Rajput *et al. Algorithms for Molecular Biology* (2024) 19:4
https://doi.org/10.1186/s13015-024-00250-w

Algorithms for Molecular Biology

RESEARCH

Open Access

Co-linear chaining on pangenome graphs

Jyotshna Rajput[1], Ghanshyam Chandra[1] and Chirag Jain[1*]

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 and how incorporating gap cost in the scoring function improves alignment accuracy. 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.

Implementation (https://github.com/at-cg/PanAligner).

Keywords Sequence alignment, Variation graph, Genome sequencing, Path cover

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 [1–7]. 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 [4, 5, 8–10]. As a result, the draft pangenome graphs published by the Human Pangenome Reference Consortium [4] and the Chinese Pangenome Consortium [11] are also cyclic. Aligning reads or assembly contigs to a directed labeled graph is a fundamental problem in computational pangenomics [12, 13]. Aligning reads to graphs is also useful for other bioinformatics tasks such as long-read de novo assembly [14–16] and long-read error correction [17, 18].

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 [19, 20]. The known conditional lower bound [21] implies that an exact algorithm significantly faster than $O(|Q||E|)$ is unlikely. This lower bound also holds for de Bruijn graphs [22]. Therefore, fast heuristics are used to process high-throughput sequencing data.

*Correspondence:
Chirag Jain
chirag@iisc.ac.in
[1] Department of Computational and Data Sciences, Indian Institute of Science, Bangalore 560012, Karnataka, India

Seed-chain-extend is a common heuristic used by modern alignment tools [23–25]. 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 (Fig. 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 [13]. Many sequence-to-graph aligners already exist that differ in terms of implementing these stages [5, 26–30]. This paper addresses the generalization of the chaining stage to cyclic pangenome graphs (Figs. 2, 3).

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 [31–37]. 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 [31].

The first effort to generalize the co-linear chaining problem to graphs was made by Makinen et al. [38]. They addressed the co-linear chaining problem on directed acyclic graphs (DAGs). The authors introduced a sparse dynamic programming algorithm whose

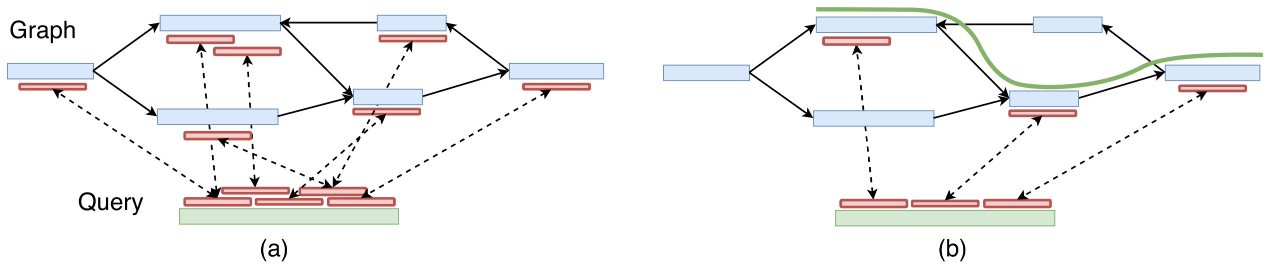

Fig. 1 An illustration of co-linear chaining for sequence-to-graph alignment. Assume that the vertices of the graph are labeled with nucleotide sequences. In panel **a**, short exact matches, i.e., anchors, are illustrated using red blocks joined by dotted lines. In panel **b**, the anchors corresponding 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 (illustrated using a green curved line)

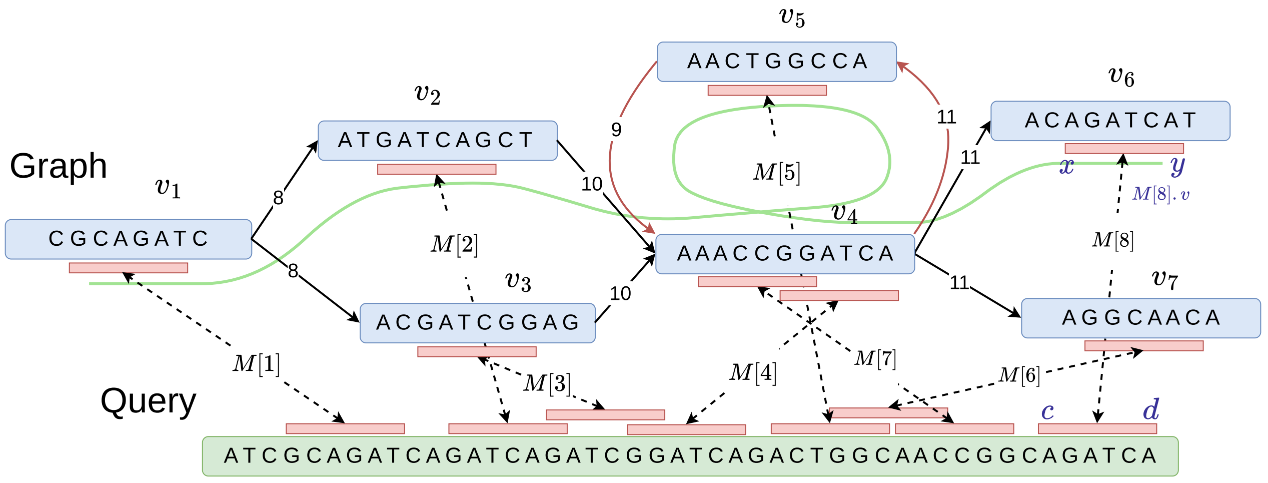

Fig. 2 An example illustrating a graph, a query sequence, and multiple anchors as input for co-linear chaining. The sequence of anchors (M[1], M[2], M[4], M[5], M[7], M[8]) forms a valid chain that visits vertex v_4 twice due to a cycle in the graph. The coordinates associated with anchor M[8] are also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

