Co-Linear Chaining on Pangenome Graphs

Jyotshna Rajput
Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India

Ghanshyam Chandra
Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India

Chirag Jain
Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India

Abstract

Pangenome reference graphs are useful in genomics because they compactly represent the genetic diversity within a species, a capability that linear references lack. However, efficiently aligning sequences to these graphs with complex topology and cycles can be challenging. The seed-chain-extend based alignment algorithms use co-linear chaining as a standard technique to identify a good cluster of exact seed matches that can be combined to form an alignment. Recent works show how the co-linear chaining problem can be efficiently solved for acyclic pangenome graphs by exploiting their small width [Makinen et al., TALG'19] and how incorporating gap cost in the scoring function improves alignment accuracy [Chandra and Jain, RECOMB'23]. However, it remains open on how to effectively generalize these techniques for general pangenome graphs which contain cycles. Here we present the first practical formulation and an exact algorithm for co-linear chaining on cyclic pangenome graphs. We rigorously prove the correctness and computational complexity of the proposed algorithm. We evaluate the empirical performance of our algorithm by aligning simulated long reads from the human genome to a cyclic pangenome graph constructed from 95 publicly available haplotype-resolved human genome assemblies. While the existing heuristic-based algorithms are faster, the proposed algorithm provides a significant advantage in terms of accuracy.

2012 ACM Subject Classification Applied computing → Computational genomics

Keywords and phrases Sequence alignment, variation graph, genome sequencing, path cover

Digital Object Identifier 10.4230/LIPIcs.WABI.2023.12

Supplementary Material *Software (Source Code)*: `https://github.com/at-cg/PanAligner`

Funding This research is supported in part by the funding from National Supercomputing Mission, India under DST/NSM/ R&D_HPC_Applications and the Science and Engineering Research Board (SERB) under SRG/2021/000044. We used computing resources provided by the National Energy Research Scientific Computing Center (NERSC), USA.

Acknowledgements The authors thank Manuel Cáceres, Shravan Mehra and Sunil Chandran for providing useful feedback.

1 Introduction

Graph-based representation of genome sequences has emerged as a prominent data structure in genomics, offering a powerful means to represent the genetic variation within a species across multiple individuals [11, 17, 26, 49, 51, 53]. A pangenome graph can be represented as a directed graph $G(V, E)$ such that vertices are labeled by characters (or strings) from the alphabet {A,C,G,T}. The topology of the graph is determined by the count and the type of variants included in the graph. For example, inversions, duplications, or copy number variation are best represented as cycles in a pangenome graph [8, 26, 27, 41, 49]. As a result, the draft pangenome graphs published by the Human Pangenome Reference Consortium [26] and the Chinese Pangenome Consortium [14] are also cyclic. Aligning reads or assembly

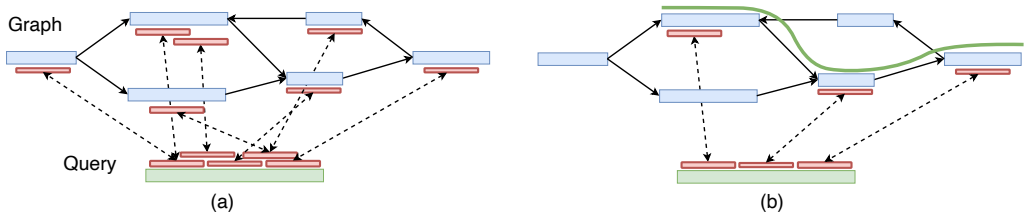


Figure 1 An illustration of co-linear chaining for sequence-to-graph alignment. Assume that the vertices of the graph are labeled with nucleotide sequences. Short exact matches, i.e., anchors, are illustrated using red blocks joined by dotted lines. In (b), the anchors corresponding to the best-scoring chain are retained, and the rest are removed. The retained anchors are combined to produce an alignment of the query sequence to the graph.

contigs to a directed labeled graph is a fundamental problem in computational pangenomics [2, 7]. Aligning reads to graphs is also useful for other bioinformatics tasks such as long-read *de novo* assembly [6, 15, 43] and long-read error correction [28, 47].

Formally, the sequence-to-graph alignment problem seeks a walk in the graph that spells a sequence with minimum edit distance from the input sequence. $O(|Q||E|)$ time alignment algorithms for this problem are already known, where Q is the query sequence [22, 36]. The known conditional lower bound [3] implies that an exact algorithm significantly faster than $O(|Q||E|)$ is unlikely. This lower bound also holds for de Bruijn graphs [18]. Therefore, fast heuristics are used to process high-throughput sequencing data.

Seed-chain-extend is a common heuristic used by modern alignment tools [21, 23, 46]. This is a three-step process. First, the seeding stage involves computing exact seed matches, such as k-mer matches, between a query sequence and a reference. These matches are referred to as *anchors*. The presence of repetitive sequences in genomes often leads to a large number of false positive anchors. Subsequently, the *chaining* stage is employed to link the subsets of anchors in a coherent manner while optimizing specific criteria (Figure 1). This procedure also eliminates the false positive anchors. Finally, the extend stage returns a base-to-base alignment along the selected anchors. Efficient generalization of the three stages to pangenome graphs is an active research topic [7]. Many sequence-to-graph aligners already exist that differ in terms of implementing these stages [5, 9, 24, 30, 42, 49]. This paper addresses the generalization of the chaining stage to cyclic pangenome graphs.

1.1 Related Work

Co-linear chaining is a mathematically rigorous method to filter anchors after the seeding stage. It has been well-studied for the sequence-to-sequence alignment case [1, 12, 13, 20, 32, 35, 40]. The input to the chaining problem is a set of N weighted anchors. An anchor can be denoted as a pair of intervals in the two sequences corresponding to the exact seed match. A chain is an ordered subset of anchors whose intervals must appear in increasing order in both sequences. The co-linear chaining problem seeks the chain with the highest score, where the score of a chain is calculated by summing the weights of the anchors in the chain and subtracting the penalty for gaps between adjacent anchors. The problem is solvable in $O(N \log N)$ time [1].

The first effort to generalize the co-linear chaining problem to graphs was made by Makinen et al. [33]. They addressed the co-linear chaining problem on directed acyclic graphs (DAGs). The authors introduced a sparse dynamic programming algorithm whose runtime complexity is parameterized in terms of the *width* of the DAG. The width of a DAG

is defined as the minimum number of paths in the DAG such that each vertex is included in at least one path. Parameterizing the complexity in terms of the width is helpful because pangenome graphs typically have small width in practice [5, 30, 33]. An optimized version of their algorithm requires $O(KN \log KN)$ time for chaining, where K is the width of the DAG [30]. This formulation has been further extended to incorporate gap cost in the scoring function [5], and for solving the longest common subsequence problem between a DAG and a sequence [44]. However, there is limited work on formulating and solving the co-linear chaining problem for general pangenome graphs which might contain cycles. One way to address this was discussed in [30, Appendix section], but the proposed formulation is oblivious to the coordinates of anchors that lie in a strongly connected component of the graph. Their algorithm works by shrinking every strongly connected component into a single vertex and applying the same algorithm developed for DAGs. With this approach, the high-scoring anchor chains in cyclic regions of the graph may result in low-quality alignments.

1.2 Contributions

In this paper, we build on top of the algorithmic techniques developed for DAGs [5, 30, 33] and propose novel formulations for cyclic pangenome graphs. Our proposed algorithm exploits the small width of pangenome graphs similar to [33]. Our approach for defining the gap cost between a pair of anchors is inspired by the corresponding function defined on DAGs [5].

We address the following three challenges that arise on cyclic pangenome graphs. First, the dynamic programming-based chaining algorithms developed for DAGs exploit the topological ordering of vertices [5, 30, 33], but such an ordering is not available in cyclic graphs. Second, computing the width and a minimum path cover can be solved in polynomial time for DAGs but is NP-hard for general instances [4]. Third, the walk corresponding to the optimal sequence-to-graph alignment can traverse a vertex multiple times if there are cycles. Accordingly, a chain of anchors should be allowed to loop through vertices. Our proposed problem formulation and the proposed algorithm address the above challenges. Our approach involves computing a path cover $\mathcal{P}$ of the input graph followed by using iterative algorithms. Let $\Gamma_c, \Gamma_l, \Gamma_d$ be the parameters that specify the count of iterations used in our algorithms (formally defined later). Our chaining algorithm solves the stated objective in $O(\Gamma_c|\mathcal{P}|N \log N + |\mathcal{P}|N \log |\mathcal{P}|N)$ time after a one-time preprocessing of the graph in $O((\Gamma_l + \Gamma_d + \log |V|)|\mathcal{P}||E|)$ time. We will show that parameters $|\mathcal{P}|, \Gamma_c, \Gamma_l, \Gamma_d$ are small in practice to justify the practicality of this approach.

