit distance problem* is as follows: given strings S_1 and S_2 and a set of n anchors $\mathcal{A}$, compute the optimal edit distance

subject to the condition that a match supported by an anchor has edit cost 0, and a match that is not supported by an anchor has an edit cost of 1.

The above problem is easily solvable in $O(|S_1||S_2|)$ time and space. First, observe that we only need to consider $O(|S_1||S_2|)$ anchors in total because only the longest anchor is useful for a fixed pair of indices i and j. Next, the standard dynamic programming recursion for solving the edit distance problem can be revised. Let $D[i,j]$ denote anchored edit distance between $S_1[1,i]$ and $S_2[1,j]$, then $D[i,j] = \min(D[i-1,j-1] + x, D[i-1,j]+1, D[i,j-1]+1)$ where $x = 0$ if $S_1[i] = S_2[j]$ and the match is supported by some anchor, and $x = 1$ otherwise.

2.3 Graph representation

It is useful to consider the following representation of an alignment of two strings S_1 and S_2. As illustrated in Figure 2, we have a set of $|S_1|$ top vertices and $|S_2|$ bottom vertices. There are two types of edges between the top and bottom vertices: (i) A solid edge from ith top vertex to the jth bottom vertex. This represents an anchor supported match between the ith symbol in S_1 and the jth symbol in S_2; (ii) A dashed edge from the ith top vertex to the jth bottom vertex. This represents a character being substituted to form a match between $S_1[i]$ and $S_2[j]$ or an exact match not supported by an anchor. All unmatched vertices are labeled with an 'x' to indicate that the corresponding symbol is deleted. An important observation is that no two edges cross in this graph representation.

In a solution to the anchored edit distance problem every solid edge must be 'supported' by an anchor. By 'supported' here we mean that the match between the corresponding symbols in S_1 and S_2 is supported by an anchor. In Figure 2, these anchors are represented with rectangles above and below the vertices. We use $\mathcal{M}$ to denote the set of vertices marked with x and all edges. We also associate an edit cost with the alignment, denoted as $EDIT(\mathcal{M})$. This is defined as the number of vertices marked with x in $\mathcal{M}$ plus the number of dashed edges in $\mathcal{M}$.

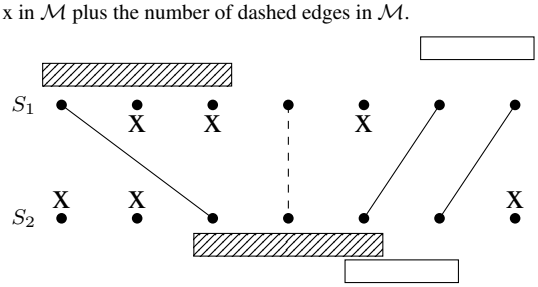


Fig. 2. Above is the graph representation of an alignment. Solid edges represent matches, dashed edges substitutions and unsupported matches, and 'x's deletions. We use the notation $\mathcal{M}$ to represent the set of vertices marked with x and all edges. Here $EDIT(\mathcal{M}) = 7$.

3 Main results

Theorem 3.1 states formally the algorithmic results for solving the co-linear chaining problem with overlap and gap costs.

Theorem 3.1. *The co-linear chaining problem with overlap and gap costs can be solved in time $\widetilde{O}(n)$. In particular, in time $O(n \log^2 n)$ for chains with fixed-length anchors; in time $O(n \log^3 n)$ for chains under weak precedence; and in time $O(n \log^4 n)$ for chains under strict precedence.*

We give a brief outline of the algorithm before its formal description in Section 3.1. The proposed algorithm still uses the recursive formula given in Section 2.1. However, it uses range minimum query (RmQ) data structures to avoid having to check every anchor $\mathcal{A}[k]$ where $\mathcal{A}[k].a < \mathcal{A}[i].a$. We avoid this exhaustive check by considering six cases concerning the optimal choice of the prior anchor. We use the best of the six distinct possibilities to determine the optimal $C[i]$ value. This $C[i]$ value is then used to update the RmQ data structures. For the strict precedence case, some of the six cases require up to four dimensions for the range minimum queries. When only weak precedence is required, we reduce this to at most three dimensions. When the fixed-length property holds (e.g., k-mers), we reduce this to two dimensions.

Next, Theorem 3.2 formalizes the connection that this work makes between co-linear chaining and anchored edit distance.

Theorem 3.2. *For a fixed set of anchors $\mathcal{A}$, the following quantities are equal: the optimal anchored edit distance, the optimal co-linear chaining cost under strict precedence, and the optimal co-linear chaining cost under weak precedence.*

The proof of Theorem 3.2 (Section 3.2) first shows that for a fixed set of anchors, the anchored edit distance of two sequences S_1 and S_2 is at most the optimal co-linear chaining cost. We do this by starting with an optimal chain of anchors under weak precedence and providing an alignment $\mathcal{M}_G$ where $EDIT(\mathcal{M}_G)$ is at most the optimal co-linear chaining cost. The alignment is obtained using a greedy algorithm that works from left-to-right, always taking the closest exact match when possible, and when not possible, a character substitution or unsupported exact match, or if none of these are possible, a deletion. The proof of the inequality uses this algorithm along with induction on the number of anchors processed.

To prove the other side of the inequality, we begin with an optimal alignment $\mathcal{M}^*$. We then find a set of anchors that are totally ordered under strict precedence and support an alignment $\mathcal{M}$ where $EDIT(\mathcal{M}) = EDIT(\mathcal{M}^*)$. Following this, we show that on any set of anchors totally ordered under strict precedence (i) the greedy algorithm outlined above for obtaining an alignment $\mathcal{M}_G$ is optimal, and (ii) $EDIT(\mathcal{M}_G)$ is the same value as the anchor chaining cost.

3.1 Algorithm for co-linear chaining with gap and overlap

3.1.1 Algorithm for chains under strict precedence

We first present the algorithm for chains under strict precedence and then provide modifications for the other cases. The first step is to sort the set of anchors $\mathcal{A}$ using the key $\mathcal{A}[\cdot].a$. Let $\mathcal{A}'$ be the sorted array. We will next use six RmQ data structures labeled $\mathcal{T}_{1a}, \mathcal{T}_{1b}, \mathcal{T}_{2a}, \mathcal{T}_{2b}, \mathcal{T}_{3a}, \mathcal{T}_{3b}$. These RmQ data structures are initialized with the following points for every anchor: For anchor $I \in \mathcal{A}'$: $\mathcal{T}_{1a}$ is initialized with the point $(I.b, I.d - I.b)$, $\mathcal{T}_{1b}$ with $(I.d, I.d - I.b)$, $\mathcal{T}_{2a}$ with $(I.b, I.c, I.d)$, $\mathcal{T}_{2b}$ with $(I.b, I.d)$, $\mathcal{T}_{3a}$ with $(I.b, I.c, I.d, I.d - I.b)$, and $\mathcal{T}_{3b}$ with $(I.b, I.d, I.d - I.b)$. All weights are initially set to ∞.

