Sequence to graph alignment using gap-sensitive co-linear chaining

Ghanshyam Chandra and Chirag Jain

Department of Computational and Data Sciences, Indian Institute of Science, Bangalore KA 560012, India
{ghanshyamc,chirag}@iisc.ac.in

Abstract. Co-linear chaining is a widely used technique in sequence alignment tools that follow seed-filter-extend methodology. It is a mathematically rigorous approach to combine short exact matches. For co-linear chaining between two sequences, efficient subquadratic-time chaining algorithms are well-known for linear, concave and convex gap cost functions [Eppstein *et al.* JACM'92]. However, developing extensions of chaining algorithms for directed acyclic graphs (DAGs) has been challenging. Recently, a new sparse dynamic programming framework was introduced that exploits small path cover of pangenome reference DAGs, and enables efficient chaining [Makinen *et al.* TALG'19, RECOMB'18]. However, the underlying problem formulation did not consider gap cost which makes chaining less effective in practice. To address this, we develop novel problem formulations and optimal chaining algorithms that support a variety of gap cost functions. We demonstrate empirically the ability of our provably-good chaining implementation to align long reads more precisely in comparison to existing aligners. For mapping simulated long reads from human genome to a pangenome DAG of 95 human haplotypes, we achieve 98.7% precision while leaving $< 2\%$ reads unmapped.
Implementation: https://github.com/at-cg/minichain

Keywords: Variation graph · Sparse dynamic programming · minimum path cover · Pangenome.

1 Introduction

A significant genetic variation rate among genomes of unrelated humans, plus the growing availability of high-quality human genome assemblies, has accelerated computational efforts to use pangenome reference graphs for common genomic analyses [24,40,41]. The latest version of industry-standard DRAGEN software by Illumina now uses a pangenome graph for mapping reads in highly polymorphic regions of a human genome [14]. For surveys of the recent algorithmic developments in this area, see [2,6,10,33]. Among the many computational tasks associated with pangenome graphs, sequence-to-graph alignment remains a core computational problem. Accurate alignments are required for variation analysis and construction of pangenome graph from multiple genomes [9,22]. Sequence-to-graph alignment is also useful in other applications including genome assembly [13] and long-read error correction [39].

Suppose a pangenome graph is represented as a character labeled DAG $G(V, E)$ where each vertex $v \in V$ is labeled with a character from alphabet {A,C,G,T}. The sequence-to-DAG alignment problem seeks a path in G that spells a string with minimum edit distance from the input query sequence. An $O(m|E|)$ time algorithm for this problem has long been known, where m is the length of input sequence [29]. Conditioned on Strong Exponential Time Hypothesis (SETH), the $O(m|E|)$ algorithm is already worst-case optimal up to sub-polynomial improvements because algorithms for computing edit distance in strongly sub-quadratic time cannot exist under SETH [3]. As a result, heuristics must be used for alignment of high-throughput sequencing data against large DAGs to obtain approximate solutions in less time and space.

All practical long read to DAG aligners that scale to large genomes rely on seed–filter–extend methodology [8,22,25,27,34]. The first step is to find a set of *anchors* which indicate short exact matches, e.g., k-mer or minimizer matches, between substrings of a sequence to subpaths in a DAG. This is followed by a clustering step that identifies promising subsets of anchors which should be kept within the alignments. Different aligners implement this step in different ways. *Co-linear chaining* is a mathematically rigorous approach to do clustering of anchors. It is well studied for the case of sequence-to-sequence alignment [1,11,12,16,26,28,32], and is widely used in present-day long read to reference sequence aligners [18,21,35,37,38]. For the sequence-to-sequence alignment case, the input to the chaining problem is a set of N weighted anchors where each anchor is a pair of intervals in the two sequences that match exactly. A *chain* is defined as an ordered subset of anchors such that their intervals appear in increasing order in both sequences (Figure 1a). The desired output of the co-linear chaining problem is the chain with maximum score where score of a chain is calculated by the sum of weights of the anchors in the chain minus the penalty corresponding to gaps between adjacent anchors. For linear gap costs, this problem is solvable in $O(N \log N)$ time by using range-search queries [1].

Solving chaining problem for sequence-to-DAG alignment remained open until Makinen *et al.* [27] introduced a framework that enables sparse dynamic programming on DAGs. Suppose K denotes cardinality of a minimum-sized set of paths such that every vertex is covered by at least one path. The algorithm in [27] works by mimicking the sequence-to-sequence chaining algorithm on each path of the *minimum path cover*. After a polynomial-time indexing of the DAG, their algorithm requires $O(KN \log N + K|V|)$ time for chaining. Parameterizing the time complexity in terms of K is useful because K is expected to be small for pangenome DAGs. This result was further improved in [25] with an $O(KN \log KN)$ time algorithm. However, the problem formulations in these works did not include gap cost. Without penalizing gaps, chaining is less effective [16]. A challenge in enforcing gap cost is that measuring gap between two loci in a DAG is not a simple arithmetic operation like in a sequence [20].

We present novel co-linear chaining problem formulations for sequence-to-DAG alignment that penalize gaps, and we develop efficient algorithms to solve them. We carefully design gap cost functions such that they enable us to adapt

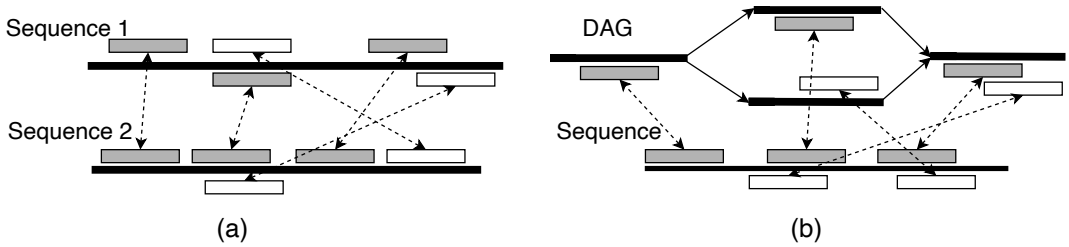


Fig. 1: Illustration of co-linear chaining for (a) sequence-to-sequence and (b) sequence-to-DAG alignment. It is assumed that vertices of DAG are labeled with strings. Pairs of rectangles joined by dotted arrows denote anchors (exact matches). A subset of these anchors that form a valid chain are shown in gray.

the sparse dynamic programming framework of Makinen *et al.* [27], and solve the chaining problem optimally in $O(KN \log KN)$ time. We implemented and benchmarked one of our proposed algorithms to demonstrate scalability and accuracy gains. Our experiments used human pangenome DAGs built by using 94 high quality *de novo* haplotype assemblies provided by the Human Pangenome Reference Consortium [24] and CHM13 human genome assembly provided by the Telomere-to-Telomere consortium [30]. Using a simulated long read dataset with 0.5× coverage, we demonstrate that our implementation achieves the highest read mapping precision (98.7%) among the existing methods (Minigraph: 98.0%, GraphAligner: 97.0% and GraphChainer: 95.1%). In this experiment, our implementation used 24 minutes and 25 GB RAM with 32 threads, demonstrating that the time and memory requirements are well within practical limits.