We implemented the proposed chaining algorithm as an open-source software PanAligner. We designed PanAligner as an end-to-end sequence-to-graph aligner using seeding and alignment code from Minigraph [24]. We evaluated the scalability and alignment accuracy of PanAligner by using a cyclic human pangenome graph constructed from 94 high-quality haplotype-resolved assemblies [26] and CHM13 human genome assembly [38]. We achieve the highest long-read mapping accuracy 98.7% using PanAligner when compared to existing methods Minigraph [24] (98.1%) and GraphAligner [42] (97.0%).

2 Notations and Problem Formulations

Pangenome graph $G(V, E, \sigma)$ is a string labeled graph such that function $\sigma : V \to \Sigma^+$ labels each vertex v with string $\sigma(v)$ over alphabet $\Sigma = \{A, C, G, T\}$. Let Q be a query sequence over Σ. Let $M[1..N]$ be an array of anchor tuples $(v, [x..y], [c..d])$ with the interpretation that substring $\sigma(v)[x..y]$ from the graph matches substring $Q[c..d]$ in the query sequence. Throughout this paper, all indices start at 1. We will assume that $|E| \geq |V| - 1$. Function *weight* assigns a user-specified weight to each anchor. For example, the weight of an anchor could be proportional to the length of the matching substring.

A path cover is a set $\mathcal{P} = \{P_1, P_2, \ldots, P_{|\mathcal{P}|}\}$ of paths in graph G such that every vertex in V is included in at least one of the $|\mathcal{P}|$ paths. We define $paths(v)$ as $\{i : P_i \text{ includes } v\}$. If $i \in paths(v)$, then let $index(v, i)$ specify the position of vertex v on path P_i. Suppose $\mathcal{R}^-(v)$ is the set of vertices in V that can reach vertex v through any walk in graph G. We will assume that the set $\mathcal{R}^-(v)$ always includes the vertex v. The value $last2reach(v, i)$ for $v \in V, i \in [1, |\mathcal{P}|]$ represents the last vertex on path P_i that belongs to set $\mathcal{R}^-(v)$. Note that $last2reach(v, i)$ does not exist if there is no vertex on path P_i that belongs to $\mathcal{R}^-(v)$. Let $N^+(v)$ and $N^-(v)$ be the set of outgoing and incoming neighbor vertices of vertex v, respectively.

We need to calculate character distances between pairs of anchors in the graph while solving the co-linear chaining problem. Assume that edge $(v, u) \in E$ has length $|\sigma(v)| > 0$. Let $D(v_1, v_2)$ denote the length of the shortest path from vertex v_1 to v_2 in G. We set $D(v_1, v_2) = \infty$ if there is no path from v_1 to v_2, whereas $D(v_1, v_2) = 0$ if $v_1 = v_2$. We use $D^\circ(v)$ to specify the length of the shortest proper cycle containing v. $D^\circ(v) = \infty$ if v is not part of any proper cycle. If P_i includes v, let $dist2begin(v, i)$ denote the length of the sub-path of path P_i from the start of P_i to v.

Our algorithm will use a balanced binary search tree data structure for executing range queries efficiently. It has the following properties.

▶ **Lemma 1** (ref. [31]). *Let n be the number of leaves in a tree, each storing a $(key, value)$ pair. The following operations can be supported in $O(\log n)$ time:*

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

Given n $(key, value)$ pairs, the tree can be constructed in $O(n \log n)$ time and $O(n)$ space.

Next, we define a precedence relation between a pair of anchors, which is a partial order among the input anchors [30].

▶ **Definition 2** (Precedence). *Given two anchors $M[i]$ and $M[j]$, we define $M[i]$ precedes $(\prec)$ $M[j]$ as follows. If $M[i].v \neq M[j].v$, then $M[i] \prec M[j]$ if and only if $M[i].d < M[j].c$ and $M[i].v$ reaches $M[j].v$. If $M[i].v = M[j].v$, then $M[i] \prec M[j]$ if and only if $M[i].d < M[j].c$, and $M[i].y < M[j].x$ or the graph has a proper cycle containing $M[i].v$.*

▶ **Definition 3** (Chain). *Given the set of anchors $\{M[1], M[2], \ldots, M[N]\}$, a chain is an ordered subset of anchors $S = s_1 s_2 \cdots s_q$ of M, such that s_j precedes s_{j+1} for all $1 \leq j < q$.*

Our co-linear chaining problem formulation seeks a chain $S = s_1 s_2 \cdots s_q$ that maximizes the chain score defined as $\sum_{j=1}^{q} weight(s_j) - \left(\sum_{j=1}^{q-1} gap_Q(s_j, s_{j+1}) + \sum_{j=1}^{q-1} gap_G(s_j, s_{j+1})\right)$. Functions gap_Q and gap_G specify the gap cost incurred in the query sequence and the graph, respectively. Although we specifically focus on problem formulations where the gap cost is calculated as the sum of gap_G and gap_Q, our approach can be extended to other gap definitions such as $|gap_G - gap_Q|$, $\min(gap_G, gap_Q)$, or $\max(gap_G, gap_Q)$, similar to [5]. We define $gap_Q(s_j, s_{j+1})$ as $s_{j+1}.c - s_j.d - 1$, which can be interpreted as the count of characters in the query sequence between the endpoints of the two anchors. Next, we will define two versions of the co-linear chaining problem that differ in their definition of gap_G. In both versions, $gap_G(s_j, s_{j+1})$ is calculated by looking at the count of characters spelled along a walk in the graph from s_j to s_{j+1}. In the first version of the problem formulation, we use the shortest path from vertex $s_j.v$ to $s_{j+1}.v$ to calculate $gap_G(s_j, s_{j+1})$.

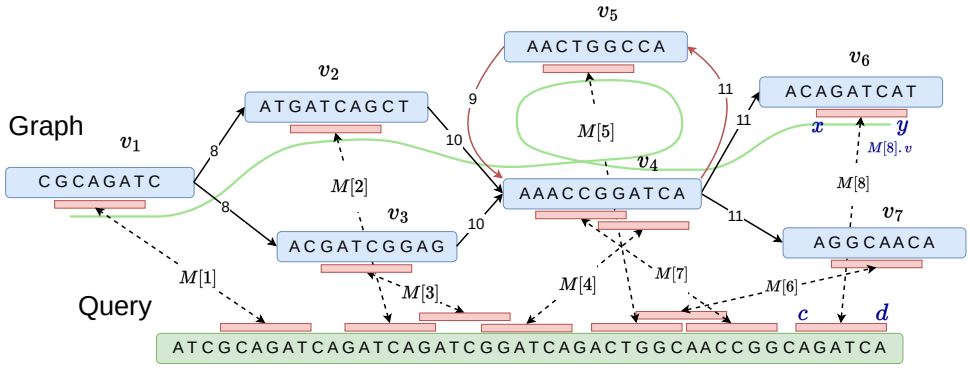


Figure 2 An example illustrating a graph, a query sequence, and multiple anchors as input for co-linear chaining. The sequence of anchors $(M[1], M[2], M[4], M[5], M[7], M[8])$ forms a valid chain that visits vertex v_4 twice due to a cycle in the graph. The coordinates associated with anchor $M[8]$ are also highlighted as an example.

▶ **Problem 4.** *Given a query sequence Q, graph $G(V, E, \sigma)$ and anchors $M[1..N]$, determine the optimal chaining score by using the following definition of gap_G:*

$$gap_G(s_j, s_{j+1}) = \begin{cases} s_{j+1}.x - s_j.y - 1 + D(s_j.v, s_{j+1}.v) & s_{j+1}.v \neq s_j.v \\ s_{j+1}.x - s_j.y - 1 & s_j.v = s_{j+1}.v \text{ and } s_j.y < s_{j+1}.x \\ s_{j+1}.x - s_j.y - 1 + D^{\circ}(s_j.v) & s_j.v = s_{j+1}.v \text{ and } s_j.y \geq s_{j+1}.x, \end{cases}$$

where (s_j, s_{j+1}) is a pair of anchors from M such that s_j precedes s_{j+1}.