We then process the anchors in sorted order and update the RmQ data structures after each iteration. On the ith iteration, for $j < i$, we let $C[j]$ be the optimal co-linear chaining cost of any ordered subset of $\mathcal{A}'[1]$, $\mathcal{A}'[2]$, ..., $\mathcal{A}'[j]$ that ends with $\mathcal{A}'[j]$. For $i > 1$, RmQ queries are used to find the optimal $j < i$ by considering six different cases. We let $I = \mathcal{A}'[i]$, $I' = \mathcal{A}'[j]$, and $C[I'] = C[j]$.

1. Case: I' disjoint from I.

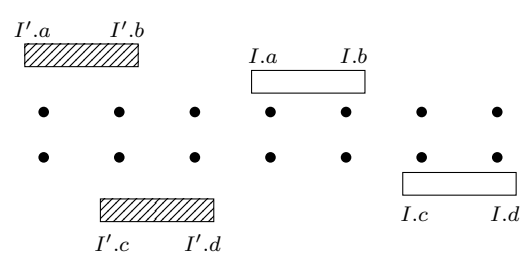


Fig. 3. Case 1.a. Colinear chaining cost is $C[I'] + g_2 = C[I'] + I.c - I'.d - 1$

a. Case: The gap in S_1 is less or equal to gap in S_2 (Fig. 3). The RmQ is of the form: $(I'.b, I'.d - I'.b) \in [0, I.a-1] \times [-\infty, I.c - I.a]$. The weight stored in $\mathcal{T}_{1a}$ for I' is $\min C[I'] - I'.d$. We query the range above and let $C_{1a} = \min C[I'] + I.c - I'.d - 1$.

b. Case: The gap in S_2 is less than gap in S_1 The RmQ is of the form: $(I'.d, I'.d - I'.b) \in [0, I.c-1] \times [I.c - I.a + 1, \infty]$. The weight stored in $\mathcal{T}_{1b}$ for I' is $\min C[I'] - I'.b$. We query the range above and let $C_{1b} = \min C[I'] + I.a - I'.b - 1$.

2. Case: I' and I overlap in only one dimension.

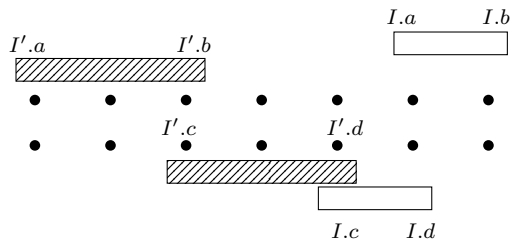


Fig. 4. Case 2.a. Chaining cost is $C[I'] + g_1 + o_2 = C[I'] + I.a - I'.b + I'.d - I.c$.

a. Case: I' and I overlap only in S_2 (Fig 4). The RmQ is of the form: $(I'.b, I'.c, I'.d) \in [0, I.a-1] \times [0, I.c] \times [I.c, I.d]$. The weight stored in $\mathcal{T}_{2a}$ for I' is $\min C[I'] - I'.b + I'.d$. We query the range above and let $C_{2a} = C[I'] + I.a - I'.b + I'.d - I.c$.

b. Case: I' and I overlap only in S_1. Since the anchors are sorted on $\mathcal{A}[\cdot].a$, this can be done with a two dimensional RmQ structure. The RmQ is of the form: $(I'.b, I'.d) \in [I.a, I.b] \times [0, I.c-1]$. The weight stored in $\mathcal{T}_{2b}$ for I' is $\min C[I'] + I'.b - I'.d$. We query the range above and let $C_{2b} = C[I'] + I.c - I'.d + I'.b - I.a$.

3. Case: I' and I overlap in both dimensions.

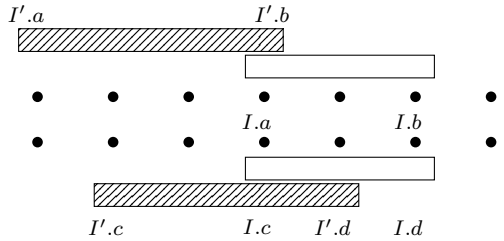


Fig. 5. Chaining cost is $C[I'] + o_2 - o_1 = C[j] + I'.d - I.c - (I'.b - I.a)$.

a. Case: Greater overlap in S_2 (Fig. 5). Here, $|o_1 - o_2| = o_2 - o_1 = I'.d - I.c - (I'.b - I.a)$. The RmQ is of the form:

$$(I'.b, I'c, I'.d, I'.d - I'.b) \in [I.a, I.b] \times [0, I.c] \times [I.c, I.d] \times [I.c - I.a + 1, \infty].$$

The weight stored in $\mathcal{T}_{3a}$ for I' is $\min C[I'] - I'.b + I'.d$. We query the range above and let $C_{3a} = C[I'] + I'.d - I.c - I'.b + I.a$.

b. Case: Greater or equal overlap in S_1. Here, $|o_1 - o_2| = o_1 - o_2 = I'.b - I.a - (I'.d - I.c)$. If $o_1 \geq o_2 > 0$, $I'.b \in [I.a, I.b]$, and $I'.a \in [0, I.a]$ then $I'.c \in [0, I.c]$. Hence, we can make the RmQ of the form:

$$(I'.b, I'.d, I'.d - I'.b) \in [I.a, I.b] \times [I.c, I.d] \times [-\infty, I.c - I.a].$$

The weight stored in $\mathcal{T}_{2b}$ for I' is $\min C[I'] + I'.b - I'.d$. We query the range above and let $C_{3b} = C[I'] + I'.b - I.a + I.c - I'.d$.

Finally, set $C[i] = \min(C_{1a}, C_{1b}, C_{2a}, C_{2b}, C_{3a}, C_{3b})$ and update the RmQ structures as shown in the Pseudo-code in Figure 3.1.1. In the pseudo-code, an RmQ structure $\mathcal{T}$ has the query method $\mathcal{T}.RmQ()$ which takes as arguments an interval for each dimension. It also has the method $\mathcal{T}.update()$, which takes a point and a weight and then updates the point to have the new weight. The four-dimensional queries for Case 3.a require $O(\log^4 n)$ time per query, causing an $O(n \log^4 n)$ time complexity.