2 Concepts and notations

2.1 Co-linear chaining on sequences revisited

Let R and Q be two sequences over alphabet $\Sigma = \{A, C, G, T\}$. Let $M[1..N]$ be an array of anchors. Each anchor is denoted using an interval pair $([x..y], [c..d])$ with the interpretation that substring $R[x..y]$ matches substring $Q[c..d]$, $x, y, c, d \in \mathbb{N}$. Anchors are typically either fixed-length matches (e.g., using k-mers) or variable-length matches (e.g., maximal exact matches). Suppose function *weight* assigns weights to the anchors. The co-linear chaining problem seeks an ordered subset $S = s_1 s_2 \cdots s_p$ of anchors from M such that

- for all $2 \leq j \leq p$, s_{j-1} precedes ($\prec$) s_j, i.e., $s_{j-1}.y < s_j.x$ and $s_{j-1}.d < s_j.c$.
- S maximises chaining score, defined as $\sum_{j=1}^{p} weight(s_j) - \sum_{j=2}^{p} gap(s_{j-1}, s_j)$. Define $gap(s_{j-1}, s_j)$ as $f(gap_R(s_{j-1}, s_j), gap_Q(s_{j-1}, s_j))$, where $gap_R(s_{j-1}, s_j) = s_j.x - s_{j-1}.y - 1$, $gap_Q(s_{j-1}, s_j) = s_j.c - s_{j-1}.d - 1$ and $f(g_1, g_2) = g_1 + g_2$.

The above problem can be trivially solved in $O(N^2)$ time and $O(N)$ space. First sort the anchors by the component $M[\cdot].x$, and let $T[1..N]$ be an integer array containing a permutation of set $[1..N]$ which specifies the sorted order, i.e.,

$M[T[1]].x \leq M[T[2]].x \leq \ldots \leq M[T[N]].x$. Define array $C[1..N]$ such that $C[j]$ is used to store a partial solution, i.e., the score of an optimal chain that ends at anchor $M[j]$. Naturally, the final score will be obtained as $\max_j C[j]$. Array C can be filled by using the following dynamic programming recursion: $C[T[j]] = weight(M[T[j]]) + \max\big(0, \max_{i:M[i] \prec M[T[j]]}(C[i] - gap(M[i], M[T[j]]))\big)$, in increasing order of j. A straight-forward way of computing $C[T[j]]$ will need an $O(N)$ linear scan of arrays C and M, resulting in overall $O(N^2)$ time. However, the $O(N^2)$ algorithm can be optimized to use $O(N \log N)$ time by using the following search tree data structure (ref. [4]).

Lemma 1. *Let n be the number of leaves in a balanced binary search tree, each storing a $(key, value)$ pair. The following operations can be supported in $O(\log n)$ time:*

- *$\mathsf{update}(k, val)$: For the leaf w with $key = k$, $value(w) \leftarrow \max(value(w), val)$.*
- *$\mathsf{RMQ}(l, r)$: Return $\max\{value(w) \mid l < key(w) < r\}$. This is range maximum query.*

Moreover, given n $(key, value)$ pairs, the balanced binary search tree can be constructed in $O(n \log n)$ time and $O(n)$ space.

The dynamic programming recursion for array $C[1..N]$ can be computed more efficiently using range maximum queries [1,11]. To achieve this, a search tree needs to be initialized, updated and queried properly (Algorithm 1). Note that $\text{argmax}_{i:M[i] \prec M[j]}(C[i] - gap(M[i], M[j]))$ is equal to $\text{argmax}_{i:M[i] \prec M[j]}(C[i] + M[i].y + M[i].d)$. Accordingly, we compute optimal $C[j]$ in Line 11 by using an $O(\log N)$ time RMQ operation of the form $M[i].d \in (0, M[j].c)$ that returns maximum $C[i] + M[i].y + M[i].d$ from search tree $\mathcal{T}$. The algorithm performs N update and N RMQ operations over search tree $\mathcal{T}$ of size at most N, thus solving the problem in $O(N \log N)$ time and $O(N)$ space.

Algorithm 1: $O(N \log N)$ time chaining between two sequences

Input: Array of weighted anchors $M[1..N]$
Output: Array $C[1..N]$ such that $C[j]$ = score of an optimal chain that ends at $M[j]$

1 Initialize search tree $\mathcal{T}$ using keys $\{M[j].d \mid 1 \leq j \leq N\}$ and values $-\infty$
2 Initialize $C[j]$ as $weight(M[j])$, for all $j \in [1, N]$
3 /* Create array Z that stores tuples of the form $(pos, task, anchor)$, where $pos \in \mathbb{N}$, $anchor \in [1, N]$ and $task \in \{0, 1\}$. $task$ is either 0 or 1 for querying or updating the search tree $\mathcal{T}$ respectively.*/
4 **for** $j \leftarrow 1$ ***to*** N **do**
5 | $Z.push(M[j].x, 0, j)$
6 | $Z.push(M[j].y, 1, j)$
7 **end**
8 **for** $z \in Z$ *in lexicographically ascending order based on the key* $(pos, task)$ **do**
9 | $j \leftarrow z.anchor, wt \leftarrow weight(M[j])$
10 | **if** $z.task = 0$ **then**
11 | | $C[j] \leftarrow \max(C[j], wt + \mathcal{T}.\mathsf{RMQ}(0, M[j].c) - M[j].x - M[j].c + 2)$
12 | **else**
13 | | $\mathcal{T}.\mathsf{update}(M[j].d, C[j] + M[j].y + M[j].d)$
14 **end**

2.2 Sparse dynamic programming on DAGs using minimum path cover

Our work builds on the work of Makinen *et al.* [27], who provided a parameterized algorithm to extend co-linear chaining on DAGs without considering gap costs. In the following, we present useful notations and a summary of their algorithm.

In a weighted string-labeled DAG $G(V, E, \sigma)$, function $\sigma : V \rightarrow \Sigma^+$ labels each vertex $v \in V$ with string $\sigma(v)$. Edge $v \rightarrow u$ has length $|\sigma(v)|$. The length of a path in G is the sum of the lengths of the edges traversed in the path. Let $Q \in \Sigma^+$ be a query sequence. Let M be an array of N anchors. An anchor is denoted using a 3-tuple of the form $(v, [x..y], [c..d])$ with the interpretation that substring $\sigma(v)[x..y]$ in DAG G matches substring $Q[c..d]$, where $x, y, c, d \in \mathbb{N}$ and $v \in V$ (Figure 2). A *path cover* of DAG $G(V, E)$ is a set of paths in G such that every vertex in V belongs to at least one path. A *minimum path cover* (MPC) is one having the minimum number of paths. If K denotes the size of MPC of DAG G, then MPC can be computed either in $O(K|E| \log |V|)$ [27] or $O(K^3|V| + |E|)$ [5] time.

To extend co-linear chaining for sequence-to-DAG alignment, we can use a search tree containing keys equal to the sequence coordinates of anchors, similar to Algorithm 1. However, the order in which the search tree should be queried and updated is not trivial with DAGs. Makinen *et al.* [27] suggested decomposing the DAG into a path cover $\{P_1, ..., P_K\}$, and then performing the computation only along these paths. The algorithm uses K search trees $\{\mathcal{T}_1, ..., \mathcal{T}_K\}$, one per path. Search tree $\mathcal{T}_i$ maintains $M[\cdot].d$ as keys and partial solutions $C[\cdot]$ as values of all the anchors that lie on path P_i. Similar to Algorithm 1, the K search trees need to be updated and queried in a proper order. Suppose $\mathcal{R}(v) \subseteq V$ denotes the set of vertices which can reach v using a path in G. Set $\mathcal{R}(v)$ always includes v. Define $last2reach(v, i)$ as the last vertex on path P_i that belongs to $\mathcal{R}(v)$, if one exists. Also define $paths(v)$ as $\{i : P_i \text{ covers } v\}$. Naturally $last2reach(v, i) = v$ iff $i \in paths(v)$. The main algorithm works by visiting vertices u of G in topological order, and executing the following two tasks:

- **Compute optimal scores of all anchors in vertex u**: First, process all the anchors for which $M[j].v = u$ in the same order that is used for co-linear chaining on two sequences (Algorithm 1). While performing an `update` task, update all search trees $\mathcal{T}_i$, for all $i \in paths(u)$. Similarly, while performing a range query, query search trees $\mathcal{T}_i$ to maximize $C[j]$.
- **Update partial scores of selected anchors outside vertex u**: Next, for all pairs (w, i), $w \in V, i \in [1, K]$ such that $last2reach(w, i) = u$ and $i \notin paths(w)$, query search tree $\mathcal{T}_i$ to update score $C[j]$ of every anchor $M[j]$ for which $M[j].v = w$.