▶ **Lemma 5.** *Problem 4 can be solved in $\Theta(|V||E| + |V|^2 \log |V| + N^2)$ time.*

Proof. Compute the shortest distance $D(v_i, v_j)$ between all pairs of vertices $v_i, v_j \in V$ in $O(|V||E| + |V|^2 \log |V|)$ time by using Dijkstra's algorithm from every vertex. Next, compute $D^{\circ}(v)$ as $\min_{u \in N^+(v)} |\sigma(v)| + D(u, v)$ in $\Theta(|E|)$ time for all $v \in V$. These computations need to be done only once for a graph. To solve the chaining problem for a given query sequence, sort the input anchor array $M[1..N]$ in non-decreasing order by the component $M[\cdot].c$. Let $C[1..N]$ be a one-dimensional table in which $C[j]$ will be the optimal score of a chain ending at anchor $M[j]$. Initialize $C[j]$ as $weight(M[j])$ for all $j \in [1, N]$. Subsequently, compute C in the left-to-right order by using the recursion $C[j] = \max_{M[i] \prec M[j]} \{C[j], weight(M[j]) - gap_Q(M[i], M[j]) - gap_G(M[i], M[j])\}$. Computing $C[j]$ takes $\Theta(N)$ time because precedence condition can be checked in constant time. Report $\max_j C[j]$ as the optimal chaining score. ◀

The above algorithm is unlikely to scale to large whole-genome sequencing datasets because it requires $\Theta(N^2)$ time for the dynamic programming recursion. Motivated by [5], we will define an alternative definition of gap_G. We will approximate the distance between a pair of vertices by using a path cover of the graph. We will later propose an efficient algorithm for the revised problem formulation.

Suppose $\mathcal{P} = \{P_1, P_2, \ldots, P_{|\mathcal{P}|}\}$ is a path cover of graph G. Consider a pair of vertices $v_1, v_2 \in V$ such that v_1 reaches v_2. For each path $i \in paths(v_1)$, consider the walk starting from v_1 along the edges of path P_i till vertex α_i, where vertex $\alpha_i = v_2$ if v_2 also lies on path P_i anywhere after v_1, i.e., $index(v_2, i) \geq index(v_1, i)$, and $\alpha_i = last2reach(v_2, i)$ otherwise. If $\alpha_i \neq v_2$, the rest of the walk till v_2 is completed by using the shortest path from vertex α_i to v_2. Denote $D_{\mathcal{P}}(v_1, v_2)$ as the length of the shortest walk among such $|paths(v_1)|$ possible walks from v_1 to v_2. Formally, we can write $D_{\mathcal{P}}(v_1, v_2)$ as

$$\min_{i \in paths(v_1)} dist2begin(\alpha_i, i) - dist2begin(v_1, i) + D(\alpha_i, v_2). \quad (1)$$

$D_{\mathcal{P}}(v_1, v_2)$ is well defined if v_2 is reachable from v_1. We set $D_{\mathcal{P}}(v_1, v_2) = \infty$ if v_2 is not reachable from v_1. Finally, if vertex v is part of a proper cycle in G, we define $D^{\circ}_{\mathcal{P}}(v)$ as the length of a specific walk that starts and ends at v, i.e., $D^{\circ}_{\mathcal{P}}(v)$ as $\min_{u \in N^+(v)} |\sigma(v)| + D_{\mathcal{P}}(u, v)$ for all $v \in V$. $D^{\circ}_{\mathcal{P}}(v) = \infty$ if v is not part of any proper cycle.

▶ **Problem 6.** *Given a query sequence Q, graph $G(V, E, \sigma)$ and anchors $M[1..N]$, determine a path cover $\mathcal{P}$ of the graph, and the optimal chaining score by using the following definition of gap_G:*

$$gap_G(s_j, s_{j+1}) = \begin{cases} s_{j+1}.x - s_j.y - 1 + D_{\mathcal{P}}(s_j.v, s_{j+1}.v) & s_{j+1}.v \neq s_j.v \\ s_{j+1}.x - s_j.y - 1 & s_j.v = s_{j+1}.v \text{ and } s_j.y < s_{j+1}.x \\ s_{j+1}.x - s_j.y - 1 + D^{\circ}_{\mathcal{P}}(s_j.v) & s_j.v = s_{j+1}.v \text{ and } s_j.y \geq s_{j+1}.x, \end{cases}$$

where (s_j, s_{j+1}) is a pair of anchors from M such that s_j precedes s_{j+1}.

3 Proposed Algorithms

A single experiment typically requires aligning millions of reads to a graph. Therefore, we will do a one-time preprocessing of the graph that will help reduce the runtime of our chaining algorithm for solving Problem 6.

3.1 Algorithms for Preprocessing the Graph

We compute the following quantities during the preprocessing stage:

- A path cover $\mathcal{P}$ of $G(V, E, \sigma)$. We require the path cover to be small (in the number of paths). However, determining the minimum path cover in a graph with cycles is an NP-hard problem. We will discuss an efficient heuristic for determining a small path cover.
- A bijective function $rank : V \to [1, |V|]$ that specifies a linear ordering of vertices. The ordering should satisfy the following property: If vertex v_2 occurs anywhere after v_1 in a path in $\mathcal{P}$, then $rank(v_2) > rank(v_1)$ for all $v_1, v_2 \in V$. Such an ordering may not exist for an arbitrary path cover but it will exist for the path cover chosen by us.
- $last2reach(v, i)$, $D(last2reach(v, i), v)$, $dist2begin(v, i)$ and $D^{\circ}_{\mathcal{P}}(v)$ for all $v \in V$ and $i \in [1, |\mathcal{P}|]$. These values will be frequently used by our chaining algorithm to compute gap costs.

We propose the following heuristic for computing a small path cover of graph $G(V, E, \sigma)$. We derive a DAG $G'(V, E', \sigma)$ from G by removing a small number of edges. Next, we determine the minimum path cover $\mathcal{P}$ of G' in $O(|\mathcal{P}||E| \log |V|)$ time by using a known algorithm [33]. Our intuition is that removing as few edges as possible will provide a close to optimal path cover of G. One way to compute G' is to use standard heuristic-based solvers for minimum feedback arc set (FAS) problem, e.g., [10], but we empirically observed that this approach could sometimes disconnect a weak component of a graph, leading to a large path cover. Therefore, instead of using FAS heuristics, we use a simple idea where we identify all strongly connected components [50] and perform a depth-first search within each strong component to remove back edges. This approach provides a DAG that has the same number of weak

components as G while removing a small number of edges in practice, thus resulting in a small path cover. Next, we compute a function $rank$ for all vertices $\in V$ by topological sorting of vertices in DAG G'.

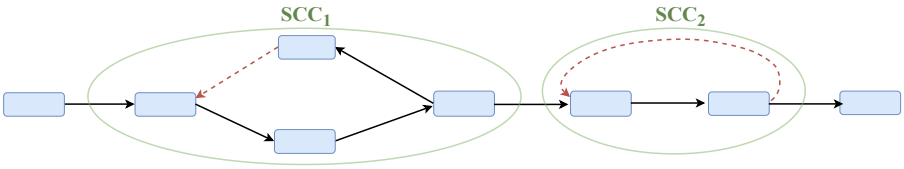


Figure 3 An illustration of the proposed heuristic used to convert a cyclic graph into a DAG. Red-dotted edges represent the removed back edges in each strongly connected component (SCC).

If there is no cycle in G, then $last2reach(v,i)$ and $D(last2reach(v,i),v)$ can be computed in $O(|\mathcal{P}||E|)$ time by using dynamic programming algorithms that process vertices in topological order [5, 33]. We extend these ideas to cyclic graphs by designing iterative algorithms. We will formally prove that as the iterations proceed, the output gets closer to the desired solution. Our approach to computing $last2reach(v,i)$ is outlined in Algorithm 1. If $last2reach(v,i)$ exists, the algorithm determines it in terms of its $rank$. We maintain an array $L2R$ to save intermediate results. $L2R(v,i)$ is initialised to $rank(v)$ if v lies on path P_i. In each iteration, we revise $L2R(v,i)$ by probing $L2R(u,i)$ for all $u \in N^-(v)$. In Lemma 7, we prove the correctness of this algorithm by arguing that all $|\mathcal{P}||V|$ values in array $L2R$ converge to their optimal values through label propagation in $\leq |V|$ iterations. Let Γ_l denote the count of iterations used by the algorithm. $L2R(v,i)$ remains 0 if $last2reach(v,i)$ does not exist.