Input: n anchors $\mathcal{A}[1, n]$ including $\mathcal{A}_{left} = \mathcal{A}[1]$ and $\mathcal{A}_{right} = \mathcal{A}[n]$.
Output: Array $C[1, n]$ s.t. $C[i]$ is the optimal co-linear chaining cost for any ordered subset of $\mathcal{A}[1, i]$ ending with $\mathcal{A}[i]$.
Let $\mathcal{A}'[1]$, ... $\mathcal{A}'[n]$ be the anchors $\mathcal{A}$ sorted on $\mathcal{A}[\cdot].a$;
Construct RmQ structures with weights set to ∞;
Initialize array C of size n to 0;
for $i \leftarrow 1$ **to** n **do**
  $I \leftarrow \mathcal{A}'[i]$;
  if $i \geq 2$ **then**
    $C_{1a} \leftarrow \mathcal{T}_{1a}.RmQ([0, I.a-1], [-\infty, I.c - I.a]) + I.c - 1$;
    $C_{1b} \leftarrow \mathcal{T}_{1b}.RmQ([0, I.c-1], [I.c - I.a + 1, \infty]) + I.a - 1$;
    $C_{2a} \leftarrow \mathcal{T}_{2a}.RmQ([0, I.a-1], [0, I.c], [I.c, I.d]) + I.a - I.c$;
    $C_{2b} \leftarrow \mathcal{T}_{2b}.RmQ([I.a, I.b], [0, I.c-1]) - I.a + I.c$;
    $C_{3a} \leftarrow \mathcal{T}_{3a}.RmQ([I.a, I.b], [0, I.c], [I.c, I.d], [I.c - I.a + 1, \infty]) + I.a - I.c$;
    $C_{3b} \leftarrow \mathcal{T}_{3b}.RmQ([I.a, I.b], [I.c, I.d], [-\infty, I.c - I.a]) - I.a + I.c$;
    `/* Take optimal choice */`
    $C[i] \leftarrow \min(C_{1a}, C_{1b}, C_{2a}, C_{2b}, C_{3a}, C_{3b})$;
  end
  `/* Update RmQ structures */`
  $\mathcal{T}_{1a}.update((I.b, I.d - I.b), C[i] - I.d)$;
  $\mathcal{T}_{1b}.update((I.d, I.d - I.b), C[i] - I.b)$;
  $\mathcal{T}_{2a}.update((I.b, I.c, I.d), C[i] - I.b + I.d)$;
  $\mathcal{T}_{2b}.update((I.b, I.d), C[i] + I.b - I.d)$;
  $\mathcal{T}_{3a}.update((I.b, I.c, I.d, I.d - I.b), C[i] - I.b + I.d)$;
  $\mathcal{T}_{3b}.update((I.b, I.d, I.d - I.b), C[i] + I.b - I.d)$;
end
return $C[1, n]$

Algorithm 1: For co-linear chaining with overlaps and gap costs

3.1.2 Modifications for weak precedence and fixed-length anchors

We first consider the case of weak precedence. In Case 3.a the anchor end $I'.d$ can be positioned arbitrarily to the right of $I.c$. Moreover, since by the first dimension of the RmQ there is positive overlap in S_1 and by the fourth dimension there is greater overlap in S_2, we know that $I'.d \geq I.c$. Hence, we can then remove the third dimension from the RmQ. The query will then be of the form $(I'.b, I'.c, I'.d - I'.b) \in [I.a, \infty] \times [0, I.c] \times [I.c - I.a + 1, \infty]$. In Case 3.b, where there is greater or equal overlap in S_1, we can similarly ignore $I'.b$, but in order to match our definition of weak precedence we must also ensure $I'.c \in [0, I.c - 1]$ (this is unnecessary for $I'.a$ in Case 3.a as the strictly greater overlap in S_2 ensures $I'.a < I.a$). We modify the query to be of the form $(I'.c, I'.d, I'.d - I'.b) \in [0, I.c - 1] \times [I.c, \infty] \times [-\infty, I.c - I.a]$. Since each RmQ has at most three dimensions the total time complexity can be brought down to $O(n \log^3 n)$.

In the case of fixed-length anchors, the RmQ for Case 2.a. can be made $(I'.b, I'.d) \in [0, I.a - 1] \times [I.c, I.d]$. The modifications for Cases 3.a and 3.b are more involved. When solving for $C[I]$, the points associated with I in $\mathcal{T}_{3a}$ and $\mathcal{T}_{3b}$ are updated as was done previously. We keep a pointer p_a to indicate the current a value of the interval, initially setting $p_a = \mathcal{A}[1].a$. Conceptually, before processing anchor I we increment p_a from its previous position to $I.a$. If for some anchor I' the end $I'.b$ is passed by p_a, we update the points associated with I' in $\mathcal{T}_{3a}$ and $\mathcal{T}_{3b}$ to have the the weight ∞. This eliminates the need to use a range query to check $I'.b \in [I.a, I.b]$, since any points not within that range are effectively removed from consideration. Hence, we can reduce the query for Case 3.a. (overlap in S_2 greater than overlap in S_1) to $(I'.c, I'.d -$

$I'.b) \in [0, I.c] \times [I.c - I.a + 1, \infty]$. and the query for Case 3.b (overlap in S_1 greater or equal to overlap in S_2) to $(I'.d, I'.d - I'.b) \in [I.c, I.d] \times [-\infty, I.c - I.a]$. To avoid the $|S_1|$ time complexity, the anchors that would be encountered while incrementing p_a can be found by looking at which anchors have b values between the previous p_a value and $I.a$. Because each update requires $O(\log^2 n)$ time, these updates cost time $O(n \log^2 n)$ in total.

3.2 Equivalence of anchored edit distance and chaining

Lemma 3.3. *Anchored edit distance $\leq$ optimal co-linear chaining cost under weak precedence $\leq$ optimal co-linear chaining cost under strict precedence.*

Proof. We start with an anchor chain under weak precedence, $\mathcal{A}[1]$, $\mathcal{A}[2]$, ..., with optimal co-linear chaining cost. We will provide an alignment with an associate anchored edit distance that is at most the chaining cost. Assume inductively that all symbols in $S_1[1, \mathcal{A}[i].b]$ and $S_2[1, \mathcal{A}[i].d]$ have been processed, that is, either matched, substituted, or deleted (represented by check-marks in Figures 6-10). The base case of this induction holds trivially for $\mathcal{A}_{left}$. We consider the anchor $\mathcal{A}[i+1]$ and the possible cases regarding its position relative to $\mathcal{A}[i]$. Symmetric cases that only swap the roles of S_1 and S_2 are ignored. To ease notation, let $I' = \mathcal{A}[i]$ and $I = \mathcal{A}[i+1]$.

1. **Case** $I'.b \geq I.b$ and $I'.d \geq I.c$ (Fig 6): To continue the alignment,

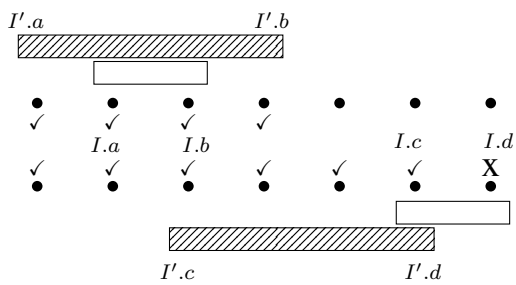


Fig. 6. Case $I', b \geq I.b$ and $I'.d \geq I.c$, the ✓ symbol is used to indicate symbols that have been processed prior to considering I.

delete the substring $S_2[I'.d + 1, I.d]$ from S_2. This has an edit cost of $I.d - I'.d$. We also have $connect(I', I) = o_1 - o_2 = I.c + I'.b - I.a - I'.d \geq I.c + I.b - I.a - I'.d = I.d - I'.d$.

2. **Case** $I'.b \geq I.b$ and $I'.d < I.c$ (Fig 7): Delete the substring

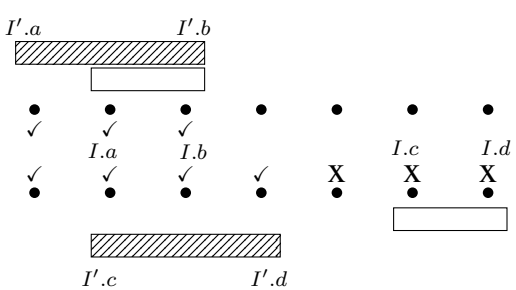


Fig. 7. $I'.b \geq I.b$ and $I'.d < I.c$.