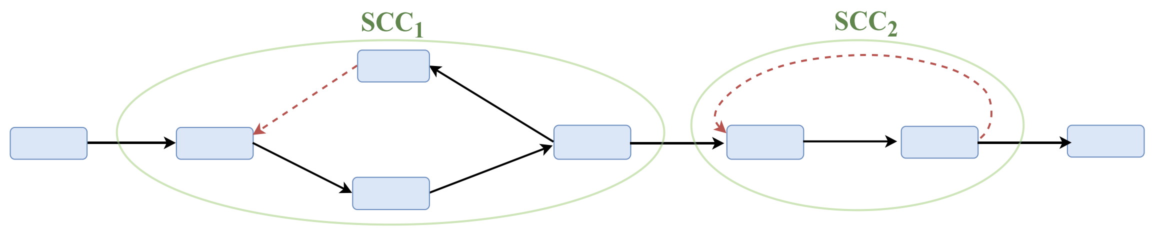

Fig. 3 An illustration of the proposed heuristic used to convert a cyclic graph into a DAG. Red-dotted edges represent the removed back edges in each strongly connected component (SCC)

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 [26, 29, 38]. An optimized version of their algorithm requires $O(KN \log KN)$ time for chaining, where K is the width of the DAG [29]. This formulation has been further extended to incorporate gap cost in the scoring function [26], and for solving the longest common subsequence problem between a DAG and a sequence [39].

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 [29, 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.

Contributions

In this paper, we build on top of the algorithmic techniques developed for DAGs [26, 29, 38] and propose novel formulations for cyclic pangenome graphs. Our proposed algorithm exploits the small width of pangenome graphs similar to [38]. Our approach for defining the gap cost between a pair of anchors is inspired by the corresponding function defined on DAGs [26].

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 [26, 29, 38], 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 directed graphs [40]. 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 during preprocessing, followed by chaining of anchors 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 $\Gamma_c, \Gamma_l, \Gamma_d$ are small in practice to justify the practicality of this approach. The runtime complexity also depends on $|\mathcal{P}|$, which is determined by our path cover finding heuristic. We show that the number of paths in our path cover is small and near-optimal in practice.

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 [28]. We evaluated the scalability and alignment accuracy of PanAligner by using a cyclic human pangenome graph constructed from 94 high-quality haplotype-resolved assemblies [4] and CHM13 human genome assembly [41]. We achieve the highest long-read mapping accuracy 98.7% using PanAligner when compared to existing methods Minigraph [28] (98.1%) and GraphAligner [30] (97.0%). PanAligner also supports a hybrid method which identifies a subset of reads that are relatively “easy-to-align” and utilizes fast Minigraph heuristics [28] for aligning them. This option significantly improves the speed of the algorithm.

Notations and problem formulations

Pangenome graph $G(V, E, \sigma)$ is a string labeled graph such that function $\sigma : V \rightarrow \Sigma^+$ 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. [42]) *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 [29].

Definition 1 (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 2 (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 [26]. 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 $gap_G(s_j, s_{j+1})$.

Problem 1 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 2 *Problem 1 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 [26], 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 following.

$$D_\mathcal{P}(v_1, v_2) = \min_{i \in paths(v_1)} dist2begin(\alpha_i, i) - dist2begin(v_1, i) + D(\alpha_i, v_2) \tag{1}$$

$D_\mathcal{P}(v_1, v_2)$ 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 2 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}.

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 to reduce the runtime of our chaining algorithm for solving Problem 2.

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. Later, we will empirically show that $|\mathcal{P}|$ is very close to optimal by comparing it to a lower bound on the size of the minimum path cover.
- A bijective function $rank : V \rightarrow [1, |V|]$ 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 [38]. 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., [43], 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 in G and perform a depth-first search within each strong component to remove back edges [44]. This approach provides a DAG that has the same number of weak components as G while removing a small number of edges in practice.

It is important to verify that the above heuristic actually results in a path cover whose size is close to optimal because the runtime complexity of our algorithms depends on $|\mathcal{P}|$. Computing the minimum path cover is difficult due to NP-hardness of the problem. Instead, we compute a lower bound on the size of the minimum path cover using a flow-based method. This method is inspired from a known relationship between minimum path cover problem and *minimum flow problem* in DAGs [38, 45].

In the minimum flow problem, the input is a directed graph with a single source, a single sink, and a demand value $\in \mathbb{Z}$ for every edge. The task is to find a flow of minimum *value* that satisfies all demands. The *value* of a flow is the sum of the flow on the edges exiting the source. We compute a new graph G^* from G by (i) replacing each vertex v with two vertices (v^-, v^+), (ii) joining all in-neighbors of v to v^-, and (iii) joining out-neighbors of v from v^+. We add a global source with an out-going edge to every vertex and a global sink with an in-coming edge from every vertex. The demand on all edges of type (v^-, v^+) is set to one in G^*. The demand on all the remaining edges is set to zero. Observe that any path cover $\mathcal{P}$ of G can be converted into a feasible flow of value $|\mathcal{P}|$ in G^*. As a result, the value of minimum flow in G^* must be less than or equal to the size of the minimum path cover in G. Thus, we can solve the minimum flow problem to know a lower bound on the size of the minimum path cover. In our experiments, we compute and use the lower bounds to establish the effectiveness of our path cover finding heuristic.

