Co-linear Chaining with Overlaps and Gap Costs

Chirag Jain[1], Daniel Gibney[2], Sharma V. Thankachan[3]

[1] Dept. of Computational and Data Sciences, Indian Institute of Science, India
[2] School of Comp. Science and Engineering, Georgia Institute of Technology, USA
[3] Dept. of Computer Science, University of Central Florida, USA
Email: chirag@iisc.ac.in, {daniel.j.gibney,sharma.thankachan}@gmail.com

Abstract. Co-linear chaining has proven to be a powerful heuristic for finding near-optimal alignments of long DNA sequences (e.g., long reads or a genome assembly) to a reference. It is used as an intermediate step in several alignment tools that employ a seed-chain-extend strategy. Despite this popularity, efficient subquadratic-time algorithms for the general case where chains support anchor overlaps and gap costs are not currently known. 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 also 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 orders of magnitude faster than edit distance, and achieves correlation coefficient above 0.9 with edit distance for closely as well as distantly related sequences.

Keywords: Edit distance · Alignment · Co-linear Chaining.

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 [3,10]. This lower-bound indicates a challenge for scaling the computation of edit distance to high-throughput sequencing data. Instead, heuristics are often used to obtain an approximate solution in less time and space. One such popular heuristic is co-linear 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 long-read mappers [6,11,14,15,23,24,26] and generic sequence aligners [2,5,13,18,22]. 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.

Several algorithms have been developed for the co-linear chaining [1,16,27,30] and even more in the context of sparse dynamic programming [8,9,19,21,32,17].

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 $O(n \log n \log \log n)$ time for a simpler variant of the problem where anchors used in a solution must be non-overlapping [1]. More recently, Mäkinen and Sahlin gave an algorithm running in $O(n \log n)$ time where anchor overlaps are allowed, but gaps between anchors are not considered in the cost-function [16]. 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 a more realistic model for anchor chaining. For example, consider a simple scenario where minimizers [25] are used to identify anchors. Suppose query and reference sequences are identical, then adjacent minimizer-anchors will likely overlap. Not allowing anchor overlaps during chaining will lead to a penalty cost associated with gaps between chained anchors despite the two strings being identical. Therefore, 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 penalizing long gaps between the anchors is unlikely to produce correct alignments. This is why both anchor overlaps and gap costs are supported during chaining in widely-used aligners, e.g., Minimap2 [12,14], Nucmer4 [18]. 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 where OPT is the optimal solution value. 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 the optimal chaining cost is equal to the anchored edit distance.
- We empirically demonstrate that computing optimal chaining cost is 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 correlation coefficients are favorable when compared to suboptimal chaining methods implemented in Minimap2 and Nucmer4.

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

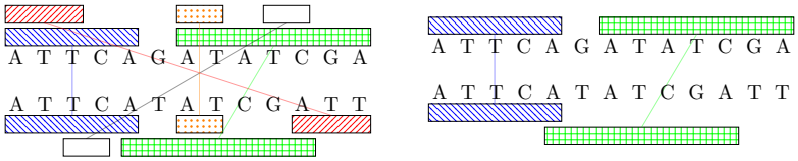

Fig. 1: (Left) Anchors representing a set of exact matches are shown as rectangles. The co-linear chaining problem is to find an optimal ordered subset of anchors subject to some cost function. (Right) A chain of overlapping anchors.

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 an exact 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 $I.d$. Here $b-a=d-c$ and $S_1[a+j]=S_2[c+j]$ for all $0 \leq j \leq b-a$. We say that the character match $S_1[a+j]=S_2[c+j]$, $0 \leq j \leq b-a$, 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 [7] 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 algorithms will make use of dynamic 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 known techniques in computational geometry, a data structure can be built in $O(n \log^{d-1} n)$ time and space, that can both answer queries and modify a point's weight in $O(\log^d n)$ time [4].

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 specify whether we mean a chain under strict precedence or under weak precedence when 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 a chain. The chaining problem asks for a chain of $m \leq n$ anchors, $\mathcal{A}'[1], \mathcal{A}'[2], \ldots, \mathcal{A}'[m]$, 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 4, 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 $g_2 = \max(0, I.c - I'.d - 1)$. Define the gap cost $g(I', I) = \max(g_1, g_2)$.
- The overlap o_1 is defined such that $I'.b - o_1$ reflects the non-overlapping prefix 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: $C[i] = \min_{\mathcal{A}'[k] \prec \mathcal{A}'[i]} \big(C[k] + connect(\mathcal{A}'[k], \mathcal{A}'[i])\big)$ where the base case associated with anchor $\mathcal{A}_{left}$ is $C[1] = 0$. The optimal chaining cost will be stored in $C[n]$ after spending $O(n^2)$ time. We will provide an $O(n \log^4 n)$ time algorithm for this problem.

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 edit cost 1.

The above problem is solvable in $O(|S_1||S_2|)$ time and space. We can assume that input does not contain redundant anchors, therefore, the count of anchors is $\leq |S_1||S_2|$. 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 of Alignment

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 character match between the ith character in S_1 and the jth character 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 a character match not supported by an anchor. All unmatched vertices are labeled with an 'x' to indicate that the corresponding character is deleted. An important observation is that no two edges cross.

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 characters 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 a particular alignment. We also associate an edit cost with the alignment, denoted as $EDIT(\mathcal{M})$. This is equal to the number of vertices marked with x in $\mathcal{M}$ plus the number of dashed edges in $\mathcal{M}$.

