GraphChainer: Co-linear Chaining for Accurate Alignment of Long Reads to Variation Graphs*

Jun Ma [1,†], Manuel Cáceres [1,†], Leena Salmela [1], Veli Mäkinen [1] and Alexandru I. Tomescu [1]

[1]Department of Computer Science, University of Helsinki, Finland
{jun.ma, manuel.caceresreyes, leena.salmela, veli.makinen, alexandru.tomescu}@helsinki.fi

Abstract

Aligning reads to a variation graph is a standard task in pangenomics, with downstream applications in e.g., improving variant calling. While the `vg toolkit` (Garrison et al., *Nature Biotechnology*, 2018) is a popular aligner of short reads, `GraphAligner` (Rautiainen and Marschall, *Genome Biology*, 2020) is the state-of-the-art aligner of long reads. `GraphAligner` works by finding candidate read occurrences based on *individually* extending the best seeds of the read in the variation graph. However, a more principled approach recognized in the community is to co-linearly chain *multiple* seeds. We present a new algorithm to co-linearly chain a set of seeds in an acyclic variation graph, together with the first efficient implementation of such a co-linear chaining algorithm into a new aligner of long reads to variation graphs, `GraphChainer`. Compared to `GraphAligner`, at a normalized edit distance threshold of 40%, it aligns 9% to 12% more reads, and 15% to 19% more total read length, on real PacBio reads from human chromosomes 1 and 22. On both simulated and real data, `GraphChainer` aligns between 97% and 99% of all reads, and of total read length. At the more stringent normalized edit distance threshold of 30%, `GraphChainer` aligns up to 29% more total real read length than `GraphAligner`.
`GraphChainer` is freely available at `https://github.com/algbio/GraphChainer`

1 Introduction

Motivation. Variation graphs are a popular representation of all the genomic diversity of a population [6, 11, 28]. While a single reference sequence can be represented by a single labeled path, a variation graph encodes each variant observed in the population via an "alternative" path between its start and end genomic positions in the reference labeled path. For example, the popular `vg toolkit` [14] builds variation graphs either from a reference genome and a set of variants, or from a set of reference genomes, and also supports aligning reads to them.

A key advantage of a variation graph is that it allows for recombinations of two or more variants, that are not necessarily present in any of the existing reference sequences used to build the graph. Namely, the paths of a variation graph now spell novel haplotypes recombining different variants, thus improving the accuracy of downstream applications, such as variant calling [14, 41, 16, 17, 38]. In fact, it has been observed [14] that mapping reads with `vg` to a variation graph leads to more accurate results than aligning them to a single reference (since it removes the reference bias). The alignment method of `vg` was developed for short reads, and while it can run also on the more-erroneous long reads, it decreases its performance (computational resources and accuracy) significantly. To the best of our knowledge, the only sequence-to-graph aligners

*This work was partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO) and partially by the Academy of Finland (grants No. 322595, 328877, 308030).

†Shared first author contribution.

tailored to long reads are `SPAligner` [10], designed for assembly graphs, `GraphAligner` [31], designed for both assembly and variation graphs, and the recent extension of `AStarix` for long reads [19]. `GraphAligner` is the state-of-the-art aligner for long reads at the chromosome level, allowing for faster and more accurate alignments.

Background. At the core of any sequence-to-graph aligner is the so-called *string matching in labeled graphs* problem, asking for a path of a node-labeled graph (e.g., a variation graph) whose spelling matches a given pattern. If allowing for approximate string matching under the minimum number of edits in the sequence, then the problem can be solved in quadratic time [2], and extensions to consider affine gap costs [21] and various practical optimizations [19, 18, 30, 20] were developed later.

Since, conditioned on SETH, no strongly subquadratic-time algorithm for edit distance on sequences can exist [3], these sequence-to-graph alignment algorithms are worst-case optimal up to subpolynomial improvements. This lower bound holds even if requiring only *exact* matches, the graph is acyclic, i.e. a DAG [12, 15], and we allow any polynomial-time indexing of the graph [13].

Given the hardness of this problem, current tools such as `vg` and `GraphAligner` employ various heuristics and practical optimizations to approximate the sequence-to-graph alignment problem, such as partial order alignment [23], as in the case of `vg`, and seed-and-extend based on minimizers [32], as in the case of `GraphAligner`. More specifically, `GraphAligner` finds the minimizers of the read that have occurrences in the graph, clusters and ranks these occurrences, similarly to `minimap` [25], and then tries to extend the best clusters using an optimized *banded* implementation of the bit-parallel edit distance computation for sequence-to-graph alignment [30]. This strategy was shown to be effective in mapping long reads, since minimizers between erroneous positions can still have an exact occurrence in the graph. However, it is simple to observe that it can lead to wrong locations in the variation graph at which to extend a *single* seed (cluster) into a full occurrence of the long read. For example, the seeds may be clustered in several regions of the graph and be separated in distance. Extending from one seed (cluster) through an erroneous zone to reach the next relatively accurate region would be hard in this case. Hence, for a long read, such an aligner may find several short alignments covering different parts of the read, but not a long alignment of the entire read. Furthermore, a seed may have many false alignments in the graph that are not useful in the alignment of the long read, but which can only be discarded when looking at its position compared to other seed hits.

A standard way to capture the global relation between the seed hits is by the *co-linear chaining* problem [29], originally defined for two sequences as follows. Given a set of *anchors* consisting of pairs of intervals in the two sequences (e.g., various types of matching substrings, such as minimizers), the goal is to find a *chain* of anchors such that their intervals appear in the same order in both sequences. If the goal is to find a chain of maximum coverage in one of the sequences, the problem can be solved in time $O(N \log N)$ [1, 36], where N is the number of anchors. Co-linear chaining is succesfully used by the popular read-to-reference sequence aligner `minimap2` [26].

The co-linear chaining problem can be naturally extended to a sequence and a labeled graph and has been previously studied for DAGs [27, 22], but now considering the anchors to be pairs of a path in the graph and a matching interval in the sequence. `GraphAligner` refers to chaining on DAGs as a principled way of chaining seeds [31, p. 16], but does not implement co-linear chaining. Also `SPAligner` appears to implement a version of co-linear chaining between a sequence and an assembly graph. It uses the BWA-MEM library [24] to compute anchors between the long read and individual edge labels of the assembly graph, and then extracts "the heaviest chain of compatible anchors using dynamic programming" [10, p.3–4]. Their compatibility relation appears to be defined as in the co-linear chaining problem, and further requires that the distance between the anchor paths is relatively close to the distance between anchor intervals in the sequence. However, the overall running time of computing distances to check compatibility is quadratic [10, Appendix].