Next, we compute a function *rank* for all vertices $\in V$ by topological sorting of vertices in DAG G'. 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 [26, 38]. 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 3, 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 $i \in paths(v)$ and 0 otherwise for all $v \in V$ and
   $i \in [1, |\mathcal{P}|]$
2: Initialize $L2R_{prev}(v,i)$ to 0 for all $v \in V$ and $i \in [1, |\mathcal{P}|]$ ▷ $L2R$ and $L2R_{prev}$ will
   hold the values of current and previous iteration respectively
3: while $\exists v \in V$, $\exists i \in [1, |\mathcal{P}|]$, $L2R(v,i) \neq L2R_{prev}(v,i)$ do
4:     for $i \in [1, |\mathcal{P}|]$ do
5:         for $v \in V$ in the increasing order of $rank(v)$ do
6:             $L2R_{prev}(v,i) \leftarrow L2R(v,i)$
7:             $L2R(v,i) \leftarrow \max_{u \in N^-(v) \cup \{v\}} L2R(u,i)$
8:         end for
9:     end for
10: end while
```

Lemma 3 *In Algorithm* 1, *L*2*R*(*v*, *i*) *converges to the rank of last*2*reach*(*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, *L*2*R*(*v*, *i*) has converged to *rank*(*last*2*reach*(*v*, *i*)) if *last*2*reach*(*v*, *i*) exists and vertex v is reachable within j hops from *last*2*reach*(*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 *L*2*R*(*v*, *i*) in Line 1. If v lies 0-hop from *last*2*reach*(*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 *last*2*reach*(*v*, *i*). Then, at least one neighbor $u \in N^-(v)$ of vertex v exists which is reachable within n hops from *last*2*reach*(*v*, *i*) and $last2reach(u, i) = last2reach(v, i)$. Based on our assumption, *L*2*R*(*u*, *i*) must have already converged to *rank*(*last*2*reach*(*u*, *i*)) before $(n + 1)^{th}$ iteration. Therefore, Line 7 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*(*last*2*reach*(*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*(*last*2*reach*(*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 $\infty$ otherwise
2:  Initialize $D_{prev}(last2reach(v,i),v)$ to $\infty$    ▷ Arrays $D$ and $D_{prev}$ will hold the
    values of the current and previous iteration, respectively
3:  while $\exists v \in V,\ \exists i \in [1, |\mathcal{P}|],\ D(last2reach(v,i),v) \neq D_{prev}(last2reach(v,i),v)$ do
4:      for $i \in [1, |\mathcal{P}|]$ do
5:          for $v \in V$ in the increasing order of $rank(v)$ do
6:              $D_{prev}(last2reach(v,i),v) \leftarrow D(last2reach(v,i),v)$
7:              if $last2reach(v,i)$ exists and $last2reach(v,i) \neq v$ then
8:                  $D(last2reach(v,i),v) \leftarrow \min_{u:u \in N^-(v), last2reach(u,i)=last2reach(v,i)}$
9:                  $D(last2reach(u,i),u) + |\sigma(u)|$
10:             end if
11:         end for
12:     end for
13: end while
```

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 Eq. 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 4 *Preprocessing of graph* $G(V, E, \sigma)$ *requires* $O((\Gamma_l + \Gamma_d + \log|V|)|\mathcal{P}||E|)$ *time.*

Co-linear chaining algorithm

We propose an iterative chaining algorithm to address Problem 2. The proposed algorithm builds on top of the known algorithms for DAGs [26, 38]. Similar to [38], 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 [26]. 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

```
Require: Array of weighted anchors $M[1..N]$, preprocessed $G(V, E, \sigma)$
Ensure: 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 $\{M[j].d \mid 1 \le j \le N\}$ 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
        if $last2reach(M[j].v, i)$ exists and $last2reach(M[j].v, i) \ne M[j].v$ then
            $v \leftarrow last2reach(M[j].v, i)$
            $Z.push(v, |\sigma(v)| + 1, 1, j, i)$
        end if
        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 if
    end for
end for
while $\exists j \in [1, N], C_{prev}[j] \ne C[j]$ 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
            $val \leftarrow (M[j].x + Dist2begin(v, i) + D(v, M[j].v) + M[j].c - 2)$
            $C[j] \leftarrow \max(C[j],\ wt + \mathcal{T}_i.\text{RMQ}(0, M[j].c) - val)$
        end if
        if $z.task = 2$ then
            $\mathcal{T}_i.\text{update}(M[j].d, C[j] + M[j].y + Dist2begin(v, i) + M[j].d)$
        end if
        if $z.task = 3$ then
            $val \leftarrow (M[j].x + Dist2begin(v, i) + D^{\circ}_{\mathcal{P}}(v) + M[j].c - 2)$
            $C[j] \leftarrow \max(C[j],\ wt + \mathcal{T}_i.\text{RMQ}(0, M[j].c) - val)$
            $\mathcal{T}_i.\text{update}(M[j].d,\ C[j] + M[j].y + Dist2begin(v, i) + M[j].d)$
        end if
    end for
    Reset all values in search tree $\mathcal{T}_i$ to $-\infty$, for all $i \in [1, |\mathcal{P}|]$
end while
```

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 value of key $M[j].d$ will be updated based on the score $C[j]$ and some other parameters. The range search trees will be used to efficiently identify the optimal preceding anchor for each anchor [26, 29, 38].

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. We update the priority of an anchor $M[j]$ in the relevant search trees using its score $C[j]$ and the coordinates (Lines 28, 33). To update the score of an anchor $M[j]$ based on the scores of its preceding anchors, we use the (i) highest priority value obtained from the search trees, (ii) the coordinates of anchor $M[j]$, and (iii) the precomputed arrays D, $dist2begin$, $D^{\circ}_{\mathcal{P}}$ (Lines 24, 25, 31, 32). The score calculations are consistent with the gap cost definition in Sect. Notations and problem formulations.