$S_2[I'.d + 1, I.d]$ from S_2, with edit cost $I.d - I'.d$. Also $connect(I', I) = o_1 + g_2 = I.c + I'.b - I.a - I'.d \geq I.c + I.b - I.a - I'.d = I.d - I'.d$.

3. **Case** $I.b > I'.b$, $I.a \leq I'.b$, $I.c \leq I'.d$ (Fig. 8): Supposing wlog that $o_1 > o_2$, delete the substring $S_2[I'.d + 1, I'.d + o_1 - o_2]$, and match the substrings $S_1[I'.b + 1, I.b]$ and $S_2[I'.d + o_1 - o_2 + 1, I.d]$. This has edit cost $o_1 - o_2$. Also, $connect(I', I) = o_1 - o_2$.
4. **Case** $I.b > I'.b$, $I.a \leq I'.b$, $I.c > I'.d$ (Fig. 9): We delete the substring $S_2[I'.d + 1, I'.d + o_1 + g_2]$ and match $S_1[I'.b + 1, I.b]$

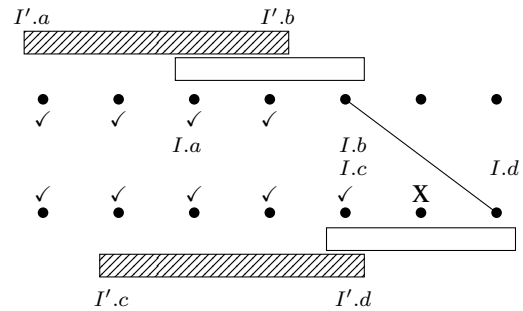


Fig. 8. Case $I.b > I'.b$, $I.a \leq I'.b$, $I.c \leq I'.d$

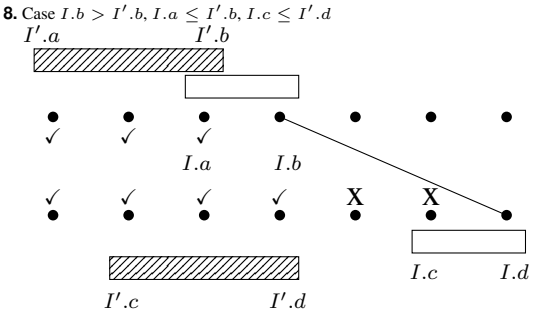


Fig. 9. Case $I.b > I'.b$, $I.a \leq I'.b$, $I.c > I'.d$

with $S_2[I'.d + o_1 + g_2 + 1, I.d]$. This has edit cost $o_1 + g_2$. Also, $connect(I', I) = o_1 + g_2$.

5. **Case** $I.a > I'.b$, $I.c > I'.d$ (Fig. 10): Supposing wlog

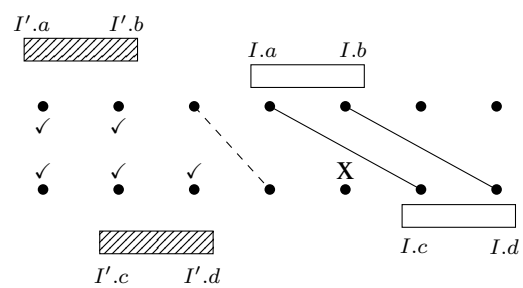


Fig. 10. Case $I.a > I'.b$, $I.c > I'.d$

$g_2 \geq g_1$, match with substitutions or unsupported exact matches $S_1[I'.b + 1, I'.b + g_1]$ and $S_2[I'.d + 1, I'.d + g_1]$. Delete the substring $S_2[I'.d + g_1 + 1, I.c - 1]$. Finally, match $S_1[I.a, I.b]$ and $S_2[I.c, I.d]$. The edits consist of g_1 of substitutions or unsupported exact matches and $g_2 - g_1$ deletions, which is g_2 edits in total. Also, $connect(I', I) = \max\{g_1, g_2\} = g_2$.

Continuing this process until $\mathcal{A}_{right}$, all symbols in S_1 and S_2 become included in the alignment, proving the first inequality. The second inequality follows from the observation that every set of anchors ordered under strict precedence is also ordered under weak precedence. □

To prove the other side of the inequality, we first present an algorithm that can remove one anchor from a pair of incomparable anchors while maintaining an alignment of equal or less cost. Note that this algorithm is for the purposes of the proof and is not particularly efficient.

Algorithm (i). Algorithm for removing incomparable anchors. Consider two incomparable anchors I and I' (Fig 11). We process the edges supported by the two anchors from right-to-left. The anchor that has the rightmost supported solid edge will be the anchor we keep. Suppose wlog it is I. Working from right-to-left, for an edge $e = (S_1[h], S_2[k])$ (either solid or dashed) encountered that is not supported by I, we replace that edge with an edge supported by I. In particular, if the closest edge to the right of e supported by I is $e' = (S_1[h'], S_2[k'])$ and $h' - h \leq k' - k$,

then we replace e with $e'' = (S_1[h], S_2[k''])$ such that e'' is supported by I. Note that at least one side of every edge supported by I' is within an interval of I. Hence, all edges supported by I' are eventually replaced as we continue this process. We then remove I'. This algorithm is repeated until a total ordering under weak precedence is possible.

Algorithm (ii). Algorithm for removing anchors with nested intervals. Consider two anchors I and I' where I' has an interval nested in one of the intervals belonging to I. If an edge is supported by I to the right of any edges supported by I', we remove the edges supported by I' by again using a right-to-left sweep. We replace those edges as was done in Algorithm (i). If there exists an edge supported by I to the left of any edge supported by I', we remove the edges supported by I' by sweeping left-to-right, replacing those edges with edges supported by I. This is done by maintaining the vertex closest to the preceding I-supported edge to the left. Once this procedure is complete, I' no longer supports any edges and can be removed. We repeat this until there are no two nested intervals.

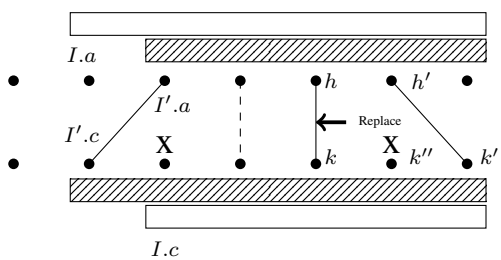


Fig. 11. Anchors I and I' are incomparable. To remove I' we sweep from right-to-left replacing edges not supported by I with edges supported by I. Here, $(S_1[h], S_2[k])$ is not supported by I and will be replaced with $(S_1[h], S_2[k''])$, which is supported by I.