If the co-linear chaining problem is restricted to DAGs, it can be solved in time $O(kN \log N + k|V|)$, after a preprocessing step taking $O(k(|V| + |E|) \log |V|)$ time [27] when suffix-prefix overlaps between anchor paths are not allowed. Here, the input is a DAG $G = (V, E)$ of *width* k, which is defined as the cardinality of a minimum-sized set of paths of the DAG covering every node at least once (a *minimum path cover*, or MPC). Thus, if k is constant, this algorithm matches the running time of co-linear chaining on two

sequences, plus an extra term that grows linearly with the size of the graph. Even though there exists an initial implementation of these ideas tested on (small) RNA splicing graphs [22], it does not scale to large graphs, such as variation graphs. Moreover, in the same publication, the authors show how to solve the general problem when suffix-prefix path overlaps of any length are allowed with an additional running time of $O(L \log^2 |V|)$ or $O(L + \#o)$[1], by using advanced data structures (FM-index, two-dimensional range search trees, generalized suffix tree).

Contributions. In this paper we compute for the first time the width of variation graphs of all human chromosomes, which we observe to be at most 9 (see Table 4 in the Appendix), further motivating the idea of parameterizing the complexity of the co-linear chaining problem on the width of the variation graph.

We first present an algorithm solving co-linear channing on a DAG, when allowing one-node suffix-prefix overlaps between anchor paths. Our solution is an extension of the previous $O(k(|V| + |E|) \log |V| + kN \log N)$ time solution [27], to efficiently consider the one-node overlapping case, but without the need of advanced data structures (Section 2.3). Moreover, we show how to divide the running time of our algorithm into $O(k^3|V| + k|E|)$ for pre-processing the graph [4, 27], and $O(kN \log kN)$ for solving co-linear chaining (Section 2.4). That is, for constant width graphs (such as variation graphs), our solution takes linear time to preprocess the graph plus $O(N \log N)$ time to solve co-linear chaining, removing the previous dependency on the graph size and matching the time complexity of the problem between two sequences. Since in practice N is usually much smaller than the DAG size, this result allows for a more efficient implementation for the algorithm in practice.

On the one hand, allowing one-node suffix-prefix overlaps expands the solution space, thus potentially improving the final solution. In fact, we empirically show that most of the optimal chains do contain one-node suffix-prefix overlaps (Appendix). On the other hand, as a theoretical contribution, we show that allowing such overlaps suffices to solve co-linear chaining in graphs with cycles in the same running time as our solution for DAGs (Appendix).

We then implement our algorithm for DAGs into `GraphChainer`, a new sequence-to-variation-graph aligner. On both simulated and real data, we show that `GraphChainer` allows for significantly more accurate alignments of long reads than the state-of-art aligner `GraphAligner`. In particular, when classifying a read as having a *good alignment* if the edit distance between the read and its best alignment is allowed to be at most 40% of the read length, `GraphChainer` aligns between 9% to 12% more real reads, of 15% to 19% more total length, on human chromosomes 1 and 22. At the same threshold for edit distance, `GraphChainer` produces a good alignment of 97%–99% of all reads, or of total read length.

While this significant increase in accuracy comes with an increase in computational requirements, `GraphChainer` can run in parallel and its requirements even on chromosome 1 are still within the capabilities of a modern high-performance computer. We also observe that the most time-intensive part of `GraphChainer` is obtaining the anchors for the input to co-linear chaining. This is currently implemented by aligning shorter fragments of the read using `GraphAligner`, however given the modularity of this step, in the future other methods for finding anchors could be explored, such as using a fast short-read graph aligner or recent developments in MEMs [33].

2 Methods

2.1 Problem definition

Given a (directed) graph $G = (V, E)$, an *anchor* A is a tuple $(W = (u, \ldots, v), I = [x, y])$, such that W is a walk of G starting in u and ending in v, and $I = [x, y]$ (with $x \leq y$ non-negative integers) is the interval of integers between x and y (both inclusive). We interpret I as an interval in the read matching the label of the walk W in G. If A is an anchor, we denote by $A.W, A.u, A.v, A.I, A.x$ and $A.y$ its walk, walk starting point, walk endpoint, interval, interval starting point and interval endpoint, respectively. Given a set of anchors $\mathcal{A} = \{A_1, \ldots, A_N\}$, a *chain* $\mathcal{C}$ *of anchors* (or just *chain*) of $\mathcal{A}$ is a sequence of anchors $A_{i_1}, \ldots, A_{i_q}$, such

[1] L corresponds to the sum of the path lengths, whereas $\#o$ is the number of overlaps between paths.

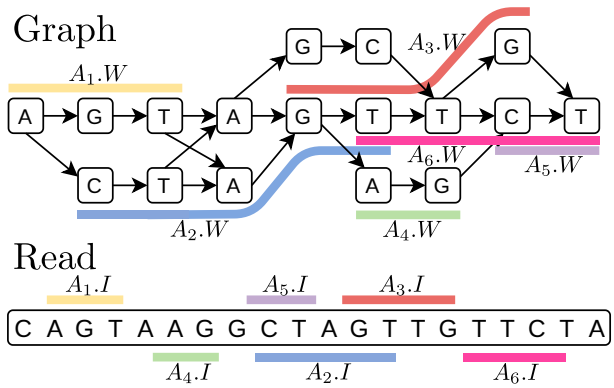

Figure 1: An example for Problem 2 consisting of a DAG, a read and a set of six anchors (paths in the graphs, paired with intervals in the read), shown here in different colors. The sequence A_1, A_4, A_5 is a chain of coverage 7. The sequence A_2, A_3 is not a chain because their paths have a suffix-prefix overlap of more than one node. The sequence A_2, A_5 is also not a chain because their intervals are not in order. Anchors A_1 and A_2 cannot be chained because $A_1.v$ cannot reach $A_2.v$. The optimum chain is A_2, A_6, of coverage 9.

that for all $j \in \{1, \ldots, q-1\}$, A_{i_j} *precedes* $A_{i_{j+1}}$, where the precedence relation is a partial order of $\mathcal{A}$ that corresponds to the conjunction of the precedence of anchor walks and of anchor intervals (hence *co-linear*). We will tailor the notion of precedence between walks and intervals for every version of the problem. We define the general co-linear chaining problem as follows:

Problem 1 (Co-linear Chaining (CLC))**.** *Given a graph $G = (V, E)$ and a set $\mathcal{A}$ of N anchors, find a chain $\mathcal{C} = A_{i_1}, \ldots, A_{i_q}$ of $\mathcal{A}$ maximizing $cov(\mathcal{C}) := |\bigcup_{j=1}^{q} A_{i_j}.I|$.*

Note that, although the ultimate objective is to align the read to the variation graph, co-linear chaining can be defined (and solved) independent of the string labels in these objects. Co-linear chaining has been solved [27] when the input graph is restricted to be a DAG, interval precedence is defined as integer inequality of interval endpoints ($A_{i_j}.y < A_{i_{j+1}}.y$), and path[2] precedence is defined as strict reachability of path extremes (namely, $A_{i_j}.v$ *strictly* reaches $A_{i_{j+1}}.u$, i.e., with a path of at least one edge) in time $O(k(|V|+|E|)\log|V| + kN\log N)$. If the precedence relation is relaxed to allow suffix-prefix overlaps of paths, then their solution has an additional $O(L\log^2|V|)$ or $O(L+\#o)$[3] term in the running time. We will show how to solve co-linear chaining in time $O(k(|V|+|E|)\log|V| + kN\log N)$ when path precedence allows a one-node suffix-prefix overlap. More precisely, we solve the following problem (see Fig. 1 for an example):

Problem 2 (One-node suffix-prefix overlap CLC)**.** *Same as Problem 1, but restricting G to be a DAG, and such that for every $A, A' \in \mathcal{A}$, we define A* precedes *A' iff $A.y < A'.y$ and $A.v$ reaches $A'.u$ (possibly with an empty path, i.e., $A.v = A'.u$).*

Removing the *strict* (reachability) from the path precedence allows one-node suffix-prefix overlaps, and no more than that (since G is a DAG).

2.2 Overview of the existing solution (without suffix-prefix overlaps)

The previous solution [27] computes an MPC of G in time $O(k(|V|+|E|)\log|V|)$, which can be reduced by a recent result [4] to parameterized linear time $O(k^3|V|+|E|)$. Then, the anchors from $\mathcal{A}$ are sorted according to a topological order of its path endpoints ($A.v$) into $A_1, \ldots, A_N$. The algorithm uses a dynamic programming algorithm to compute for every $j \in \{1, \ldots, N\}$ the array:

$$C[j] = \max\{cov(\mathcal{C}) \mid \mathcal{C} \text{ is a chain of } \{A_1, \ldots, A_j\} \text{ ending at } A_j\}.$$

[2]Recall that in a DAG all walks are paths (i.e., do not repeat nodes).
[3]L corresponds to the sum of the path lengths, whereas $\#o$ is the number of overlaps between paths.

Since chains (for this version of the problem) do not have suffix-prefix overlaps, they are all subsequences of $A_1, \ldots, A_N$, thus the optimal chain can be obtained by taking one of maximum $C[j]$. To compute $C[j]$ efficiently, the algorithm maintains one[4] data structure per path in the MPC. The data structure of the i-th path of the MPC stores the information on the C values for all the anchors whose path endpoints belong to the i-th path. Since the MPC, by definition, covers all the vertices of G, to compute a particular $C[j]$ it suffices to query all the data structures that contain anchors whose path endpoint reaches $A_j.u$ [27, Observation 1], which is done through the *forward propagation links*.

More precisely, the vertices v of G are processed in topological order:

Step 1, update structures: For every anchor A_j whose path endpoint is v, and for every i such that v belongs to the i-th path of the MPC, the data structure of the i-th path is updated with the information of $C[j]$.

Step 2, update C values: For every forward link (u, i) from v (that is, u is the last vertex, different from v, that reaches v in the i-th path), and every anchor A_j whose path starting point is u, the entry $C[j]$ is updated with the information stored in the data structure of the i-th path.

Finally, since each data structure is queried and updated $O(kN)$ times, and since these operations can be implemented in $O(\log N)$ time, the running time of the two steps for the entire computation is $O(kN \log N)$ plus $O(k|V|)$ for scanning the forward links.

2.3 One-node suffix-prefix overlaps

When trying to apply the approach from Section 2.2 to solve Problem 2 we have to overcome some problems. First, it no longer holds that every possible chain is a subsequence of $\mathcal{A}$ sorted by (topological order of) anchor path endpoints: an anchor A whose path is a single vertex v can be preceded by another anchor A' whose path ends at v, however A could appear before A' in the order. To solve this issue we first sort $\mathcal{A}$ by path endpoints, and in case of ties by interval endpoints. As such, anchors appearing after cannot precede anchors appearing before in the order, and the optimal chain can be obtained by computing the maximum $C[j]$.

A second problem is that **Step 1** assumes that $C[j]$ contains its final value (and not an intermediate computation). Since suffix-prefix path overlaps were not allowed, this was a valid assumption in the previous case, however, because one-node suffix-prefix overlaps are now allowed, the chains whose $A_j.W$ has a suffix-prefix overlap of one node (with the previous anchor) are not considered. We fix this issue by adding a new **Step 0** before **Step 1** in each iteration. **Step 0** includes (into the computation of $C[j]$) those chains whose last anchors have a one-node suffix-prefix overlap, before applying **Steps 1 and 2**. **Step 0** uses a data structure (only one[5] and it is reinitialized in every **Step 0**) that maintains the information of the C values for all anchors whose paths ends at v (the currently processed node). In this case, the information in the data structure is not propagated through forward links (done in **Step 2** as before), but this information is used to update the C values of anchors whose paths start at v.

More precisely, for every anchor A_j whose path starts or ends at v, in increasing order of $A_j.y$, we do:

Step 0.1, update C value: If the starting point of A_j's path is v, then we update $C[j]$ with the information stored in the data structure.

Step 0.2, update the structure: If the endpoint of A_j's path is v, then we update the data structure with the information of $C[j]$.

Note that in this case the update of the C value comes before the update of the data structure to ensure that single node anchor paths are computed correctly. Finally, since every anchor produces at most two

[4]The algorithm actually maintains two data structures per path in the MPC. One to compute the optimal chain whose last anchor interval overlaps the previous, and one to compute the optimal chain whose last anchor interval does not overlap the previous. For ease of explanation we join these two as one data structure.