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 5, we prove that array $C[1..N]$ converges to optimality in at most N iterations. In Lemma 6, we prove that $\Omega(N)$ iterations are required for convergence in the worst case.

Lemma 5 *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 $M[\beta_x].d$

Iteration Count		C[1]	C[3]	C[5]		C[N-5]	C[N-3]	C[N-1]		C[2]	C[4]	C[6]		C[N-4]	C[N-2]	C[N]
Initialization		1	1	1		1	1	1		1	1	1		1	1	1
1	**Stage 1**	1	2	3		$\frac{N}{2}-2$	$\frac{N}{2}-1$	$\frac{N}{2}$		2	3	4		$\frac{N}{2}-1$	$\frac{N}{2}$	$\frac{N}{2}+1$
	Stage 2	1	3	4		$\frac{N}{2}-1$	$\frac{N}{2}$	$\frac{N}{2}+1$		2	3	4		$\frac{N}{2}-1$	$\frac{N}{2}$	$\frac{N}{2}+1$
2	**Stage 1**	1	3	4	⋯	$\frac{N}{2}-1$	$\frac{N}{2}$	$\frac{N}{2}+1$	⋯	2	4	5	⋯	$\frac{N}{2}$	$\frac{N}{2}+1$	$\frac{N}{2}+2$
	Stage 2	1	3	5		$\frac{N}{2}$	$\frac{N}{2}+1$	$\frac{N}{2}+2$		2	4	5		$\frac{N}{2}$	$\frac{N}{2}+1$	$\frac{N}{2}+2$
3	**Stage 1**	1	3	5		$\frac{N}{2}$	$\frac{N}{2}+1$	$\frac{N}{2}+2$		2	4	6		$\frac{N}{2}+1$	$\frac{N}{2}+2$	$\frac{N}{2}+3$
	Stage 2	1	3	5		$\frac{N}{2}+1$	$\frac{N}{2}+2$	$\frac{N}{2}+3$		2	4	6		$\frac{N}{2}+1$	$\frac{N}{2}+2$	$\frac{N}{2}+3$
⋮																
$\frac{N}{2}-2$	**Stage 1**	1	3	5		$N-5$	$N-4$	$N-3$		2	4	6		$N-4$	$N-3$	$N-2$
	Stage 2	1	3	5		$N-5$	$N-3$	$N-2$		2	4	6		$N-4$	$N-3$	$N-2$
$\frac{N}{2}-1$	**Stage 1**	1	3	5		$N-5$	$N-3$	$N-2$		2	4	6		$N-4$	$N-2$	$N-1$
	Stage 2	1	3	5		$N-5$	$N-3$	$N-1$		2	4	6		$N-4$	$N-2$	$N-1$
$\frac{N}{2}$	**Stage 1**	1	3	5	⋯	$N-5$	$N-3$	$N-1$	⋯	2	4	6	⋯	$N-4$	$N-2$	N
	Stage 2	1	3	5		$N-5$	$N-3$	$N-1$		2	4	6		$N-4$	$N-2$	N
$\frac{N}{2}+1$	**Stage 1**	1	3	5		$N-5$	$N-3$	$N-1$		2	4	6		$N-4$	$N-2$	N
	Stage 2	1	3	5		$N-5$	$N-3$	$N-1$		2	4	6		$N-4$	$N-2$	N

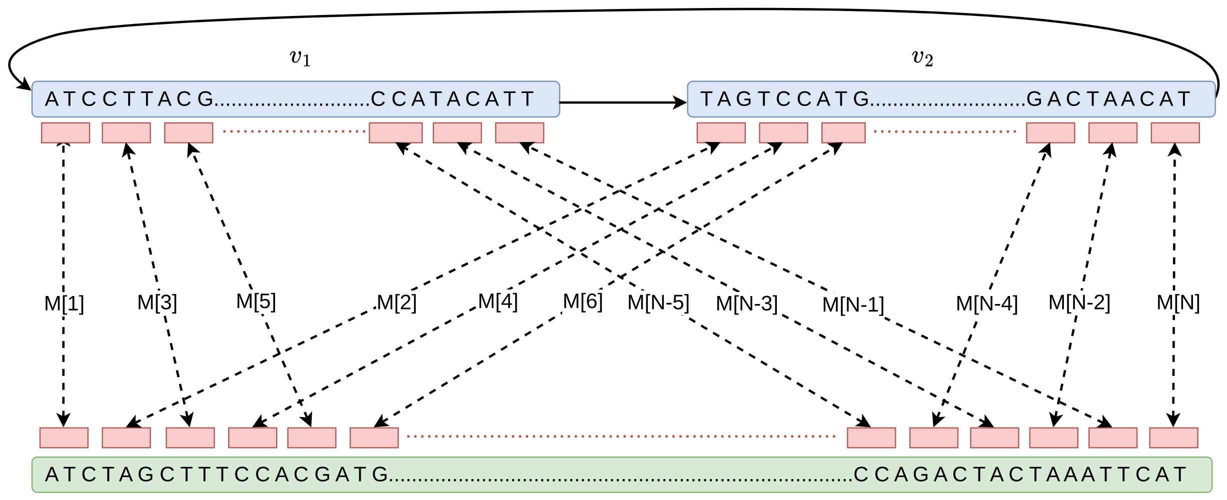
v_1
v_2
ATCCTTACG..........................CCATACATT
TAGTCCATG..........................GACTAACAT
M[1]
M[3]
M[5]
M[2]
M[4]
M[6]
M[N-5]
M[N-3]
M[N-1]
M[N-4]
M[N-2]
M[N]
ATCTAGCTTTCCACGATG..........................CCAGACTACTAAATTCAT