Lemma 3.4. *For a set of anchors supporting an alignment $\mathcal{M}$, applying Algorithm (i) followed by Algorithm (ii) produces an anchor chain under strict precedence that supports an alignment $\mathcal{M}'$ where $EDIT(\mathcal{M}') \leq EDIT(\mathcal{M})$.*

Proof. Suppose we are replacing a solid edge $e = (S_1[h], S_2[k])$ supported by an anchor other than I, the anchor we wish to keep. Let $S_1[h]$ be the vertex closest to the nearest edge to the right supported by I, say $e' = (S_1[h'], S_2[k'])$ (Fig. 11). We replace e with the edge $e'' = (S_1[h], S_2[k''])$ supported by I'. Because edges are not crossing within the alignment, deleting vertex $S_2[k]$ and matching $S_2[k'']$, does not require modifying any additional edges, maintaining the edit cost. If e was a dashed edge, replacing e with e'' converts a substitution or unsupported exact match at $S_1[k]$ to a deletion, and removes a deletion at $S_2[k'']$, decreasing the edit cost by 1. Similar statements hold for Algorithm (ii) when we process edges from either right-to-left or from left-to-right. □

After applying Algorithms (i) and (ii) to an arbitrary supporting set of anchors, we have a supporting anchor chain under strict precedence. The next algorithm is the same as the one used in the proof of Lemma 3.3. Now, not having nested intervals makes the inequality strict in Cases 1 and 2.

Algorithm. Greedy algorithm for obtaining optimal alignment. Given a chain of anchors under strict precedence, we process the anchors from left-to-right, always using the left-most solid edge possible within the current anchor. In the case of a gap between anchors in S_1 and S_2, we perform substitutions or unsupported exact matches to match the left-most unprocessed characters in the gaps of both strings. The remaining characters in the larger of the two gaps are then deleted. In the case of a gap in only one string, the characters in the gap are deleted. See the proof of Lemma 3.3 for more details.

Lemma 3.5. *For a fixed anchor chain under strict precedence, the greedy algorithm produces an optimal alignment.*

Proof. This follows from an exchange argument. Suppose there exists an optimal alignment $\mathcal{M}^*$ that is not the same as the alignment $\mathcal{M}_G$ produced by the greedy algorithm. As we process the edges from left-to-right, consider when the first discrepancy in the edges is found, the first edge e in $\mathcal{M}^*$ not equal to an edge e_G in $\mathcal{M}_G$. We claim e can be replaced with e_G without increasing the edit cost. Let e_{prev} be the previous edge on which the $\mathcal{M}^*$ and $\mathcal{M}_G$ agreed. First, we observe that e and e_G must share at least one vertex. Since the greedy algorithm always matches when possible, them not sharing a vertex could only happen due to deletions on both S_1 and S_2 in $\mathcal{M}^*$ between e_{prev} and e. But in an optimal solution, this would never occur because substitutions could be used instead to reduce the edit cost. The edge e_G being dashed and the edge e being solid cannot happen because the greedy algorithm would take a solid edge whenever possible. Because only deletions exist between e_{prev} and e in whichever string the discrepancy exists, for any remaining combination of solid/dashed edge for e_G and solid/dashed edge for e, replacing e with e_G will not increase the edit cost. □

Lemma 3.6. *For an anchor chain under strict precedence, the edit cost of the alignment produced by the greedy algorithm is equal to the chaining cost.*

Proof. This follows from induction on the number of anchors processed, using the same arguments used in the proof of Lemma 3.3. Only now we can modify Cases 1 and 2 from the proof to only consider when $I'.b = I.b$. This makes it so the edit cost of adding the vertices covered by I is equal to $connect(I', I)$ in all cases. □

Lemma 3.7. *Optimal chaining cost under strict precedence $\leq$ anchored edit distance, and optimal chaining cost under weak precedence $\leq$ anchored edit distance.*

Proof. We start with an optimal alignment $\mathcal{M}^*$. Using Lemma 3.4, by applying Algorithms (i) and (ii) to an arbitrary supporting set of anchors for $\mathcal{M}^*$, we obtain a subset of anchors totally ordered under strict precedence and supporting an alignment $\mathcal{M}$ where $EDIT(\mathcal{M}) = EDIT(\mathcal{M}^*)$. By Lemma 3.5, the edit cost of $\mathcal{M}$ is greater or equal to the edit cost of the alignment $\mathcal{M}_G$ given by the greedy algorithm on this set of anchors. And by Lemma 3.6, the co-linear chaining cost of this set of anchors is equal to the edit cost of $\mathcal{M}_G$. Combining these, we have that the optimal anchored edit distance $= EDIT(\mathcal{M}^*) \geq EDIT(\mathcal{M}) \geq EDIT(\mathcal{M}_G) =$ the co-linear chaining cost of this set of anchors $\geq$ optimal chaining cost under strict precedence $\geq$ optimal chaining cost under weak precedence. □

Combining Lemmas 3.3 and 3.7 completes the proof of Theorem 3.2.

4 Implementation

Multi-dimensional RmQs can require significant space (exponential factor w.r.t. dimension), making access-time poor due to cache-inefficiency. We can take advantage of two observations to design a more practical algorithm. First, if sequences are highly similar, their edit distance will be relatively small. Hence the anchored edit distance, denoted in this section as OPT, will be relatively small for MUM or MEM anchors. Second, if the sequences are dissimilar, then the number of MUM or MEM anchors, n, will likely be small. These observations allow us to design an alternative algorithm that runs in $O(n \cdot OPT + n \log n)$ average-case time over all possible inputs where $n \leq \max(|S_1|, |S_2|)$, i.e., the number of anchors is less than the longer sequence length. This property always holds when the anchors are MUMs and is typically true for MEMs as well. This makes the algorithm presented here a practical alternative. As before, let $\mathcal{A}$ be the initial (possibly unsorted) set of anchors, but with $\mathcal{A}_{left} = \mathcal{A}[1]$ and $\mathcal{A}_{right} = \mathcal{A}[n]$. We assume wlog $|S_1| \geq |S_2|$.

The algorithm works by using a guess for the optimal solution, B. The value B is used at every step to bound the range of anchor a values that

Input: n anchors $\mathcal{A}$ and parameters B_1 and B_2.
Output: Array $C[1,n]$ s.t. $C[i]$ is the optimal co-linear chaining cost for any ordered subset of $\mathcal{A}[1,i]$ ending with $\mathcal{A}[i]$.
Let $\mathcal{A}'[1]$, ... $\mathcal{A}'[n]$ be the set of anchors $\mathcal{A}$ sorted on $\mathcal{A}[\cdot].a$;
Initialize array C of size n to 0 and $B \leftarrow B_1$;
do
  $j \leftarrow 1$;
  for $i \leftarrow 1$ **to** n **do**
    while $\mathcal{A}'[i].a - \mathcal{A}'[j].a > B$ **do**
      $j \leftarrow j + 1$;
    end
    $C[i] \leftarrow \min\{C[k] + connect(\mathcal{A}'[k], \mathcal{A}'[i]) \mid j \leq k < i \text{ and } \mathcal{A}'[k] \prec \mathcal{A}'[i]\}$;
  end
  $B_{last} \leftarrow B$;
  $B \leftarrow B_2 \cdot B$;
while $C[n] > B_{last}$;
return $C[1,n]$

Algorithm 2: $O(OPT \cdot n + n \log n)$ average-case algorithm for co-linear anchor chaining.

need to be examined. This bounds the number of anchors that need to be considered (on average). If the value found at $C[n]$ after processing all n anchors is greater than our current guess B, we update our guess to the larger value $B_2 \cdot B$.