Algorithm 1 $O(\Gamma_l|\mathcal{P}||E|)$ time algorithm to compute $last2reach(v,i)$ for all $v \in V$ and $i \in [1,|\mathcal{P}|]$.

```
1  Initialize L2R(v,i) to rank(v) if i ∈ paths(v) and 0 otherwise for all v ∈ V and i ∈ [1,|P|]
2  Initialize L2R_prev(v,i) to 0 for all v ∈ V and i ∈ [1,|P|]
3  /*L2R and L2R_prev will hold the values of current and previous iteration respectively*/
4  while ∃v ∈ V, ∃i ∈ [1,|P|], L2R(v,i) ≠ L2R_prev(v,i) do
5      for i ∈ [1,|P|] do
6          for v ∈ V in the increasing order of rank(v) do
7              L2R_prev(v,i) ← L2R(v,i)
8              L2R(v,i) ← max_{u∈N⁻(v)∪v} L2R(u,i)
9          end
10     end
11 end
```

▶ **Lemma 7.** *In Algorithm 1, $L2R(v,i)$ converges to the rank of $last2reach(v,i)$ in at most $|V|$ iterations for all $v \in V$ and $i \in [1,|\mathcal{P}|]$.*

Proof. A vertex $v_2 \in V$ is said to be reachable within k hops from vertex $v_1 \in V$ if there exists a path with $\leq k$ edges from v_1 to v_2. We will prove by induction that Algorithm 1 satisfies the following invariant: After j iterations, $L2R(v,i)$ has converged to $rank(last2reach(v,i))$ if $last2reach(v,i)$ exists and vertex v is reachable within j hops from $last2reach(v,i)$ in G. This argument will prove the lemma because vertex $v_2 \in V$ must be reachable within $|V|-1$ hops from $v_1 \in V$ if v_2 is reachable from v_1. Base case ($j = 0$) holds due to initialisation of $L2R(v,i)$ in Line 1. If v lies 0-hop from $last2reach(v,i)$, i.e., $v = last2reach(v,i)$, then v must lie on path P_i and $rank(last2reach(v,i)) = rank(v)$. Next, assume that the

invariant is true for $j = n$. Now consider the situation after $n + 1$ iterations. Suppose $v \in V$ is reachable within $n + 1$ hops from $last2reach(v, i)$. Then, at least one neighbor $u \in N^-(v)$ of vertex v exists which is reachable within n hops from $last2reach(v, i)$ and $last2reach(u, i) = last2reach(v, i)$. Based on our assumption, $L2R(u, i)$ must have already converged to $rank(last2reach(u, i))$ before $(n+1)^{th}$ iteration. Therefore, Line 8 in Algorithm 1 ensures that $L2R(v, i) \leftarrow rank(last2reach(v, i))$ after $(n + 1)^{th}$ iteration. ◀

It is possible to design an adversarial example where the algorithm uses $\Omega(|V|)$ iterations. However, in practice, we expect the algorithm to converge quickly. Each iteration of Algorithm 1 requires $O(|\mathcal{P}||E|)$ time. Therefore, the total worst-case time of Algorithm 1 is bounded by $O(\Gamma_l|\mathcal{P}||E|)$. A similar approach is applicable to compute $D(last2reach(v, i), v)$ for all $v \in V$ and $i \in [1, |\mathcal{P}|]$ (Algorithm 2). We use Γ_d to denote the count of iterations needed in Algorithm 2. Similar to parameter Γ_l in Algorithm 1, Γ_d is also upper bounded by $|V|$. We will later show empirically that $\Gamma_l \ll |V|$ and $\Gamma_d \ll |V|$ in practice.

Algorithm 2 $O(\Gamma_d|\mathcal{P}||E|)$ time algorithm to compute $D(last2reach(v, i), v)$ for all $v \in V$ and $i \in [1, |\mathcal{P}|]$.

```
1  Initialize D(last2reach(v,i),v) to 0 if last2reach(v,i) = v and ∞ otherwise
2  Initialize D_prev(last2reach(v,i),v) to ∞
3  /*Arrays D and D_prev will hold the values of current and previous iteration respectively*/
4  while ∃v ∈ V, ∃i ∈ [1,|P|], D(last2reach(v,i),v) ≠ D_prev(last2reach(v,i),v) do
5      for i ∈ [1,|P|] do
6          for v ∈ V in the increasing order of rank(v) do
7              D_prev(last2reach(v,i),v) ← D(last2reach(v,i),v)
8              if last2reach(v,i) exists and last2reach(v,i) ≠ v then
9                  D(last2reach(v,i),v) ←
                     min_{u:u∈N⁻(v),last2reach(u,i)=last2reach(v,i)} D(last2reach(u,i),u) + |σ(u)|
10             end
11         end
12     end
13 end
```

Array $dist2begin$ is trivially precomputed in $O(|\mathcal{P}||V|)$ time. $D^\circ_\mathcal{P}(v)$ is computed as $\min_{u \in N^+(v)} |\sigma(v)| + D_\mathcal{P}(u, v)$ based on its definition. $D_\mathcal{P}(u, v)$ can be calculated by using Equation 1 for any $u, v \in V$ in $O(|\mathcal{P}|)$ time. Accordingly, computation of $D^\circ_\mathcal{P}(v)$ for all $v \in V$ is done in $O(|\mathcal{P}||E|)$ time. The following lemma summarises the worst-case time complexity of all the preprocessing steps.

▶ **Lemma 8.** *Preprocessing of graph $G(V, E, \sigma)$ requires $O((\Gamma_l + \Gamma_d + \log |V|)|\mathcal{P}||E|)$ time.*

3.2 Co-linear Chaining Algorithm

We propose an iterative chaining algorithm to address Problem 6. The proposed algorithm builds on top of the known algorithms for DAGs [5, 33]. Similar to [33], we maintain one search tree $\mathcal{T}_i$ for each path $P_i \in \mathcal{P}$. Given anchors $M[1..N]$, our algorithm will return array $C[1..N]$ such that $C[j]$ corresponds to the optimal score of a chain that ends at anchor $M[j]$.

If there are no cycles in G, then one iteration of Algorithm 3 suffices to compute the optimal chaining scores. For a DAG, a single iteration of Algorithm 3 works equivalently to the known algorithm for DAGs in [5]. In this case, Algorithm 3 would essentially visit the vertices of graph G in topological order while ensuring that $C[j]$ is optimally solved after $M[j].v$ is visited. To solve the chaining problem on cyclic graphs, we design an iterative solution where chaining scores $C[1..N]$ get closer to optimal values in each iteration. We will use Γ_c to specify the total count of iterations.

Algorithm 3 $O(\Gamma_c N|\mathcal{P}|\log N + N|\mathcal{P}|\log N|\mathcal{P}|)$ time chaining algorithm.