Based on the above tasks, once the algorithm reaches $v \in V$ in the topological ordering, the scores corresponding to anchors in vertex v would have been updated from all other vertices that reach v. A well-optimized implementation of this algorithm uses $O(KN \log KN)$ time [25]. This result assumes that the DAG is preprocessed, i.e., path cover and $last2reach$ information is precomputed in $O(K^3|V| + K|E|)$ time.

3 Problem formulations

We develop six problem formulations for co-linear chaining on DAGs with different gap cost functions. In each problem, we seek an ordered subset $S = s_1 s_2 \cdots s_p$ of anchors from array M such that

- for all $2 \leq j \leq p$, s_{j-1} precedes ($\prec$) s_j, i.e., the following three conditions are satisfied (i) $s_{j-1}.d < s_j.c$, (ii) $s_{j-1}.v \in \mathcal{R}(s_j.v)$, and (iii) $s_{j-1}.y < s_j.x$ if $s_{j-1}.v = s_j.v$.
- S maximizes the chaining score defined as $\sum_{j=1}^{p} weight(s_j) - \sum_{j=2}^{p} gap(s_{j-1}, s_j)$. Define $gap(s_{j-1}, s_j)$ as $f(gap_G(s_{j-1}, s_j), gap_S(s_{j-1}, s_j))$, where functions gap_G and gap_S will be used to specify gap cost in the DAG and the query sequence respectively.

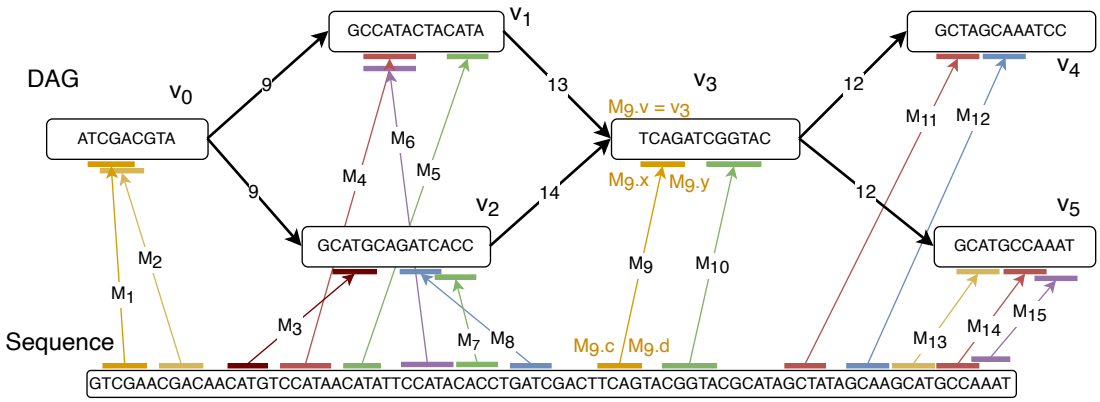


Fig. 2: An example showing multiple anchors as input for co-linear chaining. The DAG has a minimum path cover of size two $\{(v_0, v_1, v_3, v_4), (v_0, v_2, v_3, v_5)\}$. Anchors $M_1, M_4, M_5, M_9, M_{10}, M_{11}, M_{12}$ form a valid chain. The interval coordinates of anchor M_9 in the sequence and the DAG are annotated for illustration.

$gap_S(s_{j-1}, s_j)$ equals $s_j.c - s_{j-1}.d - 1$, i.e., the count of characters in sequence Q that occur between the two anchors. However, defining gap_G is not as straightforward because multiple paths may exist from $s_{j-1}.v$ to $s_j.v$, and the correct alignment path is unknown. We formulate and solve the following problems:

Problems 1a-1c: $gap_G(s_{j-1}, s_j)$ is computed by using the shortest path from $s_{j-1}.v$ to $s_j.v$. Suppose $D(v_1, v_2)$ denotes the shortest path length from vertex v_1 to v_2 in G. We seek the optimal chaining score when

$$gap_G(s_{j-1}, s_j) = D(s_{j-1}.v, s_j.v) + (s_j.x - s_{j-1}.y - 1).$$

The above expression calculates the count of characters in the string path between anchors s_{j-1} and s_j. Define Problems 1a, 1b and 1c using $f(g_1, g_2) = g_1 + g_2$, $f(g_1, g_2) = max(g_1, g_2)$ and $f(g_1, g_2) = |g_1 - g_2|$ respectively. These

definitions of function f are motivated from the previous co-linear chaining formulations for sequence-to-sequence alignment [1,28].

Problems 2a-2c: $gap_G(s_{j-1}, s_j)$ is measured using a path from $s_{j-1}.v$ to $s_j.v$ that is chosen based on path cover $\{P_1, ..., P_K\}$ of DAG G. For each $i \in paths(s_{j-1}.v)$, consider the following path in G that starts from source $s_{j-1}.v$ along the edges of path P_i till the middle vertex $last2reach(s_j.v, i)$, and finally reaches destination $s_j.v$ by using the shortest path from $last2reach(s_j.v, i)$ to $s_j.v$. Among $|paths(s_{j-1}.v)|$ such possible paths, measure $gap_G(s_{j-1}, s_j)$ using the path which minimizes $gap(s_{j-1}, s_j) = f(gap_G(s_{j-1}, s_j), gap_S(s_{j-1}, s_j))$. More precisely, $gap_G(s_{j-1}, s_j)$ equals the element of the set

$$\{dist2begin(\mu, i) - dist2begin(s_{j-1}.v, i) + D(\mu, s_j.v) + s_j.x \\ -s_{j-1}.y - 1 \mid i \in paths(s_{j-1}.v), \mu = last2reach(s_j.v, i)\}$$

which minimizes $gap(s_{j-1}, s_j)$, where $dist2begin(v, i)$ denotes the length of subpath of path P_i from the start of P_i to v. We will show that this definition enables significantly faster parameterized algorithms with respect to K. Again, define Problems 2a, 2b and 2c with $f(g_1, g_2) = g_1 + g_2$, $f(g_1, g_2) = max(g_1, g_2)$ and $f(g_1, g_2) = |g_1 - g_2|$ respectively.

4 Proposed algorithms

Our algorithm to address Problems 1a-1c uses a brute-force approach that evaluates all $O(N^2)$ pairs of anchors. We use single-source shortest distances computations for measuring gaps.