Fig. 4 A worst-case example for Algorithm 3 where it requires $\Omega(N)$ iterations to converge (Lemma 6). We show a step-by-step progress of the algorithm with each iteration. The table shows the values in array C after each iteration

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 in-place 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 6 *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 Fig. 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 $(M[1], M[2], M[3], \ldots, M[N-1], M[N])$. Similarly, the optimal chain ending at anchor $M[i]$ is $(M[1], M[2], M[3], \ldots, M[i])$.
Suppose our path cover comprises single path $v_1 \rightarrow v_2$. Each anchor is assigned a *weight* of one unit (a unit represents a multiplicative constant). In this example, each iteration of the chaining algorithm can be conceptually divided into two stages. Stage 1 corresponds to the first round of updates to array C in the order $C[1], C[3], C[5], \ldots, C[N-1], C[2], \ldots, C[N-2], C[N]$. These updates are caused due to tuples in Lines 7-8 of Algorithm 3. Stage 2 corresponds to the second round of updates to $C[1], C[3], C[5], \ldots, C[N-3], C[N-1]$ caused by tuples in Line 12. After the first iteration of the algorithm, the maximum score in array C is $\frac{N}{2}+1$. In each subsequent iteration, the maximum score increases by 1. The scores converge at iteration $\frac{N}{2}+1$. □

Table 1 Properties of four cyclic pangenome graphs 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

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 [26], Minigraph [28], and GraphChainer [29] 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 $u \rightarrow v \in E$, we add the complementary edge $\bar{v} \rightarrow \bar{u}$. 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 [28]. 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

Table 2 All four graphs have multiple weakly connected components

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	10.9 k/309.6 k	2.0/4	2.0/5	2.3/77
40H	1–36	10.9 k/309.6 k	2.0/4	2.0/5	2.4/72
80H	1–49	10.8 k/309.4 k	3.0/4	3.0/5	2.4/61
95H	1–59	10.8 k/309.4 k	3.0/4	3.0/5	2.4/64

Therefore, the size of the identified path cover of each graph is presented as a range. The other columns show the count of iterations used by our iterative algorithms for graph preprocessing and co-linear chaining (Algorithms 1, 2, 3). The iteration count statistics were gathered while aligning simulated long reads to cyclic pangenome graphs

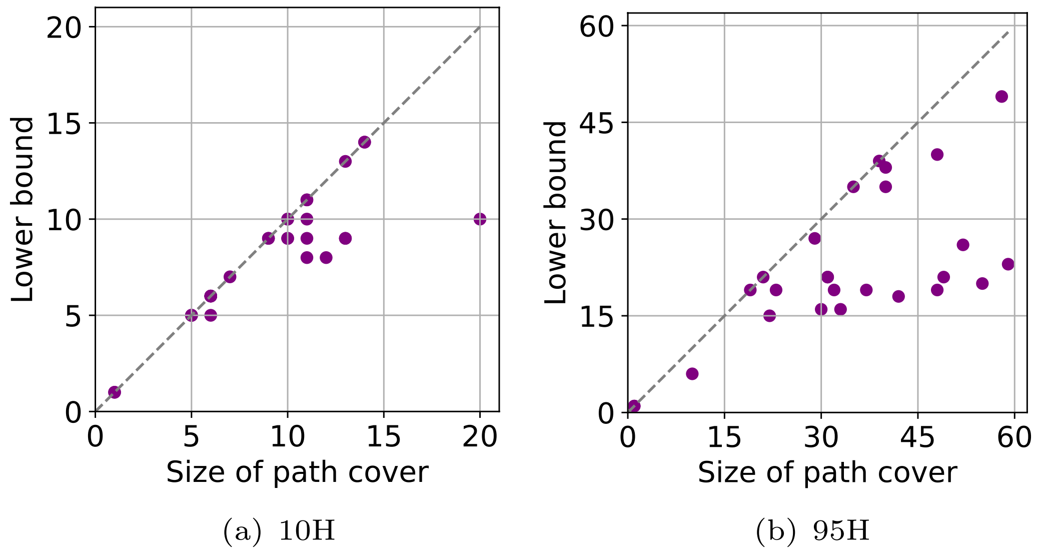

Fig. 5 A comparison of the size of the computed path cover and the lower bound on the size of the minimum path cover for each component of graphs (**a**) 10H and (**b**) 95H. Graph 10H has 30 weakly connected components. Graph 95H has 26 weakly connected components (Table 1)

during graph construction. Similar to [26, 29], 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 [24, 26, 28]. 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 [46]. We used seeding and extension code from Minigraph [28]. Seeding is done by identifying minimizer matches [47] between vertex labels of the graph and the read. The extension code produces the final base-to-base alignment by joining the chained anchors [48]. We used code from GraphChainer [29] to compute the minimum path cover of a DAG and range queries.

Co-linear chaining for sequence-to-graph alignment is generally slower than chaining between two sequences. If the optimal alignment of a read is unlikely to span more than one vertex, then it may be more efficient to use sequence-to-sequence chaining algorithm for that read. Following this intuition, we have also implemented a *hybrid method* that identifies a subset of reads which are 'easy to align' by first aligning all reads using Minigraph. Only those reads are aligned using PanAligner for which either Minigraph outputs an alignment spanning more than one vertex or Minigraph outputs a split-read alignment.