Input: Array of weighted anchors $M[1..N]$, preprocessed $G(V, E, \sigma)$
Output: Array $C[1..N]$ such that $C[j]$ equals score of an optimal chain that ends at anchor $M[j]$
Initialize search tree $\mathcal{T}_i$, for all $i \in [1, |\mathcal{P}|]$ using keys $\{M[j].d \mid 1 \leq j \leq N\}$ and values $-\infty$
Initialize $C[j]$ as $weight(M[j])$ and $C_{prev}[j] \leftarrow 0$, for all $j \in [1, N]$
/* Create array Z that stores tuples of the form $(v, pos, task, anchor, path)$ where $v \in V$, $pos \in \mathbb{N}$, $task \in \{1, 2, 3\}$, $anchor \in [1, N]$ and $path \in [1, |\mathcal{P}|]$*/
for $j \leftarrow 1$ *to* N **do**
  for $i \leftarrow 1$ *to* $|\mathcal{P}|$ **do**
    if $i \in paths(M[j].v)$ **then**
      $Z.push(M[j].v, M[j].x, 1, j, i)$
      $Z.push(M[j].v, M[j].y, 2, j, i)$
    end
    if $last2reach(M[j].v, i)$ *exists and* $last2reach(M[j].v, i) \neq M[j].v$ **then**
      $v \leftarrow last2reach(M[j].v, i)$
      $Z.push(v, |\sigma(v)| + 1, 1, j, i)$
    end
    if $M[j].v$ *is contained in a proper cycle in* G *and* $i \in paths(M[j].v)$ **then**
      $Z.push(v, |\sigma(M[j].v)| + 1, 3, j, i)$
    end
  end
end
while $\exists j \in [1, N]$, $C_{prev}[j] \neq C[j]$ **do**
  $C_{prev}[j] \leftarrow C[j]$, for all $j \in [1, N]$
  for $z \in Z$ *in lexicographically ascending order based on the key* $(rank(v), pos, task)$ **do**
    $j \leftarrow z.anchor, i \leftarrow z.path, v \leftarrow z.v, wt \leftarrow weight(M[j])$
    if $z.task = 1$ **then**
      $gap \leftarrow (M[j].x + Dist2begin(v, i) + D(v, M[j].v) + M[j].c - 2)$
      $C[j] \leftarrow \max(C[j],\ wt + \mathcal{T}_i.\mathsf{RMQ}(0, M[j].c) - gap)$
    end
    else if $z.task = 2$ **then**
      $\mathcal{T}_i.\mathsf{update}(M[j].d, C[j] + M[j].y + Dist2begin(v, i) + M[j].d)$
    end
    else if $z.task = 3$ **then**
      $gap^\circ \leftarrow (M[j].x + Dist2begin(v, i) + D^\circ_{\mathcal{P}}(v) + M[j].c - 2)$
      $C[j] \leftarrow \max(C[j],\ wt + \mathcal{T}_i.\mathsf{RMQ}(0, M[j].c) - gap^\circ)$
      $\mathcal{T}_i.\mathsf{update}(M[j].d,\ C[j] + M[j].y + Dist2begin(v, i) + M[j].d)$
    end
  end
  Reset all values in search tree $\mathcal{T}_i$ to $-\infty$, for all $i \in [1, |\mathcal{P}|]$
end

An overview of Algorithm 3 is as follows. At the beginning of each iteration, all search trees $\mathcal{T}_i$s are filled with keys $\{M[j].d \mid 1 \leq j \leq N\}$ and values $-\infty$. The values will be used to specify the priorities of anchors based on their scores $C[\,]$ and coordinates. Each iteration of our algorithm processes $v \in V$ in the increasing order of $rank(v)$. While processing v, Algorithm 3 performs three types of tasks:

1. The first type of task is to revise chaining scores $\{C[j] : M[j].v = v\}$ corresponding to the anchors that lie on vertex v. We also revise scores corresponding to those anchors that are located on vertex $u \neq v$ such that v is the last vertex on a path $\in \mathcal{P}$ to reach u. This is achieved by querying search trees $\mathcal{T}_i$ for all $i \in paths(v)$. In all these tasks, we use $D_{\mathcal{P}}(v_1, v_2)$ to calculate distance from vertex $v_1 \in V$ to vertex $v_2 \in V$.
2. Suppose score $C[j]$ is revised by using the first category tasks. The second type of task is to update the value of key $M[j].d$ in search trees $\mathcal{T}_i$ for all $i \in paths(v)$. The value gets updated if the new value is greater than the previously stored value (Lemma 1).
3. The third type of task is to again update scores $\{C[j] : M[j].v = v\}$ and search trees if v is part of a proper cycle in G. Here we use $D^\circ_{\mathcal{P}}(v)$ to calculate the distance of vertex v to itself while determining gap costs.

Lines 4–18 in Algorithm 3 build array Z that contains up to $4N|\mathcal{P}|$ tuples corresponding to all the above type of tasks. Array Z is sorted in $O(N|\mathcal{P}| \log N|\mathcal{P}|)$ time to ensure that all tasks are executed in the proper order (Line 21). Next, we start the iterative procedure. Lines 19–33 form a single iteration of the algorithm. These tasks lead to updates on score array C and the search trees. The arithmetic operations in Lines 24, 25, 31, 32 enable calculation of gap cost based on our definitions of gap_G and gap_Q in Section 2. Each iteration requires $O(N|\mathcal{P}| \log N)$ time because each task corresponds to either update or RMQ operation on a search tree of size $\leq N$. In Lemma 9, we prove that array $C[1..N]$ converges to optimality in at most N iterations. We will also prove that $\Omega(N)$ iterations are required for convergence in the worst case.

▶ **Lemma 9.** *In Algorithm 3, co-linear chaining scores $C[1..N]$ converge to optimality in $\leq N$ iterations.*

Proof. $C[j]$ always specifies the score of a chain of size ≥ 1 that ends at anchor $M[j]$ throughout the execution of the algorithm. Let $f_i(j)$ denote the optimal chaining score ending at anchor $M[j]$ over all chains of size $\leq i$. We will prove by induction that before i^{th} iteration begins, $C[j] \geq f_i(j)$ for all $j \in [1, N]$. It suffices to prove this statement because the size of a chain must be $\leq N$. Base case ($i = 1$) holds due to the initialization step in Line 2. Next, assume that before x^{th} iteration begins, $C[j] \geq f_x(j)$ holds for all $j \in [1, N]$. We will prove that the invariant holds for iteration $x + 1$.

Let $C_x[j]$ and $C_{x+1}[j]$ denote the intermediate values of $C[j]$ at the start of x^{th} and $(x+1)^{th}$ iteration, respectively. From Lines 25 and 32, we know $C_x[j] \leq C_{x+1}[j]$. If $f_{x+1}(j) = f_x(j)$, then $C_{x+1}[j] \geq C_x[j] \geq f_x(j) = f_{x+1}(j)$. Next consider the other case when $f_{x+1}(j) > f_x(j)$. Suppose the optimal chain corresponding to $f_{x+1}(j)$ is $M[\beta_1], M[\beta_2], \ldots, M[\beta_x], M[j]$ where $\beta_i \in [1, N]$ for all $i \in [1, x]$. Accordingly, $f_{x+1}(j) = weight(M[j]) + f_x(\beta_x) - gap_Q(M[\beta_x], M[j]) - gap_G(M[\beta_x], M[j])$. Based on our induction hypothesis, $C[\beta_x] \geq f_x(\beta_x)$ at the start of the x^{th} iteration. Each iteration of Algorithm 3 processes $v \in V$ by increasing the order of $rank(v)$. To prove that $C_{x+1}[j] \geq f_{x+1}(j)$, we have the following four cases to consider:

- Case 1: $rank(M[\beta_x].v) < rank(M[j].v)$. The algorithm processes vertex $M[\beta_x].v$ before vertex $M[j].v$. When $M[\beta_x].v$ is processed during the x^{th} iteration, the value of key $M[\beta_x].d$ gets updated in search trees (Line 28). $C[j]$ gets updated later. At the end of the x^{th} iteration, $C[j] \geq weight(M[j]) + f_x(\beta_x) - gap_Q(M[\beta_x], M[j]) - gap_G(M[\beta_x], M[j])$. Therefore, $C_{x+1}[j] \geq f_{x+1}(j)$.
- Case 2: $rank(M[\beta_x].v) > rank(M[j].v)$. In this case, $C[j]$ may not meet the desired threshold after $M[j].v$ is processed because $M[\beta_x].v$ is processed later than $M[j].v$. However, $M[j].v$ must be reachable from $M[\beta_x].v$ using walks through $\{last2reach(M[j].v, i) : i \in paths(M[\beta_x].v)\}$. Therefore, $C[j]$ gets updated again due to tuples created in Line 12. This will ensure that $C_{x+1}[j] \geq f_{x+1}(j)$.
- Case 3: $rank(M[\beta_x].v) = rank(M[j].v)$ and $M[\beta_x].y < M[j].x$. $rank(M[\beta_x].v) = rank(M[j].v)$ implies $M[\beta_x].v = M[j].v$. The ordering of tuples based on pos in Line 21 ensures that the value of key $M[\beta_x].d$ gets updated in search trees, and $C[j]$ gets updated afterward.
- Case 4: $rank(M[\beta_x].v) = rank(M[j].v)$ and $M[\beta_x].y \geq M[j].x$. The tuples created in Line 15 ensure that $C[j]$ is updated again after finishing the processing of vertex $M[j].v$. In this case, the gap between anchors $M[\beta_x]$ and $M[j]$ is computed by considering the distance of vertex $M[j].v$ to itself, i.e., $D^{\circ}_{\mathcal{P}}(M[j].v)$. ◀

Accordingly, the time complexity of Algorithm 3 is $O(\Gamma_c N|\mathcal{P}|\log N + N|\mathcal{P}|\log N|\mathcal{P}|)$. In our experiments, we will highlight that parameters Γ_c and $|\mathcal{P}|$ are small in practice. The space complexity of the algorithm is $O(N|\mathcal{P}| + |V||\mathcal{P}|)$ due to construction of array Z, the sorting operation on array Z, $|\mathcal{P}|$ search trees and the precomputed data structures. Next, we show that $O(N)$ upper bound on the number of iterations is tight.