Lemma 2. *Problems* 1*a,* 1*b and* 1*c can be solved optimally in* $O(N(|V| + |E| + N))$ *time.*

Proof. We will process anchors in array $M[1..N]$ one by one in a topological order of $M[\cdot].v$. If there are two anchors with equal component $M[\cdot].v$, then the anchor with lower component $M[\cdot].x$ is processed first. Suppose DAG G' is obtained by reversing the edges of G. While processing anchor $M[j]$, we will compute partial score $C[j]$, i.e., the optimal score of a chain that ends at anchor $M[j]$. We identify all the anchors that precede $M[j]$ using a depth-first traversal starting from $M[j].v$ in G'. Next, we compute single-source shortest distances from $M[j].v$ in G' which requires $O(|V| + |E|)$ time for DAGs [7]. Finally, $C[j]$ is computed as $weight(M[j]) + \max\big(0, \max_{i:M[i] \prec M[j]}(C[i] - f(gap_G(M[i], M[j]), gap_S(M[i], M[j]))\big)$ in $O(N)$ time. □

The above algorithm is unlikely to scale to a mammalian dataset. We leave it open whether there exists a faster algorithm to solve Problems 1a-c. Next, we propose $O(KN \log KN)$ time algorithm for addressing Problem 2a, assuming $O(K^3|V|+K|E|)$ time preprocessing is done for DAG G. The preprocessing stage is required to compute (a) an MPC $\{P_1, \ldots, P_K\}$ of G, (b) $last2reach(v, i)$, (c) $D(last2reach(v, i), v)$ and (d) $dist2begin(v, i)$, for all $v \in V,\ i \in [1, K]$.

Lemma 3. *The preprocessing of DAG $G(V, E, \sigma)$ can be done in $O(K^3|V| + K|E|)$ time.*

Proof. An MPC $\{P_1, \ldots, P_K\}$ can be computed in $O(K^3|V| + |E|)$ time [5]. To compute the remaining information, we will use dynamic programming algorithms that process vertices $\in V$ in a fixed topological order. Suppose function $rank : V \rightarrow [1, |V|]$ assigns rank to each vertex based on its topological ordering. Let $\mathcal{N}(v)$ denote the set of adjacent in-neighbors of v. Similar to [27], $last2reach(v, i)$ is computed in $O(K|V| + K|E|)$ time for all $v \in V,\ i \in [1, K]$. Initialize $last2reach(v, i) = 0$ for all v and i. Then, use the following recursion:

$$last2reach(v, i) = \begin{cases} rank(v) & \text{if } i \in paths(v) \\ \max_{u:u\in\mathcal{N}(v)} last2reach(u, i) & \text{otherwise} \end{cases}$$

At the end of the algorithm, $last2reach(v, i) = 0$ will hold for only those pairs (v, i) for which $last2reach(v, i)$ does not exist. Next, we compute $D(last2reach(v, i), v)$, for all $v \in V,\ i \in [1, K]$, also in $O(K|V|+K|E|)$ time. Initialize $D(last2reach(v, i), v)$ $= \infty$ for all v and i. Then, update $D(last2reach(v, i), v)$

$$= \begin{cases} 0 & \text{if } \ last2reach(v, i) = v \\ \min_{u:u\in\mathcal{N}(v), last2reach(u,i)=last2reach(v,i)} D(last2reach(u, i), u) + |\sigma(u)| & \text{otherwise} \end{cases}$$

Finally, $dist2begin(v, i)$, for all $v \in V,\ i \in [1, K]$ is computed by linearly scanning K paths in $O(K|V|)$ time. □

Lemma 4. *Assuming DAG $G(V, E, \sigma)$ is preprocessed, Problem 2a can be solved in $O(KN \log KN)$ time and $O(KN)$ space.*

Proof. The choice of gap cost definition in Problem 2a allows us to make efficient use of range-search queries. Algorithm 2 gives an outline of the proposed dynamic programming algorithm. Similar to the previously discussed algorithms (Section 2.1), it also saves partial scores in array $C[1..N]$. We use K search trees, one per path. Search tree $\mathcal{T}_i$ maintains partial scores $C[\ ]$ of those anchors $M[j]$ whose coordinates on DAG are covered by path P_i. Each search tree is initialized with keys $M[j].d$, and values $-\infty$. Subsequently, K search trees are queried and updated in a proper order.

- If $K = 1$, i.e., when DAG G is a linear chain, the condition in Line 6 is always satisfied and the term $D(v, M[j].v)$ (Line 17) is always zero. In this case, Algorithm 2 works precisely as the co-linear chaining algorithm on two sequences (Algorithm 1).
- For $K > 1$, we use $last2reach$ information associated with vertex $M[j].v$ (Lines 9-11). This ensures that partial score $C[j]$ is updated from scores of the preceding anchors on path P_i for all $i \in [1, K] \setminus paths(M[j].v)$.

All the query and update operations done in the search trees together use $O(KN \log N)$ time because the count of these operations is bounded by $O(KN)$, and the size of each tree is $\leq N$. The sorting step in Line 14 requires $O(KN \log KN)$ time to sort $O(KN)$ tuples. The overall space required by K search trees and array Z is $O(KN)$. □

Algorithm 2: $O(KN \log KN)$ time chaining algorithm for Problem 2a

Input: Array of weighted anchors $M[1..N]$, preprocessed DAG $G(V, E, \sigma)$
Output: Array $C[1..N]$ such that $C[j]$ = score of an optimal chain that ends at $M[j]$

```
Initialize search tree T_i, for all i ∈ [1, K] using keys {M[j].d | 1 ≤ j ≤ N} and values −∞
Initialize C[j] as weight(M[j]), for all j ∈ [1, N]
/* Create array Z that stores tuples of the form (v, pos, task, anchor, path), where v ∈ V,
   pos ∈ ℕ, task ∈ {0, 1}, anchor ∈ [1, N] and path ∈ [1, K].*/
for j ← 1 to N do
    for i ← 1 to K do
        if i ∈ paths(M[j].v) then
            Z.push(M[j].v, M[j].x, 0, j, i)
            Z.push(M[j].v, M[j].y, 1, j, i)
        else if last2reach(M[j].v, i) exists then
            v ← last2reach(M[j].v, i)
            Z.push(v, |σ(v)| + 1, 0, j, i)
    end
end
for z ∈ Z in lexicographically ascending order based on the key (rank(v), pos, task) do
    j ← z.anchor, i ← z.path, v ← z.v, wt ← weight(M[j])
    if z.task = 0 then
        C[j] ← max(C[j], wt + T_i.RMQ(0, M[j].c) − M[j].x − dist2begin(v, i) −
          D(v, M[j].v) − M[j].c + 2)
    else
        T_i.update(M[j].d, C[j] + M[j].y + dist2begin(v, i) + M[j].d)
end
```

For simplicity of notations, we have not allowed an anchor to span two or more connected vertices in a DAG, but the proposed framework can be easily generalized to handle this [25,27]. Finally, we design algorithms for Problems 2b and 2c by using 2-dimensional RMQs. We summarize the result below and defer the proof to Appendix.

Lemma 5. *Assuming DAG $G(V, E, \sigma)$ is preprocessed, Problems 2b and 2c can be solved in $O(KN \log^2 N + KN \log KN)$ time and $O(KN \log N)$ space.*

5 Implementation details

Among the proposed algorithms, Algorithm 2 has the best time complexity. We implemented this algorithm in C++, and developed a practical long read alignment software Minichain.

Pangenome graph representation: A path in pangenome reference graph $G(V, E, \sigma)$ spells a sequence in a single orientation, whereas a read may be sampled from either the same or the opposite orientation due to the double-stranded nature of DNA. To address this internally in Minichain, for each vertex $v \in V$, we also add another vertex $\bar{v}$ whose string label is the reverse complement of string $\sigma(v)$. For each edge $u \to v \in E$, we also add the complementary edge $\bar{v} \to \bar{u}$. This process doubles the count of edges and vertices.

Optimization for whole-genome pangenome graphs: A pan-genome reference graph associated with a complete human genome is a union of weakly connected components, one per chromosome, because there is no edge which

connects two chromosome components. We actually maintain two components per chromosome, one being the reverse complement of the other. During both preprocessing and chaining stages of the proposed algorithms, each component is treated independently. The parameter K in our time complexity results is determined by the maximum K value among the components. We use GraphChainer implementation [25] to compute minimum path cover and range queries. We also optimize runtime by performing parallel preprocessing of different components (Lemma 3) using multiple threads.