Experiments

Benchmark datasets

We constructed four cyclic pangenome graphs by using subsets of publicly available 95 haplotype-resolved human genome assemblies [4, 49]. These graphs were generated using Minigraph v0.20 [28]. We used CHM13 human genome assembly [49] 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 3 A comparison of the performance of long-read aligners using the 10H graph

	PanAligner	Hybrid method	Minigraph	GraphAligner
Indexing time (sec)	96	136	**66**	238
Alignment time (sec)	2924	605	**50**	4928
Memory usage (GB)	23.14	24.68	23.18	**23.10**
Unaligned reads	1.18%	1.18%	2.17%	**0%**
Incorrectly Aligned reads	0.79%	**0.76%**	1.19%	1.47%
Unaligned reads (MQ≥10)	3.51%	3.51%	5.85%	**0.78%**
Incorrectly Aligned reads (MQ≥10)	0.20%	**0.17%**	0.32%	0.91%

Best numbers are highlighted in bold

MQ stands for mapping quality

Table 4 A comparison of the performance of long-read aligners using the 95H graph

	PanAligner	Hybrid method	Minigraph	GraphAligner
Indexing time (sec)	83	176	**77**	272
Alignment time (sec)	9276	1899	**60**	5170
Memory usage (GB)	43.6	43.25	**24.74**	26.1
Unaligned reads	1.60%	1.60%	2.24%	**0%**
Incorrectly Aligned reads	1.28%	**1.21%**	1.93%	2.98%
Unaligned reads (MQ≥10)	4.20%	4.21%	6.21%	**0.84%**
Incorrectly Aligned reads (MQ≥10)	0.57%	**0.49%**	0.85%	2.33%

Best numbers are highlighted in bold

MQ stands for mapping quality

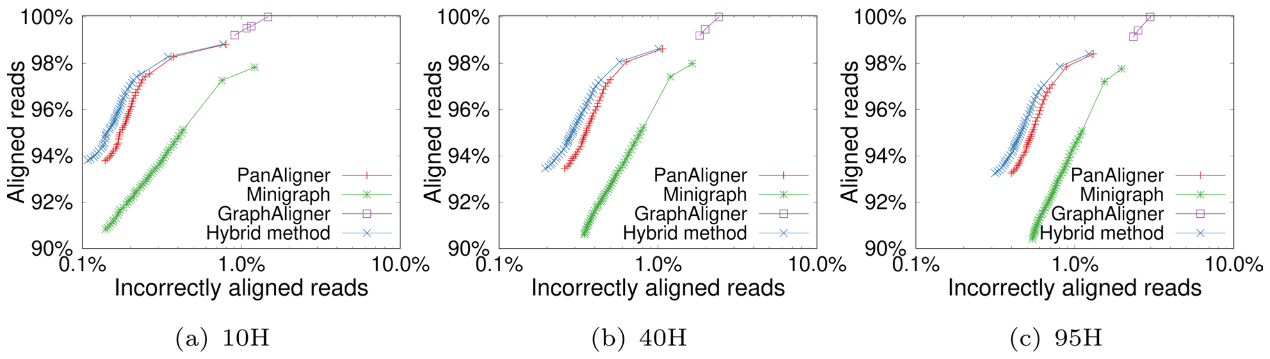

Fig. 6 The plots in panels (**a**), (**b**) and (**c**) show the fraction of aligned reads and the accuracy obtained by using all the aligners on graphs 10H, 40H, and 95H, respectively. These plots were generated by varying mapping quality cutoffs from 0 to 60. X-axis in these plots uses a logarithmic scale to indicate the percentage of incorrectly aligned reads

Evaluation methodology

We simulated long reads using PBSIM2 v2.0.1 [50] 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 [24, 26, 28]. 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` [24] 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 Sect. 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. For each component, we show a comparison of the size of the computed path cover with the lower bound on the minimum path cover size (Fig. 5). The results show that our heuristic is effective in finding a path cover whose size is close to optimal.

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

Table 5 A comparison of the performance of long-read aligners using the 95H-DAG graph

	PanAligner	Minichain	Minigraph	GraphAligner	GraphChainer
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%

Best numbers are highlighted in bold

MQ stands for mapping quality

(Algorithms 1–2) and also report them as a range for each graph. These algorithms compute *last2reach* and D arrays. Observe that the iteration count is significantly smaller in practice than the proven upper limit of $|V|$ (Lemma 3). 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 5.

Alignment of simulated reads to cyclic graphs

We assessed the performance of PanAligner and the Minigraph-PanAligner hybrid method against two existing sequence-to-graph aligners, Minigraph v2.20 [28] and GraphAligner v1.0.17b [30], that can handle cycles. Unlike PanAligner, Minigraph and GraphAligner use heuristics to join anchors. Minichain [26, 51] and GraphChainer [29] 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. Next, the hybrid method offers slightly better accuracy than PanAligner because the hybrid method uses Minigraph heuristics to align the reads which are sampled from a single vertex. Minigraph is built on top of Minimap2 [24], which is a highly optimized sequence-to-sequence aligner. We show the comparison plots in Fig. 6.

PanAligner, Minigraph and the Minigraph-PanAligner hybrid method left a small fraction of reads unaligned. This may be because (i) PanAligner and Minigraph drop high-frequency minimizer matches during the seeding step, 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 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. The runtimes of PanAligner and GraphAligner are in the same order of magnitude. The Minigraph-PanAligner hybrid method is about 5$\times$ faster than using PanAligner alone. This is because, for 95H graph, PanAligner was used to align only 12% of the total reads; the alignments for rest of the 'easy to align' reads were obtained using Minigraph. Overall, the hybrid method produces the best alignment accuracy among the four methods, and its runtime is practical for large whole-genome sequencing data.

We observe a consistent drop in alignment accuracy of all four alignment methods with increasing haplotype count (Fig. 6). This is likely because the number of combinatorial paths to which a read can align grows exponentially with respect to the haplotype count.