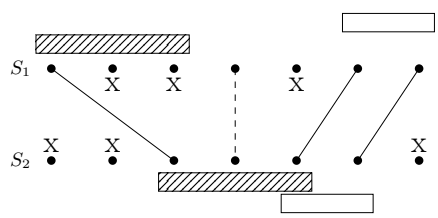

Fig. 2: The graph representation of an alignment. Solid edges represent anchor-supported character matches, dashed edges represent substitutions and unsupported matches, and x's represent deletions. We use $\mathcal{M}$ to denote an alignment. Here $EDIT(\mathcal{M}) = 7$, the total number of x's and dashed edges.

3 Our Algorithms

Theorem 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.*

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 achieve this 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.

Algorithm for chains under strict precedence. 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 ∞ except for $I = \mathcal{A}_{left}$, where the corresponding points are given weight 0. 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 $C[I'] = C[j]$.

The query for each RmQ structure is determined by the different inequalities relating $I.a$, $I.b$, $I.c$, and $I.d$ to previous anchors in the case considered. For example, in Case 1.a (Figure 3), it can be seen that $I'.b < I.a$ and $I.a - I'.b < I.c - I'.d$, making $I'.b \in [0, I.a - 1]$ and $I'.d - I'.b \in [-\infty, I.c - I.a]$, motivating the query input $[0, I.a - 1] \times [-\infty, I.c - I.a]$. At the same time, the values stored in these RmQ structures are determined by the expression for the co-linear chaining cost in that case, $C[I'] + I.c - I'.d - 1$. Note that the values stored in each RmQ structure depend only on previously processed anchors and are combined with the values $I.a$, $I.b$, $I.c$, and $I.d$ for the current anchor I being processed to obtain the appropriate cost. Hence, for $\mathcal{T}_{1a}$ we store values of the form $C[I'] - I'.d$ and combine this with $I.c$ to obtain the cost. The other cases can be similarly analyzed.

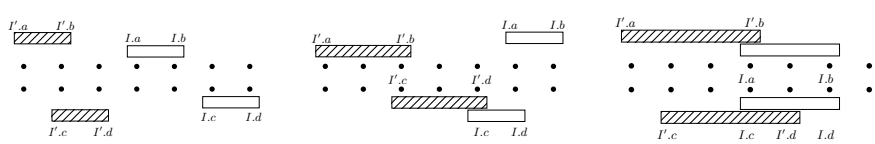

Fig. 3: (Left) Case 1.a. Colinear chaining cost is $C[I'] + g_2 = C[I'] + I.c - I'.d - 1$. (Middle) Case 2.a. Chaining cost is $C[I'] + g_1 + o_2 = C[I'] + I.a - I'.b + I'.d - I.c$. (Right) Case 3.a. Chaining cost is $C[I'] + o_2 - o_1 = C[j] + I'.d - I.c - (I'.b - I.a)$.

1. Case: I' disjoint from I.
   (a) Case: The gap in S_1 is less or equal to gap in S_2 (Fig. 3 (Left)). The range minimum query (query input) is of the form: $[0, I.a - 1] \times [-\infty, I.c - I.a]$. Let the query response (weight) from $\mathcal{T}_{1a}$ be $v_{1a} = \min\{C[I'] - I'.d : (I'.b, I'.d - I'.b) \in [0, I.a - 1] \times [-\infty, I.c - I.a]\}$ and let $C_{1a} = v_{1a} + I.c - 1$.

(b) Case: The gap in S_2 is less than gap in S_1. The range minimum query is of the form $[0, I.c-1] \times [I.c-I.a+1, \infty]$. Let the query response from $\mathcal{T}_{1b}$ be $v_{1b} = \min\{C[I'] - I'.b : (I'.d, I'.d - I'.b) \in [0, I.c-1] \times [I.c-I.a+1, \infty]\}$ and let $C_{1b} = v_{1b} + I.a - 1$.

2. Case: I' and I overlap in only one dimension.
   (a) Case: I' and I overlap only in S_2 (Fig 3 (Middle)). The range minimum query is of the form $[0, I.a-1] \times [0, I.c] \times [I.c, I.d]$. Let the query response from $\mathcal{T}_{2a}$ be $v_{2a} = \min\{C[I'] - I'.b + I'.d : (I'.b, I'.c, I'.d) \in [0, I.a-1] \times [0, I.c] \times [I.c, I.d]\}$ and let $C_{2a} = v_{2a} + I.a - I.c$.
   (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 range minimum query is of the form $[I.a, I.b] \times [0, I.c-1]$. Let the query response from $\mathcal{T}_{2b}$ be $v_{2b} = \min\{C[I'] + I'.b - I'.d : (I'.b, I'.d) \in [I.a, I.b] \times [0, I.c-1]\}$ and let $C_{2b} = v_{2b} + I.c - I.a$.
3. Case: I' and I overlap in both dimensions.
   (a) Case: Greater overlap in S_2 (Fig. 3 (Right)). Here, $|o_1 - o_2| = o_2 - o_1 = I'.d - I.c - (I'.b - I.a)$. The range minimum query is of the form $[I.a, I.b] \times [0, I.c] \times [I.c, I.d] \times [I.c-I.a+1, \infty]$. Let the query response from $\mathcal{T}_{3a}$ be $v_{3a} = \min\{C[I'] - I'.b + I'.d : (I'.b, I'c, I'.d, I'.d - I'.b) \in [I.a, I.b] \times [0, I.c] \times [I.c, I.d] \times [I.c - I.a + 1, \infty]\}$ and let $C_{3a} = v_{3a} + I.a - I.c$.
   (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, the range minimum query is of the form $[I.a, I.b] \times [I.c, I.d] \times [-\infty, I.c - I.a]$. Let the query response from $\mathcal{T}_{3b}$ be $v_{3b} = \min\{C[I'] + I'.b - I'.d : (I'.b, I'.d, I'.d - I'.b) \in [I.a, I.b] \times [I.c, I.d] \times [-\infty, I.c - I.a]\}$ and let $C_{3b} = v_{3b} - I.a + I.c$.