Computing multiple best chains and confidence scores: When a read is sampled from a repetitive region of a genome, computing read's true path of origin becomes challenging. Practical methods often report more than one alignment per read in such cases. The highest-scoring alignment is marked as *primary* alignment, and the remaining are marked as *secondary*. Additionally, based on the score difference between the primary and the highest-scoring secondary alignment, a confidence score $\in [0, 60]$ is provided as *mapping quality* that represents the likelihood that the primary alignment is correct [23]. In Minichain, we also implement an algorithm to identify multiple high-scoring chains so that multiple base-to-base alignment records can be reported to a user. Algorithm 2 returns partial scores $C[1..N]$ in the end. We perform backtracking from anchor $\text{argmax}_j\, C[j]$ to compute the optimal chain. The anchors involved in this chain are marked as *visited*. Iteratively, we check presence of another chain (a) whose score is $\geq \tau \cdot \max_j C[j]$, where $\tau \in [0, 1]$ is a user-specified threshold with default value 0.95, and (b) none of the anchors in the chain are previously *visited*. We stop when no additional chains exist that satisfy these conditions.

Computing anchors and final base-to-base alignments: In Minichain, we use the seeding and base-to-base alignment methods from Minigraph [22]. The seeding method in Minigraph works by identifying common minimizers between query sequence and string labels $\sigma(v)$ of graph vertices. Given a pre-defined ordering of all k-mers and w consecutive k-mers in a sequence, (w, k)-minimizer is the smallest k-mer among the w k-mers [36]. The common minimizer occurrences between a query and vertex labels form anchors. In our experiments, we use same parameters $k = 17, w = 11$ as Minigraph. The weight of each anchor is k times a user-specified constant which is set to 200 by default. Algorithm 2 is used to compute the best chains and discard those anchors which do not contribute to these chains. Finally, we return the filtered anchors to Minigraph's alignment module to compute base-to-base alignments [42].

6 Experiments

Benchmark datasets: We built string-labeled DAGs of varying sizes by using Minigraph v0.19 [22]. Each DAG is built by using a subset of 95 publicly available haplotype-resolved human genome assemblies [24,30]. In Minigraph, a DAG is iteratively constructed by aligning each haplotype assembly to an intermediate graph, and augmenting additional vertices and/or edges for each structural variant observed. We disabled inversion variants by using `--inv=no` parameter

to avoid introducing cycles in the DAG. CHM13 human genome assembly [30] is used as the starting reference, and we added other haplotype assemblies during DAG construction. In the CHM13 assembly, the first 24 contigs represent individual chromosome (1-22, X, Y) sequences, and the last 25th contig represents mitochondrial DNA. Using this data, we constructed five DAGs, labeled as 1H, 10H, 40H, 80H and 95H respectively. In each of these DAG labels, the integer prefix reflects the count of haplotype assemblies present in the DAG. Properties of these DAGs are shown in Table 1. Parameter K, i.e., the size of MPC, is presented as a range because different connected components in a DAG have different MPCs. For all DAGs, note that the maximum K is $\ll |V|$. We used PBSIM2 v2.0.1 [31] to simulate long reads from CHM13 human assembly. For each simulated read and each DAG, we know the true string path where the read should align. PBSIM2 input parameters were set such that we get sequencing error rate and N50 read length as 5% and 10 kbp respectively. The commands used to run the different tools are listed in Appendix.

Table 1: Properties of DAGs used in our experiments. Total sequence length indicates the sum of length of string labels of all vertices in the DAG.

DAG	**|V|**	**|E|**	**No. of structural variants**	**N50 length of vertex labels (kbp)**	**Total sequence length (Gbp)**	**K (min-max)**
1H	25	0	0	150,617	3.11	1-1
10H	141,755	203,160	61,430	225	3.15	1-9
40H	340,451	489,612	149,186	126	3.23	1-20
80H	553,271	797,528	244,282	85	3.31	1-29
95H	611,949	882,739	270,815	78	3.34	1-35

Evaluation methodology: Alignment output of a read specifies the string path in the input DAG against which the read is aligned. An appropriate evaluation criteria is needed to classify the reported string path as either correct or incorrect by comparing it to the true path. We followed a similar criteria that was used in previous studies [21,22]. First, the reported string path should include only those vertices which correspond to CHM13 assembly, i.e., it should not span an edge augmented from other haplotypes (Figure 3). Second, the reported interval in CHM13 assembly should overlap with the true interval, and the overlapping length should exceed $\geq 10\%$ length of the union of the true and the reported intervals. A correct alignment should satisfy both the conditions. We use `paftools` [21] which implements this evaluation method. All our experiments were done on AMD EPYC 7742 64-core processor with 1 TB RAM. We used 32 threads to run each aligner because all the tested tools support multi-threading by considering each read independently. Wall clock time and peak memory usage were measured using `/usr/bin/time` Linux command.

Performance comparison with existing algorithms: We compared Minichain (v1.0) to three existing sequence-to-DAG aligners: Minigraph v0.19 [22], GraphAligner

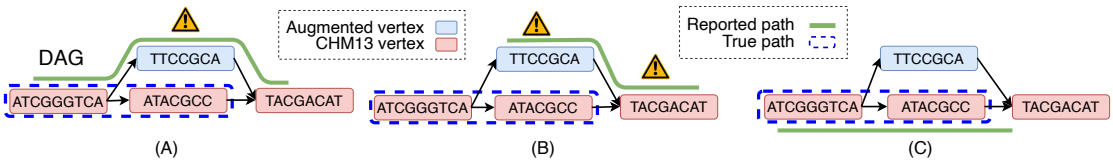


Fig. 3: Illustration of the evaluation criteria. Among the three reported paths above, only (C) is correct.

v1.0.16 [34], and GraphChainer v1.0 [25]. Minigraph uses a two-stage co-linear chaining approach. The first stage ignores edges in the graph and solves co-linear chaining between query sequence and vertex labels. The second stage combines the vertex-specific-chains. In contrast, GraphAligner does not use co-linear chaining and instead relies on its own clustering heuristics. GraphChainer solves co-linear chaining on DAG without penalizing gaps. All the aligners, except GraphChainer, also compute mapping quality (confidence score) for each alignment. We excluded optimal sequence-to-DAG aligners (e.g., [15,17]) because they do not scale to DAGs built by using entire human genomes.

We evaluated accuracy and runtime of Minichain using three DAGs 1H, 10H and 95H (Tables 2, 3, 4). While using DAG 1H, we also tested Minimap2 v2.24 [21], a well-optimized sequence-to-sequence aligner, by aligning reads directly to CHM13 genome assembly. The results show that Minichain consistently achieves the highest precision among the existing sequence-to-DAG aligners. It aligns a higher fraction of reads compared to Minigraph. The gains are also visible when mapping quality (MQ) cutoff 10 is used to filter out low-confidence alignments. GraphAligner and GraphChainer align 100% reads consistently, but this is supplemented with much higher fraction of incorrectly aligned reads. Both Minigraph and Minichain do not align 100% reads. This likely happens because the seeding method used in these two aligners filters out the most frequently occurring minimizers from DAG to avoid processing too many anchors. This can leave several reads originating from long-repetitive genomic regions as unaligned [19].

Among the four aligners, Minigraph performs the best in terms of runtime. Runtime of Minichain increases for DAG 95H because of higher value of K. However, we expect that this can be partly addressed with additional improvements in the proposed chaining algorithm, e.g., by dynamically deleting the anchors from search trees whose gap from all the remaining unprocessed anchors exceeds an acceptable limit. Overall, the results demonstrate practical advantage of Minichain for accurate long-read alignment to DAGs. Superior accuracy of Minichain is also illustrated using precision-recall plots in Figure 4.