Alignment of simulated reads to acyclic graphs

We also tested PanAligner for acyclic pangenome graphs. We followed the same procedure as [26] 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 [28]. 95H-DAG has 1.2M vertices and 1.8M edges. We also include Minichain v1.0 [26] and GraphChainer v1.0.2 [29] 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 [26]. 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.

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 [4, 10]. Existing alignment software designed for cyclic graphs are based on heuristics to join anchors [28, 30]. 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 [26, 29, 38, 52].

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.Although PanAligner is slower than heuristic methods, one can use PanAligner for only those fraction of reads that are predicted to have optimal alignments spanning more than one vertex.

In our formulation, we did not allow anchors to overlap with each other. We also did not allow an anchor to span two or more vertices in a graph for simplicity; but the proposed ideas can be generalized. PanAligner software borrows seeding logic from Minigraph [28], 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., [4, 53, 54]. 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 [51, 55, 56]. 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.

Acknowledgements
The authors thank Manuel Cáceres, Shravan Mehra, Sunil Chandran and Rob Patro for useful suggestions. Daanish Mahajan and Sudhanva Shyam Kamath helped with the software implementation.

Author contributions
All authors contributed to the conception and analysis of the ideas presented in this study, as well as the drafting and finalising of this document.

Funding
This research is supported in part by funding from the 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.

Availability of data and materials
Not applicable.

Declarations

Competing interests
The authors declare no competing interests.

Received: 21 November 2023 Accepted: 2 January 2024
Published online: 27 January 2024

References
1. Eggertsson HP, Jonsson H, Kristmundsdottir S, et al. Graphtyper enables population-scale genotyping using pangenome graphs. Nat Genet. 2017;49(11):1654–60.
2. Ekim B, Berger B, Chikhi R. Minimizer-space de bruijn graphs: whole-genome assembly of long reads in minutes on a personal computer. Cell Syst. 2021;12(10):958–68.
3. Garrison E, Sirén J, Novak AM, Hickey G, Eizenga JM, Dawson ET, Jones W, Garg S, Markello C, Lin MF, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. Nat Biotechnol. 2018;36(9):875–9. https://doi.org/10.1038/nbt.4227.
4. Liao W-W, Asri M, Ebler J, Doerr D, Haukness M, Hickey G, Lu S, Lucas JK, Monlong J, Abel HJ, et al. A draft human pangenome reference. Nature. 2023;617(7960):312–24.
5. Sirén J, Monlong J, Chang X, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. Science. 2021;374(6574):8871.
6. Wang T, Antonacci-Fulton L, Howe K, et al. The human pangenome project: a global resource to map genomic diversity. Nature. 2022;604(7906):437–46.
7. Zhou Y, Zhang Z, Bao Z, Li H, Lyu Y, Zan Y, Wu Y, Cheng L, Fang Y, Wu K, et al. Graph pangenome captures missing heritability and empowers tomato breeding. Nature. 2022;606(7914):527–34.
8. Dolzhenko E, Deshpande V, Schlesinger F, Krusche P, Petrovski R, Chen S, Emig-Agius D, Gross A, Narzisi G, Bowman B, et al. Expansionhunter: a sequence-graph-based tool to analyze variation in short tandem repeat regions. Bioinformatics. 2019;35(22):4754–6.
9. Lu TY, et al. Profiling variable-number tandem repeat variation across populations using repeat-pangenome graphs. Nat Commun. 2021;12(1):4250.
10. Paten B, Novak AM, Eizenga JM, Garrison E. Genome graphs and the evolution of genome inference. Genome Res. 2017;27(5):665–76.
11. Gao Y, Yang X, Chen H, Tan X, Yang Z, Deng L, Wang B, Kong S, Li S, Cui Y, et al. A pangenome reference of 36 Chinese populations. Nature 2023;1–10.