Lemma 4.1. *Algorithm 2 runs in $O(n \cdot OPT + n \log n)$ average-case time over all inputs where $n \leq \max(|S_1|, |S_2|)$.*

Proof. The $n \log n$ term is from the sorting the anchors. To analyze the second portion of the algorithm, we first let $X_{h,j}$ be 1 if $\mathcal{A}[h].a$ is placed at index j in S_1. Under the assumption of a random placement of anchors, $\mathbb{E}[X_{h,j}] = 1/|S_1|$. Let X_i be the number of anchors, $\mathcal{A}[h]$, where $\mathcal{A}[h].a \in [\mathcal{A}[i].a - B, \mathcal{A}[i].a - 1]$. We have that $X_i = \sum_{h=1}^{n} \sum_{j=\mathcal{A}[i].a-B}^{\mathcal{A}[i].a-1} X_{h,j}$. Letting X be the total number of anchors processed, $X = \sum_{i=1}^{n} X_i$ and

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{h=1}^{n} \sum_{j=\mathcal{A}[i].a-B}^{\mathcal{A}[i].a-1} \mathbb{E}[X_{i,j}] = \frac{n^2 \cdot B}{|S_1|} \leq nB.$$

Over all iterations, the expected time is a constant factor from $B_1 n(1 + B_2 + \ldots + B_2^{\lceil \log_{B_2} OPT \rceil}) = O(n \cdot OPT)$. □

Extending the above pseudo-code to enable semi-global chaining, i.e., free end gap on queries, is also simple. In each i-loop, the connection to anchor $\mathcal{A}_{left}$ must be always considered, and for last iteration when $i = n$, j must be set to 1. Second, a revised cost function must be used when connecting to either $\mathcal{A}_{left}$ or $\mathcal{A}_{right}$ where a gap penalty is used only for gaps over the query sequence. The experiments done in this work use an implementation of this algorithm for the case of strong-precedence and variable-length anchors.

5 Evaluation

There are multiple open-source libraries/tools that implement edit distance computation. Edlib (v1.2.6) tool by Šošić and Šikić (2017) implements Myers's bit-vector algorithm (Myers (1999)) and Ukkonen's banded algorithm (Ukkonen (1985)), and is known to be the fastest implementation currently. From a scalability perspective, the banded alignment is advantageous only when (i) sequence divergence is low, and (ii) the alignment mode being considered is global, i.e., end-to-end sequence comparison. We will show that optimal co-linear chaining cost computed using the proposed theoretically well-founded cost function provides a favorable trade-off between correlation with edit distance and scalability. In this section, we aim to show the following: (i) The proposed algorithm as well as existing chaining methods achieve significant speedup compared to Edlib, (ii) In contrast to existing chaining methods, our implementation consistently achieves high Pearson correlation (> 0.90) with edit distance while requiring modest time and memory resources, (iii) Sensitivity of our algorithm can be affected by what input anchors are provided, and (iv) Our algorithm can be useful for phylogeny reconstruction.

We implemented the proposed algorithm (Section 4) in `C++`, and refer to it as ChainX. Inputs to ChainX are a target string, one or more query strings, comparison mode (global or semi-global), anchor type preferred, i.e., maximal unique matches (MUMs) or maximal exact matches (MEMs), and a minimum match length. ChainX includes a pre-processing step to index target string using the same suffix array-based algorithm (Vyverman *et al.* (2013)) as used in Nucmer4 (Marçais *et al.* (2018)). For each query-target pair, ChainX outputs optimal chaining cost.

Existing co-linear chaining implementations. Co-linear chaining has been implemented previously as a stand-alone tool in Coconut (Abouelhoda *et al.* (2008)), Clasp (Otto *et al.* (2011)), and also used as a heuristic in commonly used genome-to-genome mapper Nucmer4 (Marçais *et al.* (2018)) and long-read mapper Minimap2 (Li (2018)). Out of these, Clasp (v1.1), Nucmer4 (v4.0.0rc1) and Minimap2 (v2.17-r941) tools are available as open-source, and used here for comparison purpose. Unlike ChainX, these tools execute their respective chaining algorithms using a maximization objective function to enable local pattern matching. Clasp, being a stand-alone chaining method returns chaining scores in its output, whereas we modified Minimap2 and Nucmer4 to print the maximum chaining score for each query-target string pair, and skip subsequent steps. To enable a fair comparison, all methods were run with single thread and same minimum anchor size 20, which is also used as a default in Nucmer4. Accordingly, ChainX, Clasp and Nucmer4 were run with MUMs of length $\geq$ 20, and Minimap2 was allowed to use minimizer k-mers of length 20. For these tests, we made use of an Intel Xeon Processor E5-2698 v3 processor with 32 cores and 128 GB RAM. All tools were required to consider only the forward strand of each query string.

ChainX and Clasp are exact solvers of co-linear chaining problem, but use different gap-cost functions. Clasp supports two cost functions which were referred to as sum-of-pair and linear gap cost functions in their paper (Otto *et al.* (2011)). Unlike ChainX, Clasp only permits non-overlapping anchors in a chain. Clasp solves the co-linear chaining using RmQ data structure, requiring $O(n \log^2 n)$ and $O(n \log n)$ time for the two gap cost functions respectively. Clasp requires a set of anchors as input, therefore, we supplied it the same set of anchors, i.e., MUMs of length ≥ 20 as used by ChainX. We tested Clasp with both of its gap-cost functions, and refer to these two versions as Clasp and Clasp-linear respectively. Minimap2 and Nucmer4 use co-linear chaining as part of their seed-chain-extend pipelines. Both Minimap2 and Nucmer2 permit anchor overlaps in a chain, as well as penalize gaps using their own functions. However, both these tools employ heuristics (e.g., enforce a maximum gap between adjacent chained anchors) for faster processing which can result in sub-optimal chaining results.

Runtime and memory comparison. We downloaded the same set of query and target strings that were used for benchmarking in Edlib paper (Šošić and Šikić (2017)). This test data allowed us to compare tools for end-to-end comparisons as well as semi-global comparisons at various degrees of similarity levels. For testing end-to-end comparisons, the target string had been mutated at various rates using mutatrix (https://github.com/ekg/mutatrix), whereas for the semi-global comparisons, a substring of the target string had been mutated. Table 1 presents runtime and memory comparison of all tools. Columns of the table are organized to show tools in three categories: edit distance (Edlib); optimal co-linear chaining tools (ChainX, Clasp, Clasp-linear); and heuristic implementations (Nucmer4, Minimap2). We make the following observations here. First, chaining methods tend to be significantly faster

Table 1. Runtime and memory usage comparison of edit distance solver Edlib and co-linear chaining methods ChainX, Clasp, Nucmer4 and Minimap2.