Finally, let $C[i] = \min(C_{1a}, C_{1b}, C_{2a}, C_{2b}, C_{3a}, C_{3b})$ and update the RmQ structures as shown in the Pseudo-code in Appendix 1. In the pseudo-code, every 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 updates the point to have the new weight. The four-dimensional RmQ structures for Case 3.a require $O(\log^4 n)$ time per query and update, causing an over time complexity that is $O(n \log^4 n)$. In Appendix 2 we present the modifications for weak precendence and fixed-length anchors.

4 Proof of Equivalence

Theorem 2. *For a fixed set of anchors $\mathcal{A}$, the following quantities are equal: the anchored edit distance, the optimal co-linear chaining cost under strict precedence, and the optimal co-linear chaining cost under weak precedence.*

The optimal co-linear chaining cost is defined using the cost function described in Section 2.1. An implication of Theorems 1 and 2 is that if only the anchored edit distance is desired (and not an optimal strictly ordered anchor chain), there exists a $O(n \log^3 n)$ for computing this value.

Theorem 2 will follow from Lemmas 1 and 2.

Lemma 1. *Anchored edit distance ≤ optimal co-linear chaining cost under weak precedence ≤ optimal co-linear chaining cost under strict precedence.*

Proof. The second inequality follows from the observation that every set of anchors ordered under strict precedence is also ordered under weak precedence. We now focus on the inequality between anchored edit distance and co-linear chaining cost under weak precedence. Starting with an anchor chain under weak precedence, $\mathcal{A}[1]$, $\mathcal{A}[2]$, ... with associated co-linear chaining cost x, we provide an alignment with an anchored edit distance that is at most x. This 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. We now present the details.

Greedy Algorithm. 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 4-6). 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 4): To continue the alignment, delete the substring $S_2[I'.d{+}1, I.d]$ from S_2. This has edit cost $I.d{-}I'.d$. We can assume both intervals of I' are not nested in intervals of I, hence $connect(I', I) = o_1 - o_2 = I'.b - I.a - I'.d + I.c \geq I.c + I.b - I.a - I'.d = I.d - I'.d$.
2. **Case** $I'.b \geq I.b$ and $I'.d < I.c$ (Fig 4): Delete the substring $S_2[I'.d+1, I.d]$ from S_2, with edit cost $I.d - I'.d$. Also $connect(I', I) = o_1 + g_2 = I'.b - I.a + I.c - I'.d \geq I.c + I.b - I.a - I'.d = I.d - I'.d$.

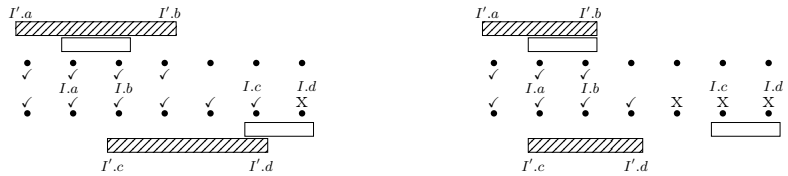

Fig. 4: Cases in Proof of Lemma 1. The ✓symbol indicates symbols processed prior to considering I. (Left) Case $I', b \geq I.b$ and $I'.d \geq I.c$ (Right) $I'.b \geq I.b$ and $I'.d < I.c$.

3. **Case** $I.b > I'.b$, $I.a \leq I'.b$, $I.c \leq I'.d$ (Fig. 5): Supposing wlog that $o_1 > o_2$, delete $S_2[I'.d+1, I'.d+o_1-o_2]$, and match $S_1[I'.b+1, I.b]$ and $S_2[I'.d+o_1-o_2+1, I.d]$. This has edit cost $o_1 - o_2$ and $connect(I', I) = o_1 - o_2$.
4. **Case** $I.b > I'.b$, $I.a \leq I'.b$, $I.c > I'.d$ (Fig. 5): We delete $S_2[I'.d+1, I'.d+o_1+g_2]$ and match $S_1[I'.b+1, I.b]$ with $S_2[I'.d+o_1+g_2+1, I.d]$. This has edit cost $o_1 + g_2$ and $connect(I', I) = o_1 + g_2$.
5. **Case** $I.a > I'.b$, $I.c > I'.d$ (Fig. 6): Supposing wlog $g_2 \geq g_1$, 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. □

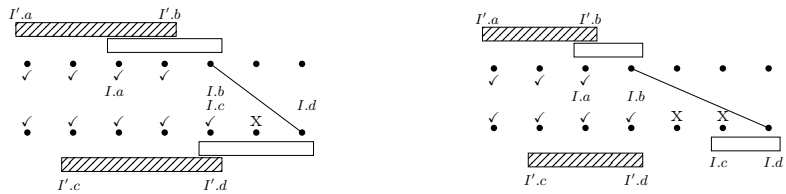

Fig. 5: Cases in Proof of Lemma 1. (Left) Case $I.b > I'.b$, $I.a \leq I'.b$, $I.c \leq I'.d$. (Right) Case $I.b > I'.b$, $I.a \leq I'.b$, $I.c > I'.d$.