12. Baaijens JA, Bonizzoni P, Boucher C, Della Vedova G, Pirola Y, Rizzi R, Sirén J. Computational graph pangenomics: a tutorial on data structures and their applications. Nat Comput. 2022;1–28.
13. Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. Brief Bioinform. 2018;19(1):118–35.
14. Cheng H, Asri M, Lucas J, Koren S, Li H. Scalable telomere-to-telomere assembly for diploid and polyploid genomes with double graph. arXiv preprint arXiv:2306.03399. 2023.
15. Garg S, Rautiainen M, Novak AM, et al. A graph-based approach to diploid genome assembly. Bioinformatics. 2018;34(13):105–14.
16. Rautiainen M, Nurk S, Walenz BP, Logsdon GA, Porubsky D, Rhie A, Eichler EE, Phillippy AM, Koren S. Telomere-to-telomere assembly of diploid chromosomes with verkko. Nat Biotechnol. 2023;1–9.
17. Luo X, Kang X, Schönhuth A. Vechat: correcting errors in long reads using variation graphs. Nat Commun. 2022;13(1):6657.
18. Salmela L, Rivals E. Lordec: accurate and efficient long read error correction. Bioinformatics. 2014;30(24):3506–14.
19. Jain C, Zhang H, Gao Y, Aluru S. On the complexity of sequence-to-graph alignment. J Comput Biol. 2020;27(4):640–54. https://doi.org/10.1089/cmb.2019.0066.
20. Navarro G. Improved approximate pattern matching on hypertext. Theoret Comput Sci. 2000;237(1–2):455–63.
21. 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, 2015;pp. 51–58.
22. Gibney D, Thankachan SV, Aluru S. 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, 2022;pp. 263–278. Springer.
23. Jain C, Rhie A, Hansen NF, Koren S, Phillippy AM. Long-read mapping to repetitive reference sequences using winnowmap2. Nat Methods. 2022;1–6.
24. Li H. Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics. 2018;34(18):3094–100. https://doi.org/10.1093/bioinformatics/bty191.
25. Sahlin K, Baudeau T, Cazaux B, Marchet C. A survey of mapping algorithms in the long-reads era. bioRxiv. 2022.
26. Chandra G, Jain C. 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, 2023;pp. 58–73. Springer.
27. Dvorkina T, Antipov D, Korobeynikov A, Nurk S. Spaligner: alignment of long diverged molecular sequences to assembly graphs. BMC Bioinformatics. 2020;21(12):1–14.
28. Li H, Feng X, Chu C. The design and construction of reference pangenome graphs with minigraph. Genome Biol. 2020;21(1).
29. Ma J, Cáceres M, Salmela L, Mäkinen V, Tomescu AI. Chaining for accurate alignment of erroneous long reads to acyclic variation graphs. Bioinformatics. 2023;39(8):460.
30. Rautiainen M, Marschall T. Graphaligner: rapid and versatile sequence-to-graph alignment. Genome Biol. 2020;21(1):253.
31. Abouelhoda M, Ohlebusch E. Chaining algorithms for multiple genome comparison. J Discrete Algorithms. 2005;3(2–4):321–41.
32. Eppstein D, Galil Z, Giancarlo R, Italiano GF. Sparse dynamic programming i: linear cost functions. J ACM. 1992;39(3):519–45.
33. Eppstein D, Galil Z, Giancarlo R, Italiano GF. Sparse dynamic programming ii: convex and concave cost functions. J ACM. 1992;39(3):546–67.
34. Jain C, Gibney D, Thankachan SV. Algorithms for colinear chaining with overlaps and gap costs. J Comput Biol. 2022;29(11):1237–51.
35. Mäkinen V, Sahlin K. Chaining with overlaps revisited. In: 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020). 2020. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
36. Myers G, Miller W. Chaining multiple-alignment fragments in sub-quadratic time. In: SODA, 1995;vol. 95, pp. 38–47.
37. Otto C, Hoffmann S, Gorodkin J, Stadler PF. Fast local fragment chaining using sum-of-pair gap costs. Algorithms Mol Biol. 2011;6(1):4.
38. Mäkinen V, Tomescu AI, Kuosmanen A, Paavilainen T, Gagie T, Chikhi R. Sparse dynamic programming on DAGs with small width. ACM Trans Algorithms (TALG). 2019;15(2):1–21.
39. Rizzo N, Cáceres M, Mäkinen V. Chaining of maximal exact matches in graphs. In: String Processing and Information Retrieval: 30th International Symposium. SPIRE 2023, Pisa, Italy, September 26–28, 2023, Proceedings. Berlin, Heidelberg: Springer; 2023. p. 353–66.
40. Cáceres M, Cairo M, Mumey B, Rizzi R, Tomescu AI. 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), 2022;pp. 359–376. SIAM.
41. Nurk S, Koren S, Rhie A, Rautiainen M, et al. The complete sequence of a human genome. Science. 2022;376(6588):44–53. https://doi.org/10.1126/science.abj6987.
42. Mäkinen V, Belazzougui D, Cunial F, Tomescu AI. Genome-scale Algorithm Design. Cambridge University Press, 2015.
43. Eades P, Lin X, Smyth WF. A fast and effective heuristic for the feedback arc set problem. Inf Process Lett. 1993;47(6):319–23.
44. Tarjan R. Depth-first search and linear graph algorithms. SIAM J Comput. 1972;1(2):146–60.
45. Ntafos SC, Hakimi SL. On path cover problems in digraphs and applications to program testing. IEEE Trans Software Eng. 1979;5:520–9.
46. Li H, Ruan J, Durbin R. Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Res. 2008;18(11):1851–8.
47. Roberts M, Hayes W, Hunt BR, Mount SM, Yorke JA. Reducing storage requirements for biological sequence comparison. Bioinformatics. 2004;20(18):3363–9.
48. Zhang H, Wu S, Aluru S, Li H. Fast sequence to graph alignment using the graph wavefront algorithm. arXiv preprint arXiv:2206.13574. 2022.
49. Nurk S, Koren S, Rhie A, Rautiainen M, Bzikadze AV, Mikheenko A, Vollger MR, Altemose N, Uralsky L, Gershman A, et al. The complete sequence of a human genome. Science. 2022;376(6588):44–53.
50. Ono Y, Asai K, Hamada M. Pbsim2: a simulator for long-read sequencers with a novel generative model of quality scores. Bioinformatics. 2021;37(5):589–95.
51. Chandra G, Jain C. Haplotype-aware sequence-to-graph alignment. bioRxiv. 2023. https://doi.org/10.1101/2023.11.15.566493.
52. Ma J. Co-linear chaining on graphs with cycles. Master's thesis, University of Helsinki, Faculty of Science. 2021. http://hdl.handle.net/10138/330781.
53. Garrison E, Guarracino A, Heumos S, Villani F, Bao Z, Tattini L, Hagmann J, Vorbrugg S, Marco-Sola S, Kubica C, et al. Building pangenome graphs. bioRxiv, 2023–04. 2023.
54. Hickey G, Monlong J, Ebler J, Novak AM, Eizenga JM, Gao Y, Marschall T, Li H, Paten B. Pangenome graph construction from genome alignments with minigraph-cactus. Nat Biotechnol. 2023;1–11.
55. Mokveld T, Linthorst J, Al-Ars Z, Holstege H, Reinders M. Chop: haplotype-aware path indexing in population graphs. Genome Biol. 2020;21:1–16.
56. Sirén J, Garrison E, Novak AM, Paten B, Durbin R. Haplotype-aware graph indexes. Bioinformatics. 2020;36(2):400–7.

Publisher's Note