Impact of increasing DAG size on accuracy: Alignment accuracy generally deteriorates as count of haplotypes increases in DAGs for all the tested aligners. For each read that was not aligned correctly, we checked if the corresponding reported string path overlaps with the true interval (Figure 3, case A). Such reads are aligned to *correct region* in the DAG but the reported path uses one or more augmented edges. The remaining fraction of incorrectly aligned reads

Table 2: Performance comparison of long read aligners using DAG 1H.

	Minichain	**Minigraph**	**GraphAligner**	**GraphChainer**	**Minimap2**
Indexing time (sec)	**65**	**65**	395	458	55
Alignment time (sec)	205	**104**	6298	6714	133
Memory usage (GB)	20.06	**19.66**	38.12	126.14	12.48
Unaligned reads	0.94%	2.11%	**0**%	**0**%	0%
Incorrect aligned reads	**0.58**%	0.66%	1.06%	1.33%	0.56%
Unaligned reads (MQ≥10)	3.89%	5.82%	0.80%	**0**%	2.29%
Incorrect aligned reads (MQ≥10)	**0.02**%	0.11%	0.53%	1.33%	0.0013%

Table 3: Performance comparison of long read aligners using DAG 10H.

	Minichain	**Minigraph**	**GraphAligner**	**GraphChainer**
Indexing time (sec)	67	**66**	321	537
Alignment time (sec)	610	**132**	5479	9642
Memory usage (GB)	**23.15**	23.16	38.41	143.94
Unaligned reads	1.17%	2.17%	**0**%	**0**%
Incorrect aligned reads	**0.80**%	1.20%	1.55%	2.10%
Unaligned reads (MQ≥10)	4.03%	5.88%	0.28%	**0**%
Incorrect aligned reads (MQ≥10)	**0.20**%	0.34%	0.99%	2.10%

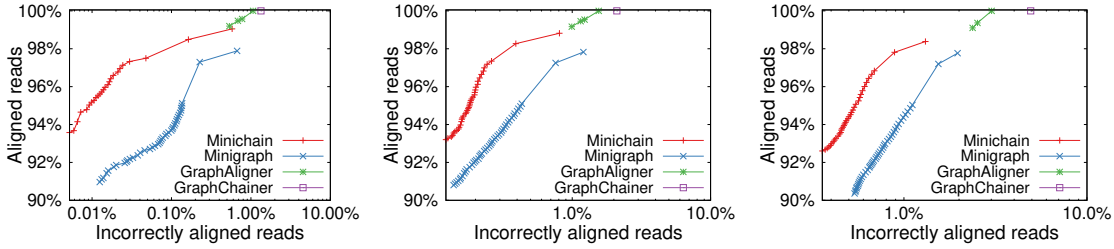
align to *wrong region* in the DAG. We observe that the fraction of incorrectly-aligned reads which align to correct region in DAG increases with increasing count of haplotypes (Figure 5). This happens because the count of alternate paths increases combinatorially with more number of haplotypes which makes precise alignment of a read to its path of origin a challenging problem. Addressing this issue requires further algorithmic improvements.

Table 4: Performance comparison of long read aligners using DAG 95H.

	Minichain	**Minigraph**	**GraphAligner**	**GraphChainer**
Indexing time (sec)	77	**71**	342	763
Alignment time (sec)	1414	**154**	5695	17336
Memory usage (GB)	**24.75**	24.76	40.79	192.36
Unaligned reads	1.62%	2.23%	**0**%	**0**%
Incorrect aligned reads	**1.31**%	1.96%	3.01%	4.92%
Unaligned reads (MQ≥10)	4.75%	6.26%	0.88%	**0**%
Incorrect aligned reads (MQ≥10)	**0.56**%	0.89%	2.38%	4.92%

Acknowledgements

This work was supported by funding from the National Supercomputing Mission, India under DST/NSM/ R&D_HPC_Applications. We used computing resources provided by the C-DAC National PARAM Supercomputing Facility, India, and the National Energy Research Scientific Computing Center, USA.



(a) DAG 1H (b) DAG 10H (c) DAG 95H

Fig. 4: Precision-recall curves obtained by using different aligners. X-axis indicates percentage of incorrectly aligned reads in log-scale. These curves are obtained by setting different mapping quality cutoffs $\in [0, 60]$. GraphChainer curve is a single point because it reports fixed mapping quality 60 in all alignments.

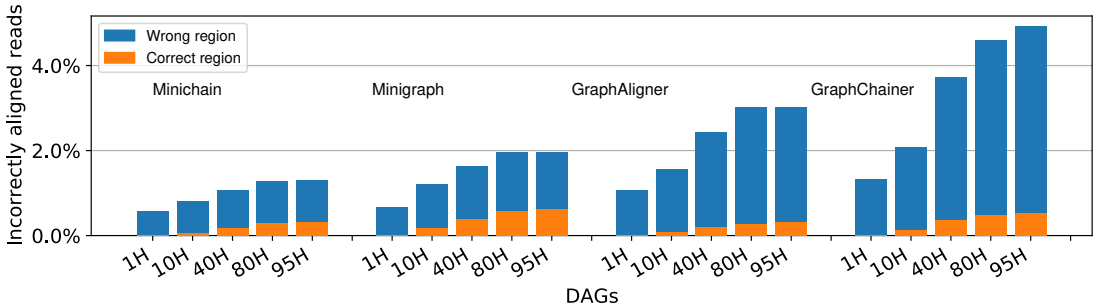


Fig. 5: The fraction of incorrectly aligned reads is shown using DAGs 1H, 10H, 40H, 80H and 95H. Each incorrectly-aligned read is further classified as aligned to either a wrong or a correct region in the DAG based on whether the reported string path overlaps with the true string path (e.g., cases A,B in Figure 3).

References

1. Abouelhoda, M., Ohlebusch, E.: Chaining algorithms for multiple genome comparison. Journal of Discrete Algorithms **3**(2-4), 321–341 (2005)
2. Baaijens, J.A., Bonizzoni, P., Boucher, C., Della Vedova, G., Pirola, Y., Rizzi, R., Sirén, J.: Computational graph pangenomics: a tutorial on data structures and their applications. Natural Computing pp. 1–28 (2022)
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 symposium on Theory of computing. 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)
5. Cáceres, M., Cairo, M., Mumey, B., Rizzi, R., Tomescu, A.I.: Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time. In: Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 359–376. SIAM (2022)
6. Computational Pan-Genomics Consortium: Computational pan-genomics: status, promises and challenges. Briefings in bioinformatics **19**(1), 118–135 (2018)