We delay the details of Lemma 2's proof to Section 4.1.

Lemma 2. *For a set of anchors $\mathcal{A}$, optimal chaining cost under strict precedence $\leq$ anchored edit distance.*

Proof. We start with an arbitrary alignment $\mathcal{M}$ supported by $\mathcal{A}$. We will show in Lemma 3 how to obtain a subset $\mathcal{B} \subseteq \mathcal{A}$ totally ordered under strict precedence and supporting an alignment $\mathcal{M}'$ where $EDIT(\mathcal{M}') \leq EDIT(\mathcal{M})$. We will then show in Lemma 4 that 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 $\mathcal{B}$. Finally, in Lemma 5 we show that the co-linear chaining cost of $\mathcal{B}$ is equal to the edit cost of $\mathcal{M}_G$. Combining, we have $EDIT(\mathcal{M}) \geq EDIT(\mathcal{M}') \geq EDIT(\mathcal{M}_G) =$ the co-linear chaining cost on $\mathcal{B} \geq$ optimal co-linear chaining cost under strict precedence for $\mathcal{A}$. The result follows from the fact that $EDIT(\mathcal{M})$ equals the anchored edit distance when $\mathcal{M}$ is an optimal alignment for $\mathcal{A}$. □

4.1 Details of Lemma 2 Proof

We apply Algorithm (i) followed by Algorithm (ii) to convert a supporting set of anchors $\mathcal{A}$ for $\mathcal{M}$ into the totally ordered subset of anchors $\mathcal{B}$ supporting $\mathcal{M}'$. Note that these algorithms are only for the purpose of the proof. Moving forward, we call an edge $e = (S_1[h], S_2[k])$ contained but not supported by I if $h \in [I.a, I.b]$ or $k \in [I.c, I.d]$ and $h - I.a \neq k - I.c$. We define for e the two edges $e' = (S_1[h], S_2[I.c + h - I.a])$ and $e'' = (S_1[I.a + k - I.c], S_2[k])$, which are supported by I.

Algorithm (i). Algorithm for removing incomparable anchors. Let I and I' be two incomparable anchors under weak precedence (Fig 6). 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, starting with that rightmost edge, for any edge e that is contained but not supported by I, we replace e with the

rightmost of e' and e''. 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. 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 wlog I' has an interval nested in one of the intervals belonging to I. Let e_R be the rightmost edge supported by I. Working from right-to-left, we replace any edge e to the left of e_R that is contained but not supported by I with the rightmost of e' and e''. Next, working from left-to-right, we replace any edge e to the right of e_R that is contained but not supported by I with the leftmost of e' and e''. These procedures combined will replace all edges supported by I' with those supported by I. We repeat this until there are no two nested intervals amongst all remaining anchors. Finally, remove all anchors that do not support any edge. We call such an anchor chain where every anchor supports at least one edge *minimal*.

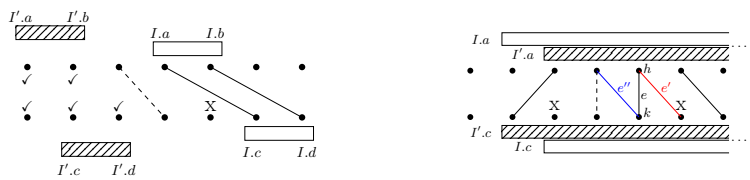

Fig. 6: (Left) Case $I.a > I'.b$, $I.c > I'.d$. (Right) Anchors I and I' are incomparable. The current alignment is shown with black solid and dashed edges. To remove I' we sweep from right-to-left, replacing edges not supported by I with edges supported by I. Here, $e = (S_1[h], S_2[k])$ is not supported by I and will be replaced with $e' = (S_1[h], S_2[I.c + h - I.a])$ (in red), which is supported by I.

Lemma 3. $EDIT(\mathcal{M}') \leq EDIT(\mathcal{M})$.

Proof. For Algorithm (i), suppose we are replacing an edge e not supported by the anchor I, the anchor we wish to keep. Suppose wlog that e' is the rightmost of e' and e'', so we replace e with e'. Because the edge immediately to the right of e is also aligned with I, deleting $S_2[k]$ and matching $S_2[I.c + h - I.a]$, does not require modifying any additional edges. If e was a solid edge the edit cost is unaltered, since the total number of deletions and matches is unaltered. If e was a dashed edge, replacing e with e' converts a substitution or unsupported exact match at $S_2[k]$ to a deletion, and removes a deletion at $S_2[I.c + h - I.a]$, decreasing the edit cost by 1. The same arguments hold for Algorithm (ii) when we replace edges from right-to-left. In Algorithm (ii) when we process edges from left-to-right, since any edges to left of the edge e being replaced are supported by I, replacing e with the leftmost of e' and e'' does not require modifying any additional edges. Again, if e is solid, the edit cost is unaltered, and if e is dashed, the edit cost is decreased by 1. □

Lemma 4. *The greedy algorithm described in the proof of Lemma 1 produces an optimal alignment for a 'minimal' anchor chain under strict precedence.*

Proof. Proof is similar to proof of Lemma 3 and is deferred to Appendix 3. □

Lemma 5. *For an anchor chain under strict precedence, the edit cost of the alignment produced by the greedy algorithm described in the proof of Lemma 1 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 1. However, only $I'.b = I.b$ needs to be considered in Cases 1 and 2 leading to equality in these cases. □

5 Implementation

In multi-dimensional RmQs, $O(n\log^{d-1} n)$ storage requirement and irregular memory access during a query can limit their efficacy in practice [4]. 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 (Algorithm 1) that requires $O(n)$ worst-case space and $O(n \cdot OPT + n \log n)$ 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 (proof is deferred to Appendix 3) . 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 $|S_1| \geq |S_2|$. We begin by sorting anchor set $\mathcal{A}$ by the component $\mathcal{A}[\cdot].a$ and making a guess for the optimal solution, B (Algorithm 1). The value B is used at every step to bound the range of $\mathcal{A}[\cdot].a$ values that need to be examined. This bounds the number of anchors that need to be considered (on average). If $C[n]$ is greater than our current guess B after processing all n anchors, we update our guess to $B_2 \cdot B$.