[5]Again, we maintain two data structures (as explained before), but we join them by ease of explanation.

Algorithm 1: Our solution to Problem 2. For a vertex u, $\mathcal{A}_u$ and $\mathcal{A}^u$ are the anchors of $\mathcal{A}$ whose path starts and ends at u, respectively, $\mathcal{A}_u^u = \mathcal{A}_u \cup \mathcal{A}^u$. The entry `forward[u]` contains the pairs (v, P_i), such that u is the last vertex, in path P_i, that reaches v. These links can be pre-computed in time $O(k|E|)$, and there are $O(k|V|)$ of them in total [27]. Data structures can answer `update`, and range maximum queries (`rMq`).

Input : DAG $G = (V, E)$, a path cover $\mathcal{P} = P_1, \ldots, P_k$ of G, anchors $\mathcal{A} = A_1, \ldots, A_N$ sorted by path endpoint $(A_i.v)$, and in case of ties by interval endpoint $(A_i.y)$.
Output: A chain $\mathcal{C}$ of maximum coverage.

```
Initialize data structures {D_i^∅, D_i^∩ | i ∈ {1,...,k}};   // of anchors with path endpoints in P_i
Initialize array C;                                          // C[j]: maximum coverage of a chain ending at A_j
for s ∈ V in topological order do
    Initialize data structure U^∅, U^∩;                      // of anchors with path endpoints equal to s
    for A_j ∈ A_s^s in increasing order of y do
        if A_j ∈ A_s then
            C[j] ← max(C[j], A_j.y − A_j.x + 1 + U^∅.rMq(0, A_j.x − 1))
            C[j] ← max(C[j], A_j.y + U^∩.rMq(A_j.x, A_j.y))
        if A_j ∈ A^s then
            U^∅.update(A_j.y, C[j])
            U^∩.update(A_j.y, C[j] − A_j.y)
    for A_j ∈ A^s do
        for P_i ∈ {P ∈ P | s ∈ P} do
            D_i^∅.update(A_j.y, C[j])
            D_i^∩.update(A_j.y, C[j] − A_j.y)
    for (t, P_i) ∈ forward[s] do
        for A_j ∈ A_t do
            C[j] ← max(C[j], A_j.y − A_j.x + 1 + D_i^∅.rMq(0, A_j.x − 1))
            C[j] ← max(C[j], A_j.y + D_i^∩.rMq(A_j.x, A_j.y))
return Maximum coverage chain C recovered from array C
```

operations in the data structure the running time of **Step 0** during the whole algorithm is $O(N \log N)$, thus maintaining the asymptotic running time of $O(k(|V| + |E|) \log |V| + kN \log N)$. Algorithm 1 shows the pseudocode of the algorithm.

Our algorithm for solving Problem 2 can be used to solve co-linear chaining in general graphs in the same running time (see Appendix).

2.4 Co-linear chaining in time $O(kN \log kN)$

As we discussed, the pseudocode of Algorithm 1 takes $O(kN)$ updates and queries to the data structures, which can be answered in $O(\log N)$ time each, thus adding up to $O(kN \log N)$ time in total. However, the for loops iterate over all the vertices of the graph, and over all the forward links, adding an $O(k|V|)$ additional time to the chaining process.

Having such $O(k|V|)$ time while chaining does not change the $O(k(|V| + |E|) \log |V| + kN \log N)$ asymptotic running time of our algorithm, but it affects the practical performance of chaining. As such, in our implementation, we decided to divide the algorithm into indexing (computing the path cover, and other data structures that are independent of the anchors) that we solve in $O(k(|V| + |E|) \log |V|)$ time, and chaining that we solve in time $O(kN \log kN)$[6].

[6]Note that we worsen running time by a k factor inside the log factor. However, in practice $k \ll N$, thus the asymptotic

To remove the chaining's dependency on the graph size we got rid of the for loops of Algorithm 1, and instead process the anchors (plus the required objects) in an order that simulates the for loop order of Algorithm 1. In the case of **Step 1**, this order is simulated by sorting the anchors by topological order of the path endpoints[7], which can be done in $O(N \log N)$ time (by previously computing the topological ordering on indexing time). For **Step 0**, we also require to sort the anchors by the topological order of the path starting points, as well as interleaving the computation of **Step 0.1** and **Step 0.2**. For interleaving the computation of **Step 0.1** and **Step 0.2** (as well as the computation of **Step 0**, **Step 1** and **Step 2**), we partition the corresponding sorted arrays, such that each part corresponds to a different vertex, which can be done in extra $O(N)$ time ($O(kN)$ time in the case of **Step 2**). To interleave the computation of the different steps, we process the sorted arrays by parts according to the topological ordering of the corresponding paths. Finally, to simulate the for loop of **Step 2**, we sort the tuples (s, P_i, A_j) (such that $(A_j.v, P_i) \in$ `forward[s]`) by topological order of s. Since there can be $O(kN)$ such tuples, this takes $O(kN \log kN)$ time.

2.5 Implementation

We implemented our algorithm to solve Problem 2 inside a new aligner of long reads to (acyclic) variation graphs, `GraphChainer`. Our C++ code is built on `GraphAligner`'s codebase. Moreover, `GraphAligner`'s alignment routine is also used as a blackbox inside our aligner as explained next (see Fig. 4 in the Appendix for an overview).

To obtain anchors, `GraphChainer` extracts *fragments* of length $\ell =$ `colinear-split-len` (default 35, experimentally set) from the long read. By default `GraphChainer` splits the input read into non-overlapping substrings of length ℓ each, which together cover the read. Additionally, `GraphChainer` can receive an extra parameter $s =$ `speed`, which samples a fragment every $\lceil s \times \ell \rceil$ positions instead. That is, if $s > 1$, the fragments do not fully cover the input read, and if $s \leq 1 - 1/\ell$, fragments are overlapping. Having less fragments to align increases the speed, but decreases the accuracy. These fragments are then aligned with `GraphAligner` to the variation graph[8], with default parameters; for each alignment reported by `GraphAligner`, we construct an anchor made up of the path in the graph corresponding to the alignment, and the interval in the long read corresponding to the fragment.