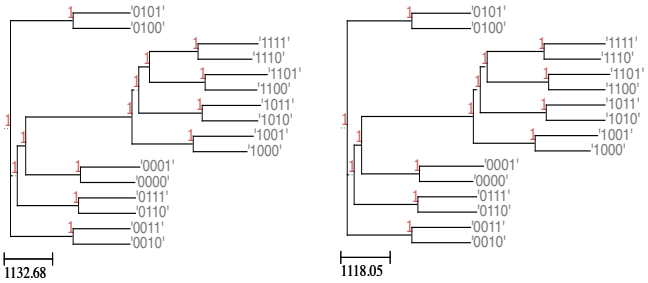
Sequence sizes	Similarity	No. of MUMs	Edlib		ChainX		Clasp		Clasp (linear)		Nucmer4		Minimap2	
			Time*	(Memory**)	Time	(Memory)	Time	(Memory)	Time	(Memory)	Time	(Memory)	Time	(Memory)
Semi-global sequence comparisons														
$10^4 \times 5 * 10^6$	99%	67	190	(17)	2.0	(57)	1.8	(57)	0.9	(57)	1.8	(60)	1.5	(75)
$10^4 \times 5 * 10^6$	97%	160	642	(17)	2.1	(57)	4.8	(57)	1.8	(57)	4.1	(60)	3.0	(75)
$10^4 \times 5 * 10^6$	94%	176	1165	(17)	2.4	(57)	5.9	(57)	2.1	(57)	3.2	(60)	1.4	(75)
$10^4 \times 5 * 10^6$	90%	135	2168	(17)	2.6	(57)	4.7	(57)	2.0	(57)	5.5	(60)	1.4	(75)
$10^4 \times 5 * 10^6$	80%	28	2360	(17)	3.4	(57)	2.5	(57)	2.2	(57)	3.4	(60)	2.4	(75)
$10^4 \times 5 * 10^6$	70%	3	4297	(17)	3.3	(57)	2.2	(57)	2.3	(57)	5.5	(60)	1.8	(75)
Global sequence comparisons														
$10^6 \times 10^6$	99%	7012	949	(8)	46.1	(24)	1236.8	(1800)	182.8	(257)	68.7	(26)	92.1	(35)
$10^6 \times 10^6$	97%	15862	1308	(8)	486.7	(24)	5363.7	(8742)	765.4	(1278)	87.8	(26)	80.5	(36)
$10^6 \times 10^6$	94%	18389	2613	(8)	654.8	(24)	11737.1	(20501)	1021.0	(1694)	113.5	(27)	65.3	(34)
$10^6 \times 10^6$	90%	14472	6233	(8)	851.2	(24)	5110.3	(8277)	115.3	(27)	121.8	(26)	53.4	(33)
$10^6 \times 10^6$	80%	2964	12506	(8)	161.5	(24)	504.8	(572)	133.7	(24)	148.9	(26)	46.2	(32)
$10^6 \times 10^6$	70%	195	29602	(8)	138.0	(23)	140.6	(23)	139.6	(23)	167.3	(26)	46.7	(32)

*Runtime is measured in milliseconds across the columns. **Memory usage is always noted in MBs.

than Edlib in most cases, and we see up to three order of magnitude speedup. Second, within optimal chaining methods, Clasp's time and memory consumption increases quickly with increase in count of anchors, which is likely due to irregular memory access and storage overhead of using a 2d-RmQ data structure. Finally, we note that Minimap2 and Nucmer4 are often faster than ChainX during global string comparisons due to their fast heuristics. All tools (except Edlib) use an indexing step such as building a k-mer hash table (Minimap2) or computing suffix array (ChainX, Clasp, Nucmer4). Indexing time was excluded from reported results, and was found to be comparable. For instance, in the case of semi-global comparisons, ChainX, Nucmer4, Minimap2 required 590 ms, 736 ms, 236 ms for index computation respectively.

Correlation with edit distance. Subsequently we checked how well the chaining cost (or score) correlates with edit distance. ChainX computes co-linear chaining cost (minimization problem), whereas remaining tools compute chaining score (maximization problem). As a result, a positive correlation with edit distance is expected in case of ChainX, and negative correlation for others. For the purpose of comparison, we use absolute value of Pearson correlation coefficient. In this experiment, we simulated 100 query strings within each similarity range: $90-100\%, 80-90\%, 75-80\%$. Table 2 shows the correlation achieved by all the tools. Here we make the following observations. First, for closely-related inputs (i.e., $90-100\%$ similarity), all tools achieve strong correlation close to one, although ChainX, Clasp and Minimap2 are superior to Clasp-linear and Nucmer4. For distantly-related inputs (i.e., $75-80\%$ similarity), ChainX and Clasp achieve superior accuracy than Clasp-linear, Nucmer4 and Minimap2. As ChainX requires much less resources in terms of runtime and memory compared to Clasp, it can be used as an alternative to edit distance. An optimal co-linear chain can also be used as a coarse-grained pairwise sequence alignment, which can be further extended to a fine-grained base-to-base alignment (not-necessarily optimal) using efficient 'seed-extension' heuristics.

Effect of anchor type and minimum length. The choice of anchor-finding strategy naturally affects the performance and accuracy of ChainX. Suppose anchors are chosen as all maximal exact matches of length $\geq l_{min}$ between a pair of strings, each being s-long. Consider one extreme case where l_{min} is set to s. In this case, ChainX can return only two values: 0 if the input strings are equal, and s otherwise. Another extreme case is when l_{min} is set to one. Then, ChainX's output is guaranteed to match edit distance between the input strings, however, the excessive count of anchors will make the chaining problem computationally prohibitive. We tested runtime and accuracy of ChainX while varying the anchor type (MUMs/MEMs) and l_{min} parameter (Table 3). When MUMs are used as anchors, then ChainX maintains good scalability, and lowering l_{min} from 20 to 10 improves accuracy, but the accuracy saturates afterwards. This is because very short matches will unlikely be unique and won't be selected as MUMs. However, when MEMs are used as anchors, accuracy continues to improve with decreasing minimum length parameter, however, runtime grows exponentially using $l_{min} = 7$. By default, ChainX uses MUMs of length ≥ 20, and requires $O(OPT \cdot n + n \log n)$ average-case time as proved in Section 4.



(a) Tree computed using Edlib (b) Tree computed using ChainX

Fig. 12. Phylogenetic trees obtained using simulated genomes.

Application to phylogeny reconstruction. Although ChainX is not explicity designed for building phylogenetic trees, ChainX's strong correlation with edit distance makes it useful to get a good estimation of mutation rate. In this experiment, we created a family of Escherichia coli genomes using the same simulation methodology as used by Marçais *et al.* (2019). The simulation framework starts with *E.coli* K-12 MG1655 (NC_000913.2) genome, and recursively creates two children by adding an insertion sequence (IS) at two random locations. Using four IS elements (IS1, IS5, IS2, and IS186), their framework created 16 genomes, and the goal is to recover the history of these insertion events. In this context, similarity metrics such as Jaccard similarity which treat genomes as 'bag of words' won't be applicable because of near-identical k-mer content, but co-linear chaining techniques are well-suited to handle this. ChainX supports an `-all2all` mode to execute all-to-all global comparisons among query strings and report chaining costs as a phylip-formatted distance matrix. Figure 12 shows that trees constructed by using Edlib and ChainX exhibit identical structure. During simulation, each genome was named using a binary number, s.t. common prefix indicates shared lineage, and siblings

Table 2. Absolute Pearson correlation coefficients of chaining costs (or scores) computed by various methods with the corresponding edit distances.