Extending the above pseudo-code to enable semi-global chaining, i.e., free anchor gap on both ends of reference sequences, 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 anchor gap over the query sequence. The experiments in the next section use an implementation of this algorithm.

6 Evaluation

There are multiple open-source libraries/tools that implement edit distance computation. Edlib (v1.2.7) [28] uses Myers's bit-vector algorithm [20] and Ukkonen's banded algorithm [29], and is known to be the fastest implementation currently.

Input: n anchors $\mathcal{A}$ and parameters B_1 and B_2.
Output: $C[1,n]$ s.t. $C[i]$ is optimal co-linear chaining cost for any ordered subset of $\mathcal{A}[1,i]$ ending at $\mathcal{A}[i]$.

```
Let A'[1], ... A'[n] be the set of anchors A sorted on A[·].a;
Initialize array C of size n to 0 and B ← B_1;
do
    j ← 1;
    for i ← 1 to n do
        while A'[i].a − A'[j].a > B do
            j ← j + 1;
        end
        C[i] ← min{C[k] + connect(A'[k], A'[i]) | j ≤ k < i and A'[k] ≺ A'[i]};
    end
    B_last ← B ;
    B ← B_2 · B;
while C[n] > B_last;
return C[1, n]
```

Algorithm 1: $O(OPT \cdot n + n \log n)$ average-case algorithm.

In this section, we aim to show that: (i) the proposed algorithm as well as existing chaining methods achieve significant speedup compared to computing exact edit distance using Edlib, and (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.

We implemented Algorithm 1 in `C++`, and refer to it as ChainX. The code is available at https://github.com/at-cg/ChainX. Inputs are a target string, 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. We include a pre-processing step to index target string using the same suffix array-based algorithm [31] used in Nucmer4 [18]. Chaining costs computed using ChainX for each query-target pair are provably-optimal.

Existing co-linear chaining implementations. Co-linear chaining has been implemented previously as a stand-alone utility [2,22] and also used as a heuristic inside widely used sequence aligners [5,14,18]. Out of these, Clasp (v1.1), Nucmer4 (v4.0.0rc1) and Minimap2 (v2.22-r1101) tools are available as open-source, and used here for comparison purpose. Unlike our algorithm where the optimization problem involves minimizing a cost function, these tools execute their respective chaining algorithms using a score maximization objective function. 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. Accordingly, ChainX, Clasp and Nucmer4 were run with MUMs of length ≥ 20, and Minimap2 was configured 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 match only the forward strand of each query string. ChainX and Clasp are both exact solvers of co-linear chaining problem, but use different gap-cost functions. Clasp only permits non-overlapping anchors in a chain, and supports two cost functions which were referred to as *sum-of-pair* and *linear* gap cost functions in their paper [22]. We tested Clasp with both of its gap-cost functions, and refer to these two versions as Clasp-sop and Clasp-linear respectively. Both the versions solve co-linear chaining using RmQ data structures, requiring $O(n \log^2 n)$ and $O(n \log n)$ time respectively. Both require a set of anchors as input, therefore, we supplied the same set of anchors, i.e., MUMs of length ≥ 20 as used by ChainX. Minimap2 and Nucmer4 use co-linear chaining as part of their seed-chain-extend pipelines. Both Minimap2 and Nucmer2 support anchor overlaps in a chain, as well as penalize gaps using custom functions. However, both these tools employ heuristics (e.g., enforce a maximum gap between adjacent chained anchors) for faster processing which can result in suboptimal chaining scores.

Runtime and memory comparison. We downloaded the same set of query and target strings that were used for benchmarking in Edlib paper [28][2]. These test strings, ranging from 10 kbp to 5000 kbp in length, allowed us to compare tools for end-to-end global sequence comparisons as well as semi-global comparisons at various degrees of similarity levels. To test end-to-end comparisons, the target string had been artificially 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 sampled and 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 solver (Edlib); optimal co-linear chaining solvers (ChainX, Clasp-sop, Clasp-linear); and heuristic implementations (Nucmer4, Minimap2). We make the following observations here. First, chaining methods (both optimal and heuristic-based) are significantly faster than Edlib in most cases, and we see up to three orders of magnitude speedup. Second, within optimal chaining methods, Clasp-sop'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 its algorithm that uses a 2d-RmQ data structure. Finally, we note that Minimap2 and Nucmer4 are often faster than exact algorithms 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-sop, Clasp-linear, 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. We checked how well the chaining cost (or score) correlates with edit distance. We use absolute value of Pearson correlation coefficients for a comparison. In this experiment, we simulated 100 query strings within three similarity ranges: $90-100\%, 80-90\%$ and $75-80\%$. Table 2

[2] https://github.com/Martinsos/edlib/tree/master/test_data

Table 1: Runtime and memory usage comparison of edit distance solver Edlib and co-linear chaining methods ChainX, Clasp, Nucmer4 and Minimap2. Runtime is measured in milliseconds across the columns, and memory usage (Mem) is noted in MBs. In this experiment, ChainX, Clasp-sop, Clasp-linear and Nucmer4 used maximal unique matches (MUMs) of length ≥ 20 as input anchors, while Minimap2 used fixed-length minimizer k-mers of size 20.