Having the input set of anchors, `GraphChainer` then solves Problem 2. The MPC index is computed with the $O(k(|V| + |E|) \log |V|)$ time algorithm [27] for its simplicity, where for its max-flow routine we implemented Dinic's algorithm [9]. Another optimization is to not enumerate over all nodes and forward links, but instead process the anchors in the corresponding order, thus making the running time (after preprocessing) independent of the graph size (see Section 2.4). The (range maximum query) data structures that our algorithm maintains per MPC path are supported by our own thread-safe implementation of a treap [34].

After the co-linear chaining algorithm outputs a maximum-coverage chain $A_{i_1}, \ldots, A_{i_q}$, `GraphChainer` connects the anchor paths to obtain a longer path, which is then reported as the answer. More precisely, for every $j \in \{1, \ldots, q-1\}$, `GraphChainer` connects $A_{i_j}.v$ to $A_{i_{j+1}}.u$ by a shortest path (in the number of nodes). Such connecting path exists by the definition of precedence in a chain, and it can be found by running a breadth-first search (BFS) from $A_i.v$. This strategy is simpler than e.g., connecting by a path of minimum edit distance with the substring of the long read between the corresponding intervals of the anchors. However, the need to connect these anchor paths originates from the lack of other anchor paths in the region to be chained, which likely indicates that this region has a high error rate in the read. Searching for optimal alignments in these erroneous regions can hardly improve the overall alignment quality, which is indeed borne out by our experiments.

Next, since our definition of co-linear chaining maximizes the coverage of the input read, it could happen that, in an erroneous chaining of anchors, the path reported by `GraphChainer` is much longer than the input read. To control this error, `GraphChainer` splits its solution whenever a path joining consecutive anchors

running time is maintained.

[7] If there are ties, they are sorted by interval endpoints. This is done in this case as well as the other sortings, but we remove it from the phrasing by ease of explanation.

[8] All calls to `GraphAligner` are implemented internally, in the same binary of `GraphChainer`.

Table 1: Statistics of each dataset (variation graphs and reads) used in our experiments. For graphs LRC and MHC1 [20], their reads were simulated on a random path of the corresponding graph. Graphs Chr22 and Chr1 were built using GRCh37 as reference and variants from the Thousand Genomes Project phase 3 release [5], reads are PacBio reads obtained from SRA (SRX4480530) and then filtered with `minimap2`.

Graph	#Nodes	Node labels bps	Width	#Reads	Total read bps	Coverage
LRC	117 787	1 099 856	4	1 093	15 872 214	15×
MHC1	479 531	5 138 362	4	5 091	74 524 274	15×
Chr22	3 197 160	52 423 213	7	136 494	2 858 621 416	56×
Chr1	18 807 963	255 754 179	9	907 572	19 617 046 919	79×

is longer (label-wise) than some parameter $g =$ `colinear-gap` (default 10 000, in the order of magnitude of the long read length), and reports the longest path after these splits.

Finally, `GraphChainer` runs `GraphAligner` again, but now for the entire long read. It then uses `edlib` [39] to compute the best alignment between the (best) alignment reported by `GraphAligner` and the one computed by co-linear chaining, which is then reported. Therefore, `GraphChainer` can be also be viewed as a refinement of `GraphAligner`'s alignment results.

3 Experiments

We run several long reads to variation graph alignment experiments, and compare `GraphChainer` against `GraphAligner`, the state-of-the-art aligner of long reads to (chromosome level) variation graphs. We excluded `SPAligner` since this is tailored for alignments to assembly graphs, making it incompatible with our experiments. We also excluded the recent extension of `AStarix` for long reads [19], since the program did not finish after 3 hours of execution on our smallest dataset (compared to ∼1 second for `GraphAligner` and `GraphChainer`, see Table 3).

3.1 Datasets

Small graphs and simulated reads. As a first test of our solution, we use two (relatively) small variation graphs and simulated long reads. The graphs, LRC and MHC1 [20] correspond to two of the most diverse variation regions in the human genome [8, 37]. For each of these graphs, we sample a reference sequence and use it to simulate a PacBio read dataset of 15x coverage with the package `Badread` [42]. To build the reference sequence we first sample a source-to-sink path from the graph by starting at a source, and repeatedly choosing an out-neighbor of the current node uniformly at random, until reaching a sink; finally, we concatenate the node labels on the path. The ambiguous characters on the simulated reference sequence are randomly replaced by one of its indicated characters. For each simulated read, we know its original location on the sampled reference sequence, which can thus also be mapped to its ground truth path on the graph.

Chromosome-level graphs and real reads. We built the variation graphs of human chromosomes 22 and 1 using the `vg toolkit`. For the graphs, we used GRCh37 as reference, and variants from the Thousand Genomes Project phase 3 release [5]. For the reads, we used the whole human genome PacBio Sequel data from HG00733 (SRA accession SRX4480530)[9]. We first aligned all the reads against GRCh37 with `minimap2` and selected only the reads that are aligned to chromosome 22 and 1, respectively, with at least 70% of their length, and have no longer alignments to other chromosomes. This filtering leads to 56× and 79× coverage on Chr22 and Chr1, respectively.

Table 1 shows more statistics of the variation graphs and read sets. Note that our biggest variation graph (Chr1) is (approximately) 5 times bigger than the biggest variation graph used in `GraphAligner`'s evaluation

[9]This is the same read set used by `GraphAligner`'s experiments [31, p.7] but without the subsampling to 15× coverage.

(Chr22) [31]. Also, note that Chr1 is one of the most complex variation graphs of the human chromosomes (see Table 4 in the Appendix). We did not test the aligners on the variation graph of the whole human genome due to the huge time and memory requirements of both aligners for this task. However, testing the aligners on one individual chromosome captures most of the complexity of the problem. The variation graph (built as above) of the whole human genome consists of the disjoint union of the variation graphs of all chromosomes (i.e., with no edges connecting these independent chromosome components). As such, one can reduce the alignment task to each of the weakly connected components of the graph (chromosome graphs) and output the "best" alignment among them.

Finally, note that variation graphs (built as above) span all genomic positions, i.e., do not collapse repeats like, e.g., de Bruijn graphs. As such, the aligners do not need extra steps to identify the corresponding repeat of an alignment: different instances of a repeat appear in different nodes (genomic positions) in the variation graph.