Seq. sizes [count of comparisons]	Similarity	Correlation coefficient				
		ChainX	Clasp	Clasp (linear)	Nucmer4	Minimap2
		Semi-global sequence comparisons				
$10^4 \times 5 * 10^6$ [100]	90%-100%	0.996	0.994	0.986	0.968	0.995
$10^4 \times 5 * 10^6$ [100]	80%-90%	0.975	0.976	0.786	0.864	0.958
$10^4 \times 5 * 10^6$ [100]	75%-80%	0.927	0.915	0.732	0.733	0.808
		Global sequence comparisons				
$10^6 \times 10^6$ [100]	90%-100%	0.999	0.997	0.994	0.991	0.999
$10^6 \times 10^6$ [100]	80%-90%	0.998	0.998	0.922	0.955	0.996
$10^6 \times 10^6$ [100]	75%-80%	0.992	0.993	0.871	0.907	0.860

Table 3. Effect of anchor computation method on the performance of ChainX.

Seq. sizes [count]	Similarity	Using MUMs								Using MEMs					
		$len \geq 20$		$len \geq 10$		$len \geq 7$		$len \geq 1$		$len \geq 20$		$len \geq 10$		$len \geq 7$	
		Time*	(coeff.**)	Time	(coeff.)	Time	(coeff.)	Time	(coeff.)	Time	(coeff.)	Time	(coeff.)	Time	(coeff.)
$10^4 \times 5 * 10^6$ [100]	90%-100%	7.2	(0.996)	2.9	(0.997)	3.5	(0.997)	11.5	(0.997)	5.1	(0.996)	8.1	(0.997)	2652	(0.998)
$10^4 \times 5 * 10^6$ [100]	80%-90%	4.5	(0.975)	5.6	(0.992)	3.2	(0.992)	4.8	(0.992)	4.5	(0.975)	7.4	(0.993)	5413	(0.995)
$10^4 \times 5 * 10^6$ [100]	75%-80%	5.3	(0.927)	5.9	(0.977)	1.9	(0.977)	3.8	(0.977)	5.0	(0.927)	10.9	(0.987)	9221	(0.992)

*Runtime is measured in seconds across the columns. **Pearson correlation coefficient.

in the last generation have equal three-digit prefix. As can be seen, the tree computed using either of the two tools resolved most of the lineages.

6 Conclusions

This work presented new algorithmic insights for co-linear chaining, a routinely used method within sequence mapping tools. We addressed the general case of this problem which allows anchor overlaps and penalizes gap cost between adjacent chained anchors. Since many commonly used mappers (e.g., Nucmer4, Minimap2) also require chaining with overlap and gap-costs, designing faster, rigorous algorithms and superior cost functions is important. We presented the first subquadratic time algorithms for multiple versions of this problem, e.g., using fixed-length (k-mers) or variable-length anchors (maximal matches), and using weak or strong precedence ordering criteria. We also provided a new cost function for the co-linear chaining problem, which made it possible to mathematically link co-linear chaining and the edit distance problem. This result is a useful addition to a prior result by Mäkinen and Sahlin (2020) where a connection between the co-linear chaining problem and the longest common subsequence problem was established. Scalability and accuracy of ChainX was demonstrated using both global and semi-global sequence comparison modes. ChainX code as well as datasets used for benchmarking are available at https://github.com/at-cg/ChainX.

References

Abouelhoda, M. and Ohlebusch, E. (2005). Chaining algorithms for multiple genome comparison. *Journal of Discrete Algorithms*, **3**(2-4), 321–341.

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2008). Coconut: an efficient system for the comparison and analysis of genomes. *BMC bioinformatics*, **9**(1), 476.

Backurs, A. and Indyk, P. (2015). Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 51–58.

Chaisson, M. J. and Tesler, G. (2012). Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC bioinformatics*, **13**(1), 238.

de Berg, M., Cheong, O., van Kreveld, M. J., and Overmars, M. H. (2008). *Computational geometry: algorithms and applications, 3rd Edition*. Springer.

Delcher, A. L., Kasif, S., *et al.* (1999). Alignment of whole genomes. *Nucleic acids research*, **27**(11), 2369–2376.

Eppstein, D., Galil, Z., Giancarlo, R., and Italiano, G. F. (1992a). Sparse dynamic programming i: linear cost functions. *Journal of the ACM (JACM)*, **39**(3), 519–545.

Eppstein, D., Galil, Z., *et al.* (1992b). Sparse dynamic programming ii: convex and concave cost functions. *Journal of the ACM (JACM)*, **39**(3), 546–567.

Jain, C., Rhie, A., Hansen, N., Koren, S., and Phillippy, A. M. (2020). A long read mapping method for highly repetitive reference sequences. *bioRxiv*.

Kurtz, S. *et al.* (2004). Versatile and open software for comparing large genomes. *Genome biology*, **5**(2), R12.

Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, **34**(18), 3094–3100.

Li, H., Feng, X., and Chu, C. (2020). The design and construction of reference pangenome graphs with minigraph. *Genome Biology*, **21**(1), 265.

Mäkinen, V. and Sahlin, K. (2020). Chaining with overlaps revisited. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Marçais, G., Delcher, A. L., *et al.* (2018). Mummer4: A fast and versatile genome alignment system. *PLoS computational biology*, **14**(1), e1005944.

Marçais, G., DeBlasio, D., Pandey, P., and Kingsford, C. (2019). Locality-sensitive hashing for the edit distance. *Bioinformatics*, **35**(14), i127–i135.

Morgenstern, B. (2002). A simple and space-efficient fragment-chaining algorithm for alignment of DNA and protein sequences. *Applied Mathematics Letters*, **15**(1), 11–16.

Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, **46**(3), 395–415.

Myers, G. and Miller, W. (1995). Chaining multiple-alignment fragments in sub-quadratic time. In *SODA*, volume 95, pages 38–47.

Otto, C., Hoffmann, S., Gorodkin, J., and Stadler, P. F. (2011). Fast local fragment chaining using sum-of-pair gap costs. *Algorithms for Molecular Biology*, **6**(1), 4.

Ren, J. and Chaisson, M. (2020). lra: the long read aligner for sequences and contigs. *bioRxiv*.

Sahlin, K. and Makinen, V. (2020). Accurate spliced alignment of long RNA sequencing reads. *bioRxiv*.

Sedlazeck, F. J. *et al.* (2018). Accurate detection of complex structural variations using single-molecule sequencing. *Nature methods*, **15**(6), 461–468.

Shibuya, T. and Kurochkin, I. (2003). Match chaining algorithms for cDNA mapping. In *Algorithms in Bioinformatics, Third International Workshop, WABI 2003, Budapest, Hungary, September 15-20, 2003, Proceedings*, pages 462–475.

Šošić, M. and Šikić, M. (2017). Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, **33**(9), 1394–1395.

Ukkonen, E. (1985). Algorithms for approximate string matching. *Information and control*, **64**(1-3), 100–118.

Uricaru, R. *et al.* (2011). Novel definition and algorithm for chaining fragments with proportional overlaps. *Journal of Computational Biology*, **18**(9), 1141–1154.

Vyverman, M., De Baets, B., *et al.* (2013). essamem: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, **29**(6), 802–804.

Wilbur, W. J. and Lipman, D. J. (1983). Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences*, **80**(3), 726–730.