Similarity	No. of MUMs	Edlib Time (Mem)	ChainX Time (Mem)	Clasp-sop Time (Mem)	Clasp-linear Time (Mem)	Nucmer4 Time (Mem)	Minimap2 Time (Mem)
Semi-global pairwise sequence comparisons, sequence sizes $10^4 \times 5 * 10^6$							
99%	67	190 **(17)**	2.0 (57)	1.8 (57)	**0.9** (57)	1.8 (60)	1.9 (75)
97%	160	642 **(17)**	2.9 (57)	4.8 (57)	**1.8** (57)	4.1 (60)	2.3 (75)
94%	176	1165 **(17)**	3.0 (57)	5.9 (57)	2.1 (57)	3.2 (60)	**1.6** (75)
90%	135	2168 **(17)**	5.6 (57)	4.7 (57)	2.0 (57)	5.5 (60)	**1.9** (75)
80%	28	2360 **(17)**	4.2 (57)	2.5 (57)	**2.2** (57)	3.4 (60)	4.3 (75)
70%	3	4297 **(17)**	3.7 (57)	2.2 (57)	2.3 (57)	5.5 (60)	**1.1** (75)
Global pairwise sequence comparisons, sequence sizes $10^6 \times 10^6$							
99%	7012	949 **(8)**	**47.2** (24)	1236.8 (1800)	182.8 (257)	68.7 (26)	193.5 (35)
97%	15862	1308 **(8)**	490.4 (24)	5363.7 (8742)	765.4 (1278)	**87.8** (26)	179.0 (36)
94%	18389	2613 **(8)**	677.9 (24)	11737.1 (20501)	1021.0 (1694)	**113.5** (27)	116.9 (34)
90%	14472	6233 **(8)**	851.5 (24)	5110.3 (8277)	115.3 (27)	121.8 (26)	**94.8** (33)
80%	2964	12506 **(8)**	158.8 (24)	504.8 (572)	133.7 (24)	148.9 (26)	**69.5** (32)
70%	195	29602 **(8)**	136.5 (23)	140.6 (23)	139.6 (23)	167.3 (26)	**55.6** (32)

shows the correlation achieved by all the tools. Here we observe that ChainX and Clasp-sop are more consistent in terms of maintaining high correlation across all similarity ranges. Between the two, ChainX was shown to offer superior scalability in terms of runtime and memory usage (Table 1). Hence, ChainX can be useful in practice when good performance and accuracy is desired across a wide similarity range.

Table 2: Absolute Pearson correlation coefficients of chaining costs (or scores) computed by various methods with the corresponding edit distances. 100 query strings were simulated and matched to the target string within each similarity range.

Seq. sizes	Similarity	Correlation coefficient				
		ChainX	Clasp-sop	Clasp-linear	Nucmer4	Minimap2
Semi-global sequence comparisons						
$10^4 \times 5 * 10^6$	90%-100%	**0.996**	0.994	0.986	0.968	0.995
$10^4 \times 5 * 10^6$	80%-90%	0.975	**0.976**	0.786	0.864	0.958
$10^4 \times 5 * 10^6$	75%-80%	**0.927**	0.915	0.732	0.733	0.808
Global sequence comparisons						
$10^6 \times 10^6$	90%-100%	**0.999**	0.997	0.994	0.991	**0.999**
$10^6 \times 10^6$	80%-90%	**0.998**	**0.998**	0.922	0.955	0.996
$10^6 \times 10^6$	75%-80%	0.992	**0.993**	0.871	0.907	0.952

Effect of anchor type and minimum match length. How many anchors are given as input will naturally affect the performance and output quality of a chaining algorithm. We tested runtime and correlation with edit distance achieved by ChainX while varying the anchor type (MUMs/MEMs) and minimum match-length l_{min} parameter (Table 3). When MUMs are used as anchors, we observe good scalability, and lowering l_{min} from 20 to 10 improves the correlation, but the correlation saturates afterwards. This is because very short exact matches will unlikely be unique and won't be selected as MUMs. However, when MEMs

Table 3: Effect of anchor pre-computation method on the performance of ChainX. Total runtime to do 100 pairwise semi-global sequence comparisons (sequence size: $10^4 \times 5 * 10^6$) is measured in seconds, and correlation (corr.) with the corresponding edit distances is computed using Pearson correlation coefficient.

Similarity	Using MUMs			Using MEMs		
	$len \geq 20$	$len \geq 10$	$len \geq 7$	$len \geq 20$	$len \geq 10$	$len \geq 7$
	Time (corr.)	Time (corr.)	Time (corr.)	Time (corr.)	Time (corr.)	Time (corr.)
90%-100%	7.2 (0.996)	**2.9** (0.997)	3.5 (0.997)	5.1 (0.996)	8.1 (0.997)	2652 **(0.998)**
80%-90%	4.5 (0.975)	5.6 (0.992)	**3.2** (0.992)	4.5 (0.975)	7.4 (0.993)	5413 **(0.995)**
75%-80%	5.3 (0.927)	5.9 (0.977)	**1.9** (0.977)	5.0 (0.927)	10.9 (0.987)	9221 **(0.992)**

are used as anchors, correlation continues to improve with decreasing minimum length parameter, however, runtime grows exponentially. Excessive count of anchors improves the correlation but then anchor chaining becomes computationally demanding.