▶ **Lemma 10.** *The count of iterations required by Algorithm 3 is $\Omega(N)$ in the worst-case.*

Proof. An example where Algorithm 3 requires $\Omega(N)$ iterations is shown in Figure 4. The graph has two vertices forming a cycle. Assume that *weight* of all N input anchors is equal and sufficiently high to outweigh the gap cost between any pair of anchors. As $M[1] \prec M[2] \prec M[3] \ldots \prec M[N]$, the sequence of anchors in the optimal chain is $(M[1], M[2], M[3], \ldots, M[N-1], M[N])$. After the first iteration of the algorithm, the size of the highest scoring chain computed until then will be $\frac{N}{2} + 1$. The size will grow slowly by one in each subsequent iteration. A step-by-step dry run of the algorithm is left for the journal version of this paper. ◀

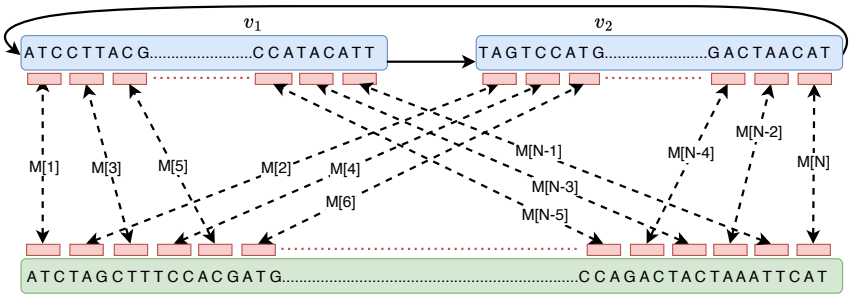


Figure 4 A worst-case example for Algorithm 3 where it requires $\Omega(N)$ iterations to converge.

4 Implementation

We have implemented the proposed algorithm in `C++` (https://github.com/at-cg/PanAligner). We call our software as PanAligner. PanAligner is developed as an end-to-end long-read aligner for cyclic pangenome graphs. We borrow open-source code from Minichain [5], Minigraph [24], and GraphChainer [30] for other necessary components besides co-linear chaining. While using PanAligner, a user needs to provide a graph (`GFA` format) and a set of reads or contigs (`fasta` or `fastq` format) as input. We use the standard data structure to store the pangenome graph while accounting for double stranded nature of DNA sequences. 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 add the complementary edge $\bar{v} \to \bar{u}$. This enables read alignment irrespective of which strand the read was sequenced from.

For the benchmark, we built pangenome graphs by using Minigraph v0.20 [24]. Minigraph augments large insertion, deletion, and inversion variants into the graph while incrementally aligning each input assembly. Inversion variants can introduce cycles in the graph because Minigraph augments them by linking the vertices from opposite strands. The graph contains multiple weakly connected components because the components corresponding to different chromosomes are never linked during graph construction. Similar to [5, 30], we consider each weak component independently during both the preprocessing and co-linear chaining stages to enable efficient multithreading and memory optimization.

We defined our problem formulation to produce an optimal chain, but we actually compute multiple best chains, similar to [5, 23, 24]. This is because there can be multiple high-scoring alignments of a read on the graph. PanAligner also outputs a mapping quality score between 0 to 60 to indicate the confidence score for each alignment [25]. We used seeding and extension code from Minigraph [24]. Seeding is done by identifying minimizer matches [45] between vertex labels of the graph and the read. The extension code produces the final base-to-base alignment by joining the chained anchors [52]. We used code from GraphChainer [30] to compute the minimum path cover of a DAG and range queries.

5 Experiments

Benchmark Datasets

We constructed four cyclic pangenome graphs by using subsets of publicly available 95 haplotype-resolved human genome assemblies [26, 37]. These graphs were generated using Minigraph v0.20 [24]. We used CHM13 human genome assembly [37] as the starting sequence during graph construction in all four graphs. We refer to these graphs as 10H, 40H, 80H, and 95H, where the prefix integer represents the count of haplotypes in each graph. The properties of these graphs are provided in Table 1.

Table 1 Properties of four cyclic pangenome graphs that were used for evaluation.

Graph	|V|	|E|	No. of weak components	No. of structural variants	N50 length of vertex labels (kb)
10H	283,296	406,292	30	61,523	225
40H	679,846	978,122	28	149,163	127
80H	1,106,286	1,594,980	26	244,372	85
95H	1,224,853	1,765,222	26	270,888	79

Evaluation Methodology

We simulated long reads using PBSIM2 v2.0.1 [39] from CHM13 assembly with N50 length 10 kb, 0.5× sequencing coverage and 5% error-rate to approximately mimic the properties of long-reads. We labeled the IDs of the simulated reads with their true interval coordinates in the CHM13 assembly for correctness evaluation. To confirm the correctness of a read alignment, we used similar criteria from prior studies [5, 23, 24]. We require that the reported walk corresponding to a correct alignment should only use the vertices corresponding to the CHM13 assembly in the graph, and it should overlap with the true walk. We used `paftools` [23] to automate this evaluation. By default, it requires the overlapping portion to be at least 10% of the union of the true and the reported walk length. We executed all experiments on a computer with AMD EPYC 7763 64-core processor and 512 GB RAM. We ran each aligner using 32 threads to leverage the multi-threading capabilities of the tested aligners. All aligners process reads in parallel. We used the `/usr/bin/time -v` command to measure wall clock time and peak memory usage.

Size of Path Cover and Count of Iterations

Finding a suitable path cover $\mathcal{P}$ of the input graph such that $|\mathcal{P}| \ll |V|$ is a crucial step in our proposed framework because the scalability of our algorithms depends on this parameter. We discussed a heuristic to compute path cover in Section 3.1 because determining minimum

path cover in general graphs is NP-hard. Table 2 shows the sizes of path covers computed by our heuristic in all four graphs. Recall that our algorithms process the weakly connected components of a graph independently. In each graph, we indicate the size of the path cover as a range because path covers vary per component. The results show that our heuristic is effective in finding a small path cover (in the number of paths).

Table 2 All four graphs have multiple weakly connected components. Therefore, the size of the identified path cover of each graph is presented as a range. The other columns show the statistics for the number of anchors and the number of iterations used by our iterative algorithms (Algorithms 1, 2, 3). The statistics were gathered while aligning simulated long reads to cyclic pangenome graphs.

Graph	Size of path cover (min-max)	Number of anchors		Number of iterations					
				Array *last2reach*		Array D		Chaining	
		Mean	Max	Mean	Max	Mean	Max	Mean	Max
10H	1-20	10,900	309,591	2.0	4	2.0	5	2.3	77
40H	1-36	10,850	309,603	2.0	4	2.0	5	2.4	72
80H	1-49	10,804	309,364	3.0	4	3.0	5	2.4	61
95H	1-59	10,798	309,367	3.0	4	3.0	5	2.4	64

The number of anchors N that were provided as input to the co-linear chaining algorithm varies per read. We report the mean and maximum value in Table 2. Observe that N does not change much with increasing haplotype count. Next, we evaluate the count of iterations Γ_l, Γ_d used by our graph preprocessing algorithms (Algorithms 1–2) and also report them as a range for each graph. These algorithms compute *last*2*reach* and D arrays. Observe that the iteration count is significantly smaller in practice than the proven upper limit of $|V|$ (Lemma 7). This is because the worst-case situation is not observed in practice. Accordingly, there is minimal time overhead during the preprocessing phase.

The count of iterations Γ_c required by our chaining algorithm (Algorithm 3) varies per component as well as per read. We collect the iteration count statistics as follows. For a single read, we define the iteration count as the maximum number of iterations used over all components. Based on this definition, we report the average and the maximum count over all reads in Table 2. Observe that the average count is < 2.5 using all four graphs. The maximum count is < 100. These numbers are again significantly better compared to the upper bound from Lemma 9.