3.2 Evaluation metrics and experimental setup

For each read, the aligners output one or several paths of the variation graph to which the read was aligned. We classify a read as having a "good alignment" if the edit distance (computed using `edlib`) between the read and at least one path to which it is aligned is strictly smaller than a given threshold $0 < \sigma \leq 1$ multiplied by the read length. For example, given a read of length 10 000, an alignment with edit distance 2 000 is considered good for any $\sigma > 0.2$ and not good for any $\sigma \leq 0.2$. Thus, σ can be interpreted as edit distance upper bound *normalized* by the read length.

This criterion is an approximate generalization of the evaluation metric used earlier [31, 26], which classifies an alignment as "correct" if the overlap between the longest alignment of the read and the genomic position from which it was simulated is at least 10% (which roughly corresponds to the setting $\sigma = 0.9$). The main benefits of our definition of "good alignment" are that it applies also to alignments of real reads where no ground truth path is available, and it provides a more complete picture of the accuracy of the alignments by varying the value of σ. In our opinion it is too permissive to classify a "good alignment" for an overlap of at least 10% with its ground truth path (i.e., $\sigma = 0.9$); instead, more informative values of σ should be those moderately above the error rate of the reads.

Moreover, on simulated data where the ground truth path from where the read was simulated is available, we also compute the edit distance between the best alignment and the label of the ground truth path, and analogously classify a read as having a "good alignment" based on the value of σ. We also compute the *normalized* overlap (in genomic positions) between the best alignment and the ground truth. In this case, we classify a read as having a "good alignment" if its normalized overlap is greater than another given threshold $0 < \delta \leq 1$.

All experiments were conducted on a server with AMD Ryzen Threadripper PRO 3975WX CPU with 32 cores and 504GB of RAM. All aligners were run using 30 threads. Time and peak memory usage of each program were measured with the GNU `time` command. The commands used to run each tool are listed in the Appendix.

3.3 Results and discussion

Small graphs and simulated reads (LRC and MHC1). For LRC and $\sigma = 0.3$ or more (as well as $\delta = 0.3$ or more), `GraphChainer` has at least 3–4% more good alignments, when classifying a good alignment w.r.t. the read, w.r.t. the ground truth path, and w.r.t. the overlap (in genomic positions) with the ground truth, see Fig. 2. Since the long reads have varying sizes, we also compute the total length of the reads having a good alignment ("good length"), and express this relative to the total read length, see Table 2. We observe that `GraphChainer` reports 4–5% more total good length. Similar numbers (by about 1% smaller) hold also for MHC1. Moreover, in both graphs and $\delta = 0.1$ (criterion used in `GraphAligner`'s publication [31, p. 3]), `GraphAligner` has less than 96% of good alignments, whereas `GraphChainer` has over 99%.

Table 2: Percentage of reads having a good alignment with respect to the read ("Good alignments"), and the percentage of all the read base pairs inside reads with a good alignment ("Good length"), for $\sigma \in \{0.3, 0.4\}$ (i.e., the edit distance between the read and its best alignment can be up to 30% or 40% of the read length, respectively). Percentages in parentheses are relative improvements w.r.t. `GraphAligner`.

Graph	Aligner	Good alignments		Good length	
		$\sigma = 0.3$	$\sigma = 0.4$	$\sigma = 0.3$	$\sigma = 0.4$
LRC	`GraphAligner`	91.40%	93.14%	90.60%	92.60%
	`GraphChainer --speed 1`	94.42% (+3.30%)	97.35% (+4.52%)	93.52% (+3.21%)	97.04% (+4.80%)
MHC1	`GraphAligner`	92.69%	94.11%	92.05%	93.52%
	`GraphChainer --speed 1`	94.72% (+2.18%)	97.21% (+3.30%)	93.92% (+2.03%)	96.52% (+3.21%)
Chr22	`GraphAligner`	82.26%	88.32%	74.78%	82.95%
	`GraphChainer --speed 1`	96.30% (+17.06%)	98.85% (+11.92%)	96.25% (+28.71%)	98.92% (+19.24%)
	`GraphChainer --speed 2`	95.12% (+15.63%)	98.30% (+11.29%)	94.96% (+26.98%)	98.41% (+18.63%)
	`GraphChainer --speed 3`	94.15% (+14.46%)	97.87% (+10.81%)	93.77% (+25.39%)	97.95% (+18.07%)
Chr1	`GraphAligner`	83.96%	89.55%	77.55%	85.04%
	`GraphChainer --speed 1`	94.68% (+12.77%)	97.98% (+9.42%)	94.47% (+21.82%)	98.16% (+15.43%)
	`GraphChainer --speed 2`	92.79% (+10.53%)	97.02% (+8.35%)	91.92% (+18.53%)	97.05% (+14.12%)
	`GraphChainer --speed 3`	91.42% (+8.89%)	96.22% (+7.46%)	89.79% (+15.79%)	96.00% (+12.89%)

Table 3: Running time (in format hh:mm:ss) and peak memory (Mem, in GBs) when aligning simulated and real PacBio reads. Column "Real time" corresponds to the wall-clock time of running the aligners with 30 threads.

Graph	Aligner	Mem	CPU time	Real time
LRC	`GraphAligner`	1.24	00:00:33	00:00:01
LRC	`GraphChainer --speed 1`	1.36	00:00:29	00:00:01
MHC1	`GraphAligner`	2.53	00:02:38	00:00:07
MHC1	`GraphChainer --speed 1`	3.02	00:02:22	00:00:08
Chr22	`GraphAligner`	8.30	01:58:11	00:04:12
Chr22	`GraphChainer --speed 1`	13.19	04:44:58	00:09:56
Chr22	`GraphChainer --speed 2`	13.57	03:45:56	00:07:57
Chr22	`GraphChainer --speed 3`	13.77	03:27:30	00:07:20
Chr1	`GraphAligner`	19.57	13:37:22	00:28:42
Chr1	`GraphChainer --speed 1`	61.31	72:30:45	02:28:12
Chr1	`GraphChainer --speed 2`	61.44	45:47:19	01:34:21
Chr1	`GraphChainer --speed 3`	61.64	37:11:24	01:17:05

Chromosome-level graphs and real reads (Chr22 and Chr1). For Chr22, the difference between `GraphAligner` and `GraphChainer` is even clearer, see Fig. 3. Since these graphs are larger, `GraphAligner`'s seeds are more likely to have false occurrences in the graph, and thus extending each seed (cluster) *individually* leads to worse alignments in more cases. As shown in Table 2, for $\sigma = 0.4$, `GraphChainer`'s improvements in reads with a good alignment, and in total length of reads with a good alignment, are up to 11.92%, and 19.24%, respectively. For the more stringent condition $\sigma = 0.3$, these improvements are even higher, up to 17.06% and 28.71%, respectively.