7 Conclusions

This work provides new algorithms for co-linear chaining, a fundamental problem in bioinformatics. Variants of this technique have been regularly used in alignment tools since four decades [32]. We addressed an open problem pertaining to the general case of this problem which allows anchor overlaps and penalizes gap cost between adjacent chained anchors. The proposed algorithms for multiple versions of this problem are provably-optimal and efficient, and can be incorporated in read mappers. We also discussed a new cost function for the co-linear chaining problem that enabled us to establish the first mathematical link between co-linear chaining and the edit distance problem. This result is a useful addition to a prior result [16] where a connection between the co-linear chaining problem and the longest common subsequence problem was made.

Acknowledgements

This research is supported in part by the U.S. National Science Foundation (NSF) grants CCF-1704552, CCF-1816027, CCF-2112643, and funding from the Indian Institute of Science.

References

1. Abouelhoda, M., Ohlebusch, E.: Chaining algorithms for multiple genome comparison. Journal of Discrete Algorithms 3(2-4), 321–341 (2005)
2. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Coconut: an efficient system for the comparison and analysis of genomes. BMC bioinformatics 9(1), 476 (2008)
3. Backurs, A., Indyk, P.: 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. pp. 51–58 (2015)

4. de Berg, M., Cheong, O., van Kreveld, M.J., Overmars, M.H.: Computational geometry: algorithms and applications, 3rd Edition. Springer (2008), https://www.worldcat.org/oclc/227584184
5. Bray, N., Dubchak, I., Pachter, L.: Avid: A global alignment program. Genome research 13(1), 97–102 (2003)
6. Chaisson, M.J., Tesler, G.: Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. BMC bioinformatics 13(1), 238 (2012)
7. Delcher, A.L., Kasif, S., et al.: Alignment of whole genomes. Nucleic acids research 27(11), 2369–2376 (1999)
8. Eppstein, D., Galil, Z., Giancarlo, R., Italiano, G.F.: Sparse dynamic programming i: linear cost functions. Journal of the ACM (JACM) 39(3), 519–545 (1992)
9. Eppstein, D., Galil, Z., et al.: Sparse dynamic programming ii: convex and concave cost functions. Journal of the ACM (JACM) 39(3), 546–567 (1992)
10. Hoppenworth, G., Bentley, J.W., Gibney, D., Thankachan, S.V.: The fine-grained complexity of median and center string problems under edit distance. In: 28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy. vol. 173, pp. 61:1–61:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
11. Jain, C., Rhie, A., Hansen, N., Koren, S., Phillippy, A.M.: A long read mapping method for highly repetitive reference sequences. bioRxiv (2020)
12. Kalikar, S., Jain, C., Md, V., Misra, S.: Accelerating long-read analysis on modern CPUs. bioRxiv (2021)
13. Kurtz, S., et al.: Versatile and open software for comparing large genomes. Genome biology 5(2), R12 (2004)
14. Li, H.: Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics 34(18), 3094–3100 (2018)
15. Li, H., Feng, X., Chu, C.: The design and construction of reference pangenome graphs with minigraph. Genome Biology 21(1), 265 (2020)
16. Mäkinen, V., Sahlin, K.: Chaining with overlaps revisited. In: 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark. vol. 161, pp. 25:1–25:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
17. Mäkinen, V., Tomescu, A.I., Kuosmanen, A., Paavilainen, T., Gagie, T., Chikhi, R.: Sparse dynamic programming on dags with small width. ACM Trans. Algorithms 15(2), 29:1–29:21 (2019), https://doi.org/10.1145/3301312
18. Marçais, G., Delcher, A.L., et al.: Mummer4: A fast and versatile genome alignment system. PLoS computational biology 14(1), e1005944 (2018)
19. Morgenstern, B.: A simple and space-efficient fragment-chaining algorithm for alignment of DNA and protein sequences. Applied Mathematics Letters 15(1), 11–16 (2002)
20. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM (JACM) 46(3), 395–415 (1999)
21. Myers, G., Miller, W.: Chaining multiple-alignment fragments in sub-quadratic time. In: SODA. vol. 95, pp. 38–47 (1995)
22. Otto, C., Hoffmann, S., Gorodkin, J., Stadler, P.F.: Fast local fragment chaining using sum-of-pair gap costs. Algorithms for Molecular Biology 6(1), 4 (2011)
23. Ren, J., Chaisson, M.J.: lra: A long read aligner for sequences and contigs. PLOS Computational Biology 17(6), e1009078 (2021)
24. Sahlin, K., Mäkinen, V.: Accurate spliced alignment of long RNA sequencing reads. Bioinformatics 37(24), 4643–4651 (2021)

25. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. pp. 76–85 (2003)
26. Sedlazeck, F.J., et al.: Accurate detection of complex structural variations using single-molecule sequencing. Nature methods 15(6), 461–468 (2018)
27. Shibuya, T., Kurochkin, I.: Match chaining algorithms for cDNA mapping. In: Algorithms in Bioinformatics, Third International Workshop, WABI 2003, Budapest, Hungary, September 15-20, 2003, Proceedings. pp. 462–475 (2003)
28. Šošić, M., Šikić, M.: Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. Bioinformatics 33(9), 1394–1395 (2017)
29. Ukkonen, E.: Algorithms for approximate string matching. Information and control 64(1-3), 100–118 (1985)
30. Uricaru, R., et al.: Novel definition and algorithm for chaining fragments with proportional overlaps. Journal of Computational Biology 18(9), 1141–1154 (2011)
31. Vyverman, M., De Baets, B., et al.: essamem: finding maximal exact matches using enhanced sparse suffix arrays. Bioinformatics 29(6), 802–804 (2013)
32. Wilbur, W.J., Lipman, D.J.: Rapid similarity searches of nucleic acid and protein data banks. Proceedings of the National Academy of Sciences 80(3), 726–730 (1983)