Alignment of Simulated Reads to Cyclic Graphs

We assessed the performance of PanAligner against two sequence-to-graph aligners, Minigraph v2.20 [24] and GraphAligner v1.0.17b [42], that can handle cycles. Unlike PanAligner, Minigraph and GraphAligner use heuristics to join anchors. Minichain [5] and GraphChainer [30] were excluded from this comparison because they do not support cyclic graphs.

We highlight the accuracy, runtime, and memory usage of different aligners using graphs 10H and 95H in Tables 3 and 4, respectively. Observe that PanAligner outperformed Minigraph and GraphAligner in terms of accuracy, i.e., the fraction of correctly aligned reads. This advantage is even more apparent if low-confidence alignments with mapping quality < 10 are ignored. We show the comparison plots in Figure 5. Both PanAligner and Minigraph left a small fraction of reads unaligned. This may be because (i) both methods drop high-frequency minimizer matches, and (ii) they do not consider low-scoring chains for the extension stage. In contrast, GraphAligner achieved higher recall by aligning all reads, but this came at the expense of lower accuracy.

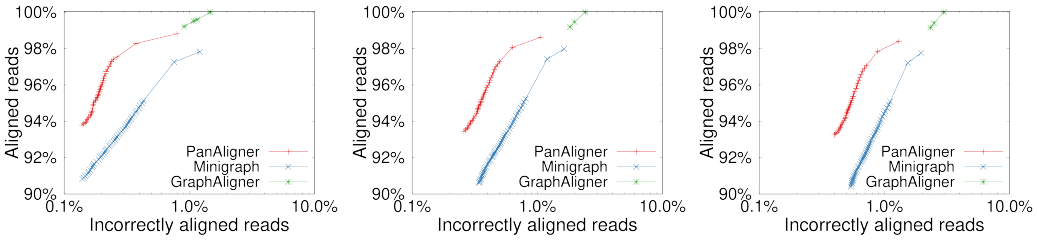
Table 3 A comparison of the performance of long-read aligners using the 10H graph. MQ stands for mapping quality.

	PanAligner	Minigraph	GraphAligner
Indexing time (sec)	96	**57**	238
Alignment time (sec)	2924	**46**	4928
Memory usage (GB)	**23.14**	23.19	24.68
Unaligned reads	1.18%	2.17%	**0%**
Incorrectly Aligned reads	**0.79%**	1.19%	1.47%
Unaligned reads (MQ≥10)	3.51%	5.85%	**0.78%**
Incorrectly Aligned reads (MQ≥10)	**0.20%**	0.32%	0.91%

Table 4 A comparison of the performance of long-read aligners using the 95H graph. MQ stands for mapping quality.

	PanAligner	Minigraph	GraphAligner
Indexing time (sec)	83	**61**	272
Alignment time (sec)	9276	**58**	5170
Memory usage (GB)	43.6	**24.68**	26.1
Unaligned reads	1.60%	2.24%	**0%**
Incorrectly Aligned reads	**1.28%**	1.93%	2.98%
Unaligned reads (MQ≥10)	4.20%	6.21%	**0.84%**
Incorrectly Aligned reads (MQ≥10)	**0.57%**	0.85%	2.33%

Table 2 shows that the size of the path cover computed by our heuristic increases by roughly a factor of three from 10H to 95H. We can see how this parameter proportionally affects PanAligner's runtime in Tables 3 and 4. PanAligner's runtime is significantly higher than Minigraph for both 10H and 95H graphs because it uses an iterative algorithm. Runtimes of PanAligner and GraphAligner are in the same order of magnitude. PanAligner's computational needs are within practical limits, thus making it an effective method for accurately aligning long reads or contigs to cyclic pangenome graphs. We observe a consistent drop in alignment accuracy of all three aligners with increasing haplotype count (Figure 5). This is likely because the number of combinatorial paths to which a read can align grows exponentially with respect to the haplotype count.



(a) 10H. **(b)** 40H. **(c)** 95H.

Figure 5 The plots show the fraction of aligned reads and the accuracy obtained by using all the aligners on graphs 10H, 40H, and 95H. These plots were generated by varying mapping quality cutoffs from 0 to 60. X-axis in these plots uses a logarithmic scale to indicate the percentage of incorrectly aligned reads.

Alignment of Simulated Reads to Acyclic Graphs

We also tested PanAligner for acyclic pangenome graphs. We followed the same procedure as [5] to generate a DAG from 95 haplotype-phased assemblies and refer to this graph as 95H-DAG. This graph was generated by disabling inversion variants during graph construction in Minigraph [24]. 95H-DAG has 1.2M vertices and 1.8M edges. We also include Minichain v1.0 [5] and GraphChainer v1.0.2 [30] in this comparison. GraphChainer uses a co-linear chaining algorithm for DAGs without penalizing gaps. For DAG inputs, the problem formulation in PanAligner becomes equivalent to the one used in Minichain [5]. A single iteration of our algorithms suffices for DAGs. Therefore, we simply check if the input graph is a DAG at the preprocessing stage, and run a single iteration of Algorithms 1–3. PanAligner achieves similar performance as Minichain in terms of speed and accuracy for DAGs (Table 5). It compares favorably to other methods in terms of accuracy.

Table 5 A comparison of the performance of long-read aligners using the 95H-DAG graph. MQ stands for mapping quality.

	Pan-Aligner	**Minichain**	**Mini-graph**	**Graph-Aligner**	**Graph-Chainer**
Indexing time(sec)	78	77	**62**	276	575
Alignment time(sec)	2406	2515	**50**	5136	23710
Memory usage (GB)	30.04	25.61	**24.79**	26.12	185.83
Unaligned reads	1.62%	1.62%	2.23%	**0%**	**0%**
Incorrectly Aligned reads	**1.28%**	1.29%	1.92%	3.06%	4.93%
Unaligned reads (MQ≥10)	4.75%	4.75%	6.26%	0.85%	**0%**
Incorrectly Aligned reads (MQ≥10)	**0.53%**	0.54%	0.84%	2.41%	4.93%

6 Discussion

Co-linear chaining is a fundamental technique for scalable sequence alignment. Several classes of structural variants, such as duplications, tandem repeat polymorphism, and inversions, are best represented as cycles in pangenome graphs [41, 26]. Existing alignment software designed for cyclic graphs are based on heuristics to join anchors [24, 42]. We proposed the first practical problem formulation and an efficient algorithm for co-linear chaining on pangenome graphs with cycles. We gave a rigorous analysis of the algorithm's time complexity. The proposed algorithm serves as a useful generalization of the previously known ideas for DAGs [5, 29, 30, 33]. We implemented the proposed algorithm as an open-source software PanAligner. We demonstrated that PanAligner scales to large pangenome graphs built by using haplotype-phased human genome assemblies. It offers superior alignment accuracy compared to existing methods.

In our formulation, we did not allow anchors to span two or more vertices in a graph for simplicity of notations, but the proposed ideas can be generalized. PanAligner software borrows seeding logic from Minigraph [24], which also restricts anchors within a single vertex. This simplification is appropriate if the graph only includes structural variants (> 50 bp). The current version of PanAligner software may not be suitable for graphs which include substitution and indel variants.

Future work will be directed in the following directions. First, we will test the performance of PanAligner on pangenome graphs that are constructed by using alternative methods, e.g., [16, 19, 26]. Second, we will explore formulations for haplotype-constrained co-linear

chaining to control the exponential growth of combinatorial search space with the increasing number of haplotypes [34, 48]. Third, we will generalize the proposed techniques for aligning reads to long-read genome assembly graphs which also contain cycles. It will be interesting to understand whether the small width assumption is appropriate for assembly graphs.

References

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

2 Jasmijn A Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Natural Computing*, pages 1–28, 2022.

3 Arturs Backurs and Piotr Indyk. 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*, pages 51–58, 2015.

4 Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. 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)*, pages 359–376. SIAM, 2022.

5 Ghanshyam Chandra and Chirag Jain. Sequence to graph alignment using gap-sensitive co-linear chaining. In *Research in Computational Molecular Biology: 27th Annual International Conference, RECOMB 2023, Istanbul, Turkey, April 16–19, 2023, Proceedings*, pages 58–73. Springer, 2023.