For Chr1, the improvement in the two metrics is slightly smaller, but still significant. For $\sigma = 0.4$, the improvements are up to 9.42% in terms of good alignments and 15.43% in terms of good length, and up to 12.77% and 21.82%, for $\sigma = 0.3$. For both chromosomes and $\sigma = 0.4$, the total length of reads with a good `GraphChainer` alignment is over 98% of the total read length.

Performance. Table 3 shows the running time and peak memory of both aligners. Even though `GraphChainer` takes more time and memory resources, these are still within the capabilities of a modern high-performance computer. Moreover, the aligners are naively parallelized at read level, thus they scale with a larger number of cores. Both running times are larger on Chr1 due to its bigger size, but also to its larger read coverage (recall Table 1). To explore other tradeoffs between speed and accuracy, we ran `GraphChainer` with `--speed` in $\{1, 2, 3\}$. We observe that the alignment accuracy remains significantly above `GraphAligner`'s (especially in terms of good length), while the running time shrinks by up to half. Table 6 in the Appendix shows the

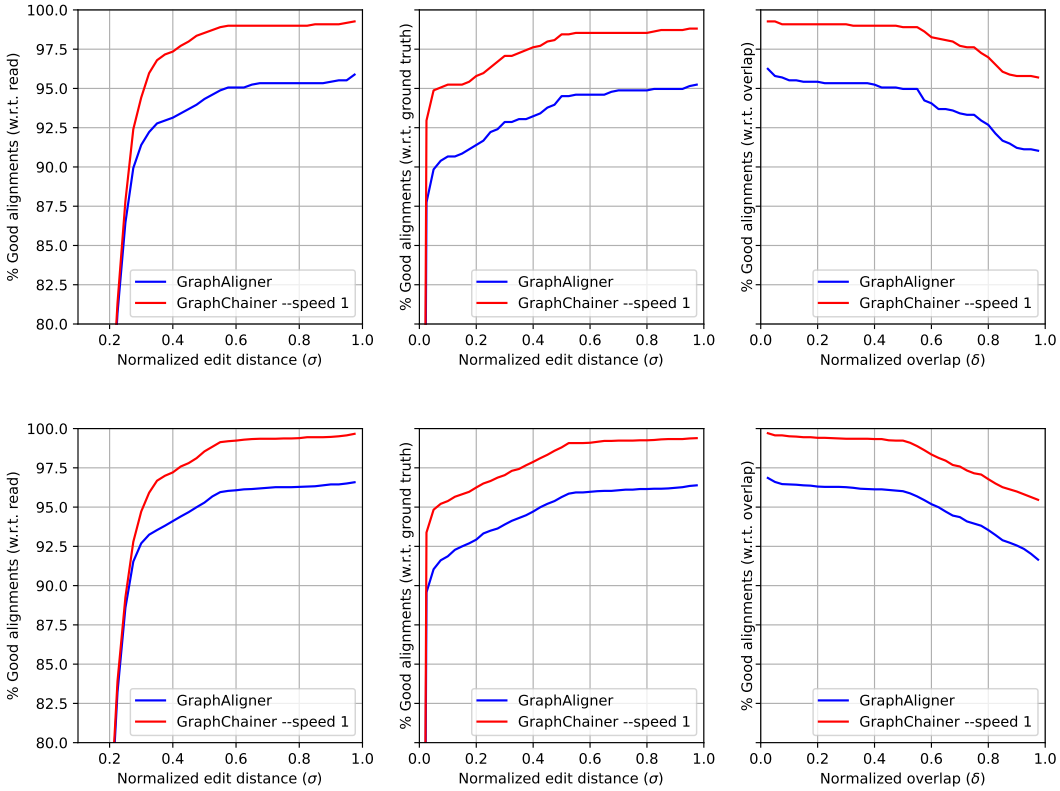

Figure 2: Plots show the percentages of good alignments w.r.t reads (left), ground truth (center), and overlap with ground truth (right), on LRC (top row) and MHC1 (bottom row).

running times of each phase of the `GraphChainer`'s alignment: further improvements are possible by improving the method for obtaining anchors, e.g. by using a fast short-read graph aligner or recent developments in MEMs [33].

4 Conclusions

The pangenomic era has given rise to several methods, including the `vg toolkit` [14], for accurately and efficiently aligning short reads to variation graphs. However, these tools fail to scale when considering long reads and much less work has been published around this problem. `GraphAligner` [31] is the state-of-the-art for aligning long reads to (chromosome level) variation graphs. Here, we have presented the first efficient implementation of co-linear chaining on a DAG and applied it to aligning long reads to variation graphs. We showed that our new method, `GraphChainer`, significantly improves the alignments of `GraphAligner`, on real PacBio reads and chromosome-level variation graphs.

Thus, we showed that co-linear chaining, successfully used by sequence-to-sequence aligners [26], is a useful technique also in the context of variation graphs. Moreover, the fact that human variation graphs have small width (Table 4), combined with our co-linear chaining algorithm running in time $O(kN \log kN)$ (once the graph is preprocessed), are key in making co-linear chaining efficient on DAGs in practice. We hope that these facts could spur the adoption of this technique by future variation graph aligners, as well as in other applications.