Appendix

1 Algorithm Pseudocode

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 2: For co-linear chaining with overlaps and gap costs

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. 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 Missing proofs

Lemma 4. *For a minimal anchor chain under strict precedence, the greedy algorithm described in Lemma 1 produces an optimal alignment.*

Proof. This follows from an exchange argument. Suppose there exists an optimal alignment $\mathcal{M}$ on the anchor chain $\mathcal{B}$ 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 leftmost edges $e = (S_1[h], S_2[k])$ in $\mathcal{M}$ and $e_G = (S_1[h_G], S_2[k_G])$ in $\mathcal{M}_G$ that are not equal. Let e_{prev} be the previous edge on which the $\mathcal{M}$ and $\mathcal{M}_G$ coincided. We claim e can be replaced with e_G without increasing the edit cost.

- Case: e_G is solid and left of e. Then e can clearly be exchanged for e_G with no increase in edit cost.
- Case: e_G is solid and not left of e. We assume WLOG $k < k_G$ (Figure 7). Suppose e_G is supported by an anchor I. We claim that (i) $h \geq h_G$, and (ii) there exists an edge $\tilde{e} = (S_1[\tilde{h}], S_2[\tilde{k}])$ in $\mathcal{M}$ to the right of e and supported by I. Note that $\tilde{h} - h_G + 1$ solid edges can be obtained by matching $S_1[h_G, \tilde{h}]$ to $S_2[k_G, \tilde{k}]$, and that this alignment has at most the edit cost incurred within the intervals $[h_G, \tilde{h}]$ in S_1 and $[k, \tilde{k}]$ in S_2 by $\mathcal{M}$. Thanks to (i) and (ii),

swapping all of the edges in $\mathcal{M}$ inside these intervals with the $\tilde{h} - h_G + 1$ edges supported by I mentioned above can be done without effecting edges outside those intervals. The proofs for claims (i) and (ii) are shown below.

i $h \geq h_G$*:* Suppose to the contrary, that $h < h_G$. Let $e_{prev} = (S_1[h_{prev}], S_2[k_{prev}])$. Since $h_{prev} < h < h_G$ and $k_{prev} < k < k_G$, this would imply that there are deletions (x's) at both $S_1[h]$ and $S_2[k]$ in $\mathcal{M}_G$. However, the greedy algorithm would instead create an edge including $S_1[h]$ or $S_2[k]$ (or both), causing there to be an edge to the left of e_G and to the right of e_{prev}. This contradicts our assumption that e_G is the first edge in $\mathcal{M}_G$ to the right of e_{prev}.

ii *An edge* $\tilde{e} = (S_1[\tilde{h}], S_2[\tilde{k}])$ *exists in* $\mathcal{M}$ *to the right of* e *and supported by* I*:* We claim e_G is the leftmost edge supported by I in $\mathcal{M}_G$. Otherwise, since only consecutive edges supported by an anchor are taken in the greedy algorithm (Cases 3, 4, 5 in proof of Lemma 1), $e_{prev} = (S_1[h_G - 1], S_2[k_G - 1])$. Then e_{prev} was also in $\mathcal{M}$ by our assumption that $\mathcal{M}_G$ and $\mathcal{M}$ agreed until e_G. However, $e_{prev} = (S_1[h_G - 1], S_2[k_G - 1])$ would cross or share a vertex with $e = (S_1[h], S_2[k])$ in $\mathcal{M}$ since $h_G - 1 < h$ and $k \leq k_G - 1$, a contradiction. Now, if no other edges supported by I are in $\mathcal{M}$, then $\mathcal{B}$ is not minimal. Hence, we can assume that an edge $\tilde{e} = (S_1[\tilde{h}], S_2[\tilde{k}])$ exists in $\mathcal{M}$ to the right of e and supported by I.

– Case: e_G is a dashed edge. Since dashed edges are only created in Case 5 of the proof of Lemma 1 (Figure 6), $e_{prev} = (S_1[h_G - 1], S_2[k_G - 1])$ must have been either the rightmost supported edge for some anchor, or another dashed edge. Additionally, no other supported edges are possible that include either $S_1[h]$ or $S_2[k]$. Here the dashed edge e_G is optimal and swapping e with e_G will reduce the edit cost. □

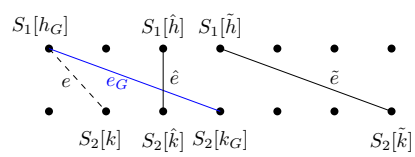

Fig. 7: Consider a greedy alignment $\mathcal{M}_G$ and optimal alignment $\mathcal{M}$ where the leftmost difference is e and e_G (in blue). Here, $\hat{e}$ prevents swapping e and e_G without modifying additional edges in $\mathcal{M}$.

Lemma 6. *Algorithm 1 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 located 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

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

The total expected time is a constant factor from $B_1 n(1+B_2+\ldots+B_2^{\lceil \log_{B_2} OPT \rceil}) = O(n \cdot OPT)$. $\square$