6 Haoyu Cheng, Mobin Asri, Julian Lucas, Sergey Koren, and Heng Li. Scalable telomere-to-telomere assembly for diploid and polyploid genomes with double graph. *arXiv preprint*, 2023. `arXiv:2306.03399`.

7 Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in bioinformatics*, 19(1):118–135, 2018.

8 Egor Dolzhenko, Viraj Deshpande, Felix Schlesinger, Peter Krusche, Roman Petrovski, Sai Chen, Dorothea Emig-Agius, Andrew Gross, Giuseppe Narzisi, Brett Bowman, et al. Expansionhunter: a sequence-graph-based tool to analyze variation in short tandem repeat regions. *Bioinformatics*, 35(22):4754–4756, 2019.

9 Tatiana Dvorkina, Dmitry Antipov, Anton Korobeynikov, and Sergey Nurk. Spaligner: alignment of long diverged molecular sequences to assembly graphs. *BMC bioinformatics*, 21(12):1–14, 2020.

10 Peter Eades, Xuemin Lin, and W.F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, October 1993.

11 Hannes P Eggertsson, Hakon Jonsson, Snaedis Kristmundsdottir, et al. Graphtyper enables population-scale genotyping using pangenome graphs. *Nature genetics*, 49(11):1654–1660, 2017.

12 David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F Italiano. Sparse dynamic programming i: linear cost functions. *Journal of the ACM*, 39(3):519–545, 1992.

13 David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F Italiano. Sparse dynamic programming ii: convex and concave cost functions. *Journal of the ACM*, 39(3):546–567, 1992.

14 Yang Gao, Xiaofei Yang, Hao Chen, Xinjiang Tan, Zhaoqing Yang, Lian Deng, Baonan Wang, Shuang Kong, Songyang Li, Yuhang Cui, et al. A pangenome reference of 36 chinese populations. *Nature*, pages 1–10, 2023.

15 Shilpa Garg, Mikko Rautiainen, Adam M Novak, et al. A graph-based approach to diploid genome assembly. *Bioinformatics*, 34(13):i105–i114, 2018.

16 Erik Garrison, Andrea Guarracino, Simon Heumos, Flavia Villani, Zhigui Bao, Lorenzo Tattini, Jörg Hagmann, Sebastian Vorbrugg, Santiago Marco-Sola, Christian Kubica, et al. Building pangenome graphs. *bioRxiv*, pages 2023–04, 2023.

17 Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36(9):875–879, October 2018. doi:10.1038/nbt.4227.

18 Daniel Gibney, Sharma V Thankachan, and Srinivas Aluru. The complexity of approximate pattern matching on de Bruijn graphs. In *Research in Computational Molecular Biology: 26th Annual International Conference, RECOMB 2022, San Diego, CA, USA, May 22–25, 2022, Proceedings*, pages 263–278. Springer, 2022.

19 Glenn Hickey, Jean Monlong, Jana Ebler, Adam M Novak, Jordan M Eizenga, Yan Gao, Tobias Marschall, Heng Li, and Benedict Paten. Pangenome graph construction from genome alignments with minigraph-cactus. *Nature Biotechnology*, pages 1–11, 2023.

20 Chirag Jain, Daniel Gibney, and Sharma V Thankachan. Algorithms for colinear chaining with overlaps and gap costs. *Journal of Computational Biology*, 29(11):1237–1251, 2022.

21 Chirag Jain, Arang Rhie, Nancy F Hansen, Sergey Koren, and Adam M Phillippy. Long-read mapping to repetitive reference sequences using winnowmap2. *Nature Methods*, pages 1–6, 2022.

22 Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. On the complexity of sequence-to-graph alignment. *Journal of Computational Biology*, 27(4):640–654, April 2020. doi:10.1089/cmb.2019.0066.

23 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, May 2018. doi:10.1093/bioinformatics/bty191.

24 Heng Li, Xiaowen Feng, and Chong Chu. The design and construction of reference pangenome graphs with minigraph. *Genome Biology*, 21(1), October 2020.

25 Heng Li, Jue Ruan, and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008.

26 Wen-Wei Liao, Mobin Asri, Jana Ebler, Daniel Doerr, Marina Haukness, Glenn Hickey, Shuangjia Lu, Julian K Lucas, Jean Monlong, Haley J Abel, et al. A draft human pangenome reference. *Nature*, 617(7960):312–324, 2023.

27 Tsung-Yu Lu, Human Genome Structural Variation Consortium, et al. Profiling variable-number tandem repeat variation across populations using repeat-pangenome graphs. *Nature Communications*, 12(1):4250, 2021.

28 Xiao Luo, Xiongbin Kang, and Alexander Schönhuth. Vechat: correcting errors in long reads using variation graphs. *Nature Communications*, 13(1):6657, 2022.

29 Jun Ma. Co-linear chaining on graphs with cycles. Master's thesis, University of Helsinki, Faculty of Science, 2021. URL: http://hdl.handle.net/10138/330781.

30 Jun Ma, Manuel Cáceres, Leena Salmela, Veli Mäkinen, and Alexandru I Tomescu. Chaining for accurate alignment of erroneous long reads to acyclic variation graphs. *bioRxiv*, 2022. doi:10.1101/2022.01.07.475257.

31 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

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

33 Veli Mäkinen, Alexandru I Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse dynamic programming on DAGs with small width. *ACM Transactions on Algorithms (TALG)*, 15(2):1–21, 2019.

34 Tom Mokveld, Jasper Linthorst, Zaid Al-Ars, Henne Holstege, and Marcel Reinders. Chop: haplotype-aware path indexing in population graphs. *Genome biology*, 21:1–16, 2020.

35 Gene Myers and Webb Miller. Chaining multiple-alignment fragments in sub-quadratic time. In *SODA*, volume 95, pages 38–47, 1995.

36 Gonzalo Navarro. Improved approximate pattern matching on hypertext. *Theoretical Computer Science*, 237(1-2):455–463, 2000.

37 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022.

38 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, April 2022. doi:10.1126/science.abj6987.

39 Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. Pbsim2: a simulator for long-read sequencers with a novel generative model of quality scores. *Bioinformatics*, 37(5):589–595, 2021.

40 Christian Otto, Steve Hoffmann, Jan Gorodkin, and Peter F Stadler. Fast local fragment chaining using sum-of-pair gap costs. *Algorithms for Molecular Biology*, 6(1):4, 2011.

41 Benedict Paten, Adam M Novak, Jordan M Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome research*, 27(5):665–676, 2017.

42 Mikko Rautiainen and Tobias Marschall. Graphaligner: rapid and versatile sequence-to-graph alignment. *Genome biology*, 21(1):253, 2020.

43 Mikko Rautiainen, Sergey Nurk, Brian P Walenz, Glennis A Logsdon, David Porubsky, Arang Rhie, Evan E Eichler, Adam M Phillippy, and Sergey Koren. Telomere-to-telomere assembly of diploid chromosomes with verkko. *Nature Biotechnology*, pages 1–9, 2023.

44 Nicola Rizzo, Manuel Cáceres, and Veli Mäkinen. Chaining of maximal exact matches in graphs. *arXiv preprint*, 2023. arXiv:2302.01748.

45 Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

46 Kristoffer Sahlin, Thomas Baudeau, Bastien Cazaux, and Camille Marchet. A survey of mapping algorithms in the long-reads era. *bioRxiv*, 2022.

47 Leena Salmela and Eric Rivals. Lordec: accurate and efficient long read error correction. *Bioinformatics*, 30(24):3506–3514, 2014.

48 Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020.

49 Jouni Sirén, Jean Monlong, Xian Chang, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, 2021.

50 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

51 Ting Wang, Lucinda Antonacci-Fulton, Kerstin Howe, et al. The human pangenome project: a global resource to map genomic diversity. *Nature*, 604(7906):437–446, April 2022.

52 Haowen Zhang, Shiqi Wu, Srinivas Aluru, and Heng Li. Fast sequence to graph alignment using the graph wavefront algorithm. *arXiv preprint*, 2022. arXiv:2206.13574.

53 Yao Zhou, Zhiyang Zhang, Zhigui Bao, Hongbo Li, Yaqing Lyu, Yanjun Zan, Yaoyao Wu, Lin Cheng, Yuhan Fang, Kun Wu, et al. Graph pangenome captures missing heritability and empowers tomato breeding. *Nature*, 606(7914):527–534, 2022.