7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2022)
8. Dvorkina, T., Antipov, D., Korobeynikov, A., Nurk, S.: Spaligner: alignment of long diverged molecular sequences to assembly graphs. BMC bioinformatics **21**(12), 1–14 (2020)
9. Eggertsson, H.P., Jonsson, H., Kristmundsdottir, S., et al.: Graphtyper enables population-scale genotyping using pangenome graphs. Nature genetics **49**(11), 1654–1660 (2017)
10. Eizenga, J.M., Novak, A.M., Sibbesen, J.A., Heumos, S., Ghaffaari, A., Hickey, G., Chang, X., Seaman, J.D., Rounthwaite, R., Ebler, J., et al.: Pangenome graphs. Annual review of genomics and human genetics **21**, 139 (2020)
11. Eppstein, D., Galil, Z., Giancarlo, R., Italiano, G.F.: Sparse dynamic programming i: linear cost functions. Journal of the ACM **39**(3), 519–545 (1992)
12. Eppstein, D., Galil, Z., Giancarlo, R., Italiano, G.F.: Sparse dynamic programming ii: convex and concave cost functions. Journal of the ACM **39**(3), 546–567 (1992)
13. Garg, S., Rautiainen, M., Novak, A.M., et al.: A graph-based approach to diploid genome assembly. Bioinformatics **34**(13), i105–i114 (2018)
14. Illumina: DRAGEN v3.10.4 software release notes. `https://support.illumina.com/content/dam/illumina-support/documents/downloads/software/dragen/200016065_00_DRAGEN-3.10-Customer-Release-Notes.pdf`, accessed: 2022-08-08
15. Ivanov, P., Bichsel, B., Vechev, M.: Fast and optimal sequence-to-graph alignment guided by seeds. In: International Conference on Research in Computational Molecular Biology. pp. 306–325. Springer (2022)
16. Jain, C., Gibney, D., Thankachan, S.V.: Co-linear chaining with overlaps and gap costs. In: International Conference on Research in Computational Molecular Biology (RECOMB). pp. 246–262. Springer (2022)
17. Jain, C., Misra, S., Zhang, H., Dilthey, A., Aluru, S.: Accelerating sequence alignment to graphs. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 451–461. IEEE (2019)
18. Jain, C., Rhie, A., Hansen, N.F., Koren, S., Phillippy, A.M.: Long-read mapping to repetitive reference sequences using winnowmap2. Nature Methods pp. 1–6 (2022)
19. Jain, C., Rhie, A., Zhang, H., Chu, C., Walenz, B.P., Koren, S., Phillippy, A.M.: Weighted minimizer sampling improves long read mapping. Bioinformatics **36**(Supplement_1), i111–i118 (2020)
20. Jain, C., Zhang, H., Dilthey, A., Aluru, S.: Validating paired-end read alignments in sequence graphs. In: 19th International Workshop on Algorithms in Bioinformatics (WABI 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
21. Li, H.: Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics **34**(18), 3094–3100 (may 2018). https://doi.org/10.1093/bioinformatics/bty191
22. Li, H., Feng, X., Chu, C.: The design and construction of reference pangenome graphs with minigraph. Genome Biology **21**(1) (oct 2020)
23. Li, H., Ruan, J., Durbin, R.: Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome research **18**(11), 1851–1858 (2008)
24. Liao, W.W., Asri, M., Ebler, J., Doerr, D., Haukness, M., Hickey, G., Lu, S., Lucas, J.K., Monlong, J., Abel, H.J., et al.: A draft human pangenome reference. bioRxiv (2022). https://doi.org/10.1101/2022.07.09.499321
25. Ma, J., Cáceres, M., Salmela, L., Mäkinen, V., Tomescu, A.I.: Graphchainer: Co-linear chaining for accurate alignment of long reads to variation graphs. bioRxiv (2022)

26. Mäkinen, V., Sahlin, K.: Chaining with overlaps revisited. In: 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)
27. Mäkinen, V., Tomescu, A.I., Kuosmanen, A., Paavilainen, T., Gagie, T., Chikhi, R.: Sparse dynamic programming on DAGs with small width. ACM Transactions on Algorithms **15**(2), 1–21 (Apr 2019)
28. Myers, G., Miller, W.: Chaining multiple-alignment fragments in sub-quadratic time. In: SODA. vol. 95, pp. 38–47 (1995)
29. Navarro, G.: Improved approximate pattern matching on hypertext. Theoretical Computer Science **237**(1-2), 455–463 (2000)
30. Nurk, S., Koren, S., Rhie, A., Rautiainen, M., et al.: The complete sequence of a human genome. Science **376**(6588), 44–53 (apr 2022). https://doi.org/10.1126/science.abj6987
31. Ono, Y., Asai, K., Hamada, M.: PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores. Bioinformatics **37**(5), 589–595 (sep 2020). https://doi.org/10.1093/bioinformatics/btaa835
32. 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)
33. Paten, B., Novak, A.M., Eizenga, J.M., Garrison, E.: Genome graphs and the evolution of genome inference. Genome research **27**(5), 665–676 (2017)
34. Rautiainen, M., Marschall, T.: Graphaligner: rapid and versatile sequence-to-graph alignment. Genome biology **21**(1), 1–28 (2020)
35. Ren, J., Chaisson, M.J.: lra: A long read aligner for sequences and contigs. PLOS Computational Biology **17**(6), e1009078 (2021)
36. Roberts, M., Hayes, W., Hunt, B.R., Mount, S.M., Yorke, J.A.: Reducing storage requirements for biological sequence comparison. Bioinformatics **20**(18), 3363–3369 (jul 2004). https://doi.org/10.1093/bioinformatics/bth408
37. Sahlin, K., Baudeau, T., Cazaux, B., Marchet, C.: A survey of mapping algorithms in the long-reads era. bioRxiv (2022)
38. Sahlin, K., Mäkinen, V.: Accurate spliced alignment of long RNA sequencing reads. Bioinformatics **37**(24), 4643–4651 (2021)
39. Salmela, L., Rivals, E.: Lordec: accurate and efficient long read error correction. Bioinformatics **30**(24), 3506–3514 (2014)
40. Sirén, J., Monlong, J., Chang, X., et al.: Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. Science **374**(6574), abg8871 (2021)
41. Wang, T., Antonacci-Fulton, L., Howe, K., et al.: The human pangenome project: a global resource to map genomic diversity. Nature **604**(7906), 437–446 (apr 2022)
42. Zhang, H., Wu, S., Aluru, S., Li, H.: Fast sequence to graph alignment using the graph wavefront algorithm. arXiv preprint arXiv:2206.13574 (2022)

Appendix

1 Algorithms for solving Problems 2b and 2c

To support Lemma 5, we propose $O(KN \log^2 N + KN \log KN)$ time algorithms to solve Problems 2b and 2c. We assume that DAG $G(V, E, \sigma)$ is preprocessed already (Lemma 3). Unlike Algorithm 2 which uses 1D orthogonal range queries, here we will use 2D orthogonal range queries. When function $f(gap_G(s_{j-1}, s_j), gap_S(s_{j-1}, s_j))$ for gap cost is defined as either $\max(gap_G(s_{j-1}, s_j), gap_S(s_{j-1}, s_j))$ or $|gap_G(s_{j-1}, s_j) - gap_S(s_{j-1}, s_j)|$, the second dimension is used to check whether $gap_G(s_{j-1}, s_j) \geq gap_S(s_{j-1}, s_j)$ holds or not. The following search tree data structure is used to support 2D orthogonal range queries (ref. [4]).

Lemma 6. *Let n be the number of $(key, value)$ entries where a key is defined as a tuple (k_1, k_2) of type $\mathbb{Z} \times \mathbb{Z}$. The following operations can be supported in $O(\log^2 n)$ time:*

- *update$((k_1, k_2), val)$: For the entry w with $key = (k_1, k_2)$, $value(w) \leftarrow \max(value(w), val)$.*
- *RMQ$((l_1, r_1), (l_2, r_2))$: Return $\max\{value(w) \mid l_1 < key(w).k_1 < r_1,\ l_2 < key(w).k_2 < r_2\}$. This is 2-dimensional range maximum query. Input ranges can be provided as either open or closed intervals.*

Given n $(key, value)$ entries, a search tree data structure to support the above operations can be constructed in $O(n \log n)$ time and space.

Algorithm 3 outlines our solution to Problem 2b. The pseudocode follows a similar structure as Algorithm 2. We use $2K$ search trees $\mathcal{T}_i$ and $\mathcal{I}_i$, for all $i \in [1, K]$ (Line 1). Search trees $\mathcal{T}_i$ execute update operations to handle those cases where gap cost should equal $gap_S(s_{j-1}, s_j)$ (Line 13). In Line 8, the first dimension of the range query in $\mathcal{T}_i$ is to restrict search over those anchors which precede anchor $M[j]$ (same as Algorithm 2), and the second dimension is to further restrict search over those preceding anchors $M[i]$ for which $gap_S(M[i], M[j]) \geq gap_G(M[i], M[j])$. Similarly, search trees $\mathcal{I}_i$ set value of each key to handle cases where gap cost equals $gap_G(s_{j-1}, s_j)$ (Line 14). In Line 9, the first dimension of the range query in $\mathcal{I}_i$ is to restrict search over those anchors which precede anchor $M[j]$, and the second dimension further restricts search over those anchors $M[i]$ for which $gap_S(M[i], M[j]) \leq gap_G(M[i], M[j])$. Throughout the algorithm, we perform $O(KN)$ RMQ and update operations. Each operation uses $O(log^2 N)$ time because size of each tree is at most N. Therefore, these operations require overall $O(KN \log^2 N)$ time. The sorting operation used in Line 4 uses $O(KN \log KN)$ time to sort $O(KN)$ tuples. The combined storage complexity of $2K$ search trees is $O(KN \log N)$. This completes the description of Algorithm 3. Algorithm 4 addresses Problem 2c, and follows a similar intuition.

Algorithm 3: $O(KN \log^2 N + KN \log KN)$ time co-linear chaining algorithm to solve Problem 2b

Input: Array of weighted anchors $M[1..N]$, preprocessed DAG $G(V, E, \sigma)$
Output: Array $C[1..N]$ such that $C[j]$ = score of an optimal chain that ends at $M[j]$

1 Initialize search trees $\mathcal{T}_i$ and $\mathcal{I}_i$, for all $i \in [1, K]$, using keys $\{(M[j].d, dist2begin(M[j].v, i) + M[j].y - M[j].d) \mid 1 \leq j \leq N \text{ and } i \in paths(M[j].v)\}$ and values $-\infty$
2 Initialize $C[j]$ as $weight(M[j])$, for all $j \in [1, N]$
3 Build array Z as described in Algorithm 2 (Lines 4-13)
4 **for** $z \in Z$ *in lexicographically ascending order based on the key* $(rank(v), pos, task)$ **do**
5 $\quad j \leftarrow z.anchor, i \leftarrow z.path, v \leftarrow z.v$
6 $\quad$ **if** $z.task = 0$ **then**
7 $\quad\quad p \leftarrow dist2begin(v, i) + D(v, M[j].v) + M[j].x$
8 $\quad\quad q \leftarrow \mathcal{T}_i.\mathsf{RMQ}((0, M[j].c), [p - M[j].c, +\infty)) - (M[j].c - 1)$
9 $\quad\quad r \leftarrow \mathcal{I}_i.\mathsf{RMQ}((0, M[j].c), (-\infty, p - M[j].c]) - (p - 1)$
10 $\quad\quad C[j] \leftarrow \max(C[j],\ weight(M[j]) + q,\ weight(M[j]) + r)$
11 $\quad$ **else**
12 $\quad\quad \mathsf{s} \leftarrow dist2begin(v, i) + M[j].y$
13 $\quad\quad \mathcal{T}_i.\mathsf{update}((M[j].d, s - M[j].d), C[j] + M[j].d)$
14 $\quad\quad \mathcal{I}_i.\mathsf{update}((M[j].d, s - M[j].d), C[j] + s)$
15 **end**

Algorithm 4: $O(KN \log^2 N + KN \log KN)$ time co-linear chaining algorithm to solve Problem 2c

Input: Array of weighted anchors $M[1..N]$, preprocessed DAG $G(V, E, \sigma)$
Output: Array $C[1..N]$ such that $C[j]$ = score of an optimal chain that ends at $M[j]$

1 Initialize search trees $\mathcal{T}_i$ and $\mathcal{I}_i$ as described in Algorithm 3 in Line 1
2 Initialize $C[j]$ as $weight(M[j])$, for all $j \in [1, N]$
3 Build array Z as described in Algorithm 2 (Lines 4-13)
4 **for** $z \in Z$ *in lexicographically ascending order based on the key* $(rank(v), pos, task)$ **do**
5 $\quad j \leftarrow z.anchor, i \leftarrow z.path, v \leftarrow z.v$
6 $\quad$ **if** $z.task = 0$ **then**
7 $\quad\quad p \leftarrow dist2begin(v, i) + D(v, M[j].v) + M[j].x$
8 $\quad\quad q \leftarrow \mathcal{T}_i.\mathsf{RMQ}((0, M[j].c), [p - M[j].c, +\infty)) - (M[j].c - p)$
9 $\quad\quad r \leftarrow \mathcal{I}_i.\mathsf{RMQ}((0, M[j].c), (-\infty, p - M[j].c]) - (p - M[j].c)$
10 $\quad\quad C[j] \leftarrow \max(C[j],\ weight(M[j]) + q,\ weight(M[j]) + r)$
11 $\quad$ **else**
12 $\quad\quad \mathsf{s} \leftarrow dist2begin(v, i) + M[j].y$
13 $\quad\quad \mathcal{T}_i.\mathsf{update}((M[j].d, s - M[j].d), C[j] + M[j].d - s)$
14 $\quad\quad \mathcal{I}_i.\mathsf{update}((M[j].d, s - M[j].d), C[j] + s - M[j].d)$
15 **end**

2 Commands used for empirical evaluation

Table 5: Command-line arguments used to run various tools. The scripts to reproduce results are available in Minichain GitHub repository.

Purpose	**Tool**	**Commands**
Read simulation	pbsim2 (v2.0.1, bioconda) paftools.js (commit:15cade0)	pbsim --depth 0.5 --length-min 10000 --length-mean 10000 --length-max 10000 --accuracy-mean 0.95 --accuracy-max 0.95 --accuracy-min 0.95 --hmm_model P6C4.model CHM13Y.fa k8-Linux paftools.js pbsim2fq CHM13Y.fa.fai *.maf > CHM13Y_reads.fa
Graph generation	minigraph (v0.19-r551)	minigraph -t32 -cxggs --inv=no CHM13Y.fa HG002.1.fa HG002.2.fa HG00438.1.fa HG00438.2.fa HG005.1.fa HG005.2.fa HG00621.1.fa HG00621.2.fa HG00673.1.fa > CHM13Y_10H.gfa 2> log_10H.txt
Read mapping	minimap2 (v2.24-r1122) minigraph (v0.19-r551) minichain (v1.0) GraphAligner (commit:4e2ca66) GraphChainer (commit:59c9c67)	minimap2 -t32 -cx map-pb CHM13Y.fa CHM13Y_reads.fa > CHM13Y.paf minigraph -t32 -cx lr CHM13Y_10H.gfa CHM13Y_reads.fa > CHM13Y_10H.gaf minichain -t32 -cx lr CHM13Y_10H.gfa CHM13Y_reads.fa > CHM13Y_10H.gaf GraphAligner -t32 -x vg -g CHM13Y_10H.gfa -f CHM13Y_reads.fa -a CHM13Y_10H.gaf GraphChainer -t32 -g CHM13Y_10H.gfa -f CHM13Y_reads.fa -a CHM13Y_10H.gaf
Conversion to stable GAF co-ordinates (GraphAligner and GraphChainer)	mgutils.js (commit:8619249)	k8-Linux mgutils.js stableGaf CHM13Y_10H.gfa CHM13Y_10H.gaf > CHM13Y_10H_stable.gaf
Evaluation of read alignments	paftools.js	k8-Linux paftools.js mapeval CHM13Y_10H.gaf