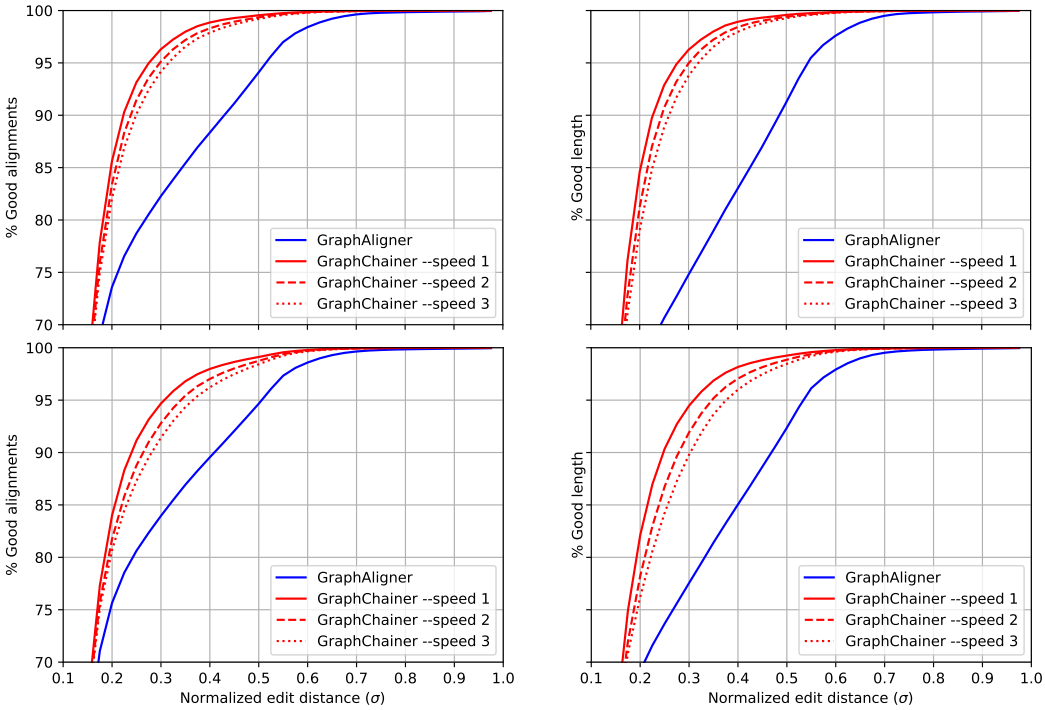

Figure 3: Percentage of reads with a good alignment w.r.t. the read (left), and the percentage of total read length in reads with a good alignment w.r.t. the read (right), on Chr22 (top) and Chr1 (bottom).

References

[1] Mohamed Abouelhoda. A chaining algorithm for mapping cDNA sequences to multiple genomic sequences. In *International Symposium on String Processing and Information Retrieval*, pages 1–13. Springer, 2007.

[2] Amihood Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. *J. Algorithms*, 35(1):82–99, 2000.

[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. *arXiv preprint arXiv:2107.05717*, 2021. To appear in the Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms (SODA 2022).

[5] Laura Clarke, Susan Fairley, Xiangqun Zheng-Bradley, Ian Streeter, Emily Perry, Ernesto Lowy, Anne-Marie Tassé, and Paul Flicek. The international Genome sample resource (IGSR): A worldwide collection of genome variation incorporating the 1000 Genomes Project data. *Nucleic Acids Research*, 45(D1):D854–D859, 09 2016. `doi:10.1093/nar/gkw829`.

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

[7] Edsger Wybe Dijkstra. *A discipline of programming*, volume 613924118. prentice-hall Englewood Cliffs, 1976.

[8] Alexander Dilthey, Charles Cox, Zamin Iqbal, Matthew R Nelson, and Gil McVean. Improved genome inference in the MHC using a population reference graph. *Nature genetics*, 47(6):682–688, 2015.

[9] Efim A Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[10] Tatiana Dvorkina, Dmitry Antipov, Anton Korobeynikov, and Sergey Nurk. SPAligner: alignment of long diverged molecular sequences to assembly graphs. *BMC Bioinformatics*, 21(12):306, 2020. `doi:10.1186/s12859-020-03590-7`.

[11] Jordan M Eizenga, Adam M Novak, Jonas A Sibbesen, Simon Heumos, Ali Ghaffaari, Glenn Hickey, Xian Chang, Josiah D Seaman, Robin Rounthwaite, Jana Ebler, et al. Pangenome graphs. *Annual review of genomics and human genetics*, 21:139–162, 2020.

[12] Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 55:1–55:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ICALP.2019.55`.

[13] Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. In Tomás Bures, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdzinski, Claus Pahl, Florian Sikora, and Prudence W. H. Wong, editors, *SOFSEM 2021: Theory and Practice of Computer Science - 47th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2021, Bolzano-Bozen, Italy, January 25-29, 2021, Proceedings*, volume 12607 of *Lecture Notes in Computer Science*, pages 608–622. Springer, 2021. `doi:10.1007/978-3-030-67731-2\_44`.

[14] 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, 2018.

[15] Daniel Gibney, Gary Hoppenworth, and Sharma V. Thankachan. Simple reductions from Formula-SAT to pattern matching on labeled graphs and subtree isomorphism. In Hung Viet Le and Valerie King, editors, *4th Symposium on Simplicity in Algorithms, SOSA 2021, Virtual Conference, January 11-12, 2021*, pages 232–242. SIAM, 2021. `doi:10.1137/1.9781611976496.26`.

[16] Glenn Hickey, David Heller, Jean Monlong, Jonas A Sibbesen, Jouni Sirén, Jordan Eizenga, Eric T Dawson, Erik Garrison, Adam M Novak, and Benedict Paten. Genotyping structural variants in pangenome graphs using the vg toolkit. *Genome biology*, 21(1):1–17, 2020.

[17] Bhavna Hurgobin and David Edwards. SNP discovery using a pangenome: has the single reference approach become obsolete? *Biology*, 6(1):21, 2017.

[18] Pesho Ivanov, Benjamin Bichsel, Harun Mustafa, André Kahles, Gunnar Rätsch, and Martin Vechev. AStarix: Fast and optimal sequence-to-graph alignment. In *International Conference on Research in Computational Molecular Biology*, pages 104–119. Springer, 2020.

[19] Pesho Ivanov, Benjamin Bichsel, and Martin Vechev. Fast and optimal sequence-to-graph alignment guided by seeds. *bioRxiv*, 2021. URL: `https://www.biorxiv.org/content/early/2021/11/08/2021.11.05.467453`, `arXiv:https://www.biorxiv.org/content/early/2021/11/08/2021.11.05.467453.full.pdf`, `doi:10.1101/2021.11.05.467453`.

[20] Chirag Jain, Sanchit Misra, Haowen Zhang, Alexander Dilthey, and Srinivas Aluru. Accelerating sequence alignment to graphs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–461, 2019. `doi:10.1109/IPDPS.2019.00055`.

[21] 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, 2020.

[22] Anna Kuosmanen, Topi Paavilainen, Travis Gagie, Rayan Chikhi, Alexandru Tomescu, and Veli Mäkinen. Using minimum path cover to boost dynamic programming on DAGs: Co-linear chaining extended. In Benjamin J. Raphael, editor, *Research in Computational Molecular Biology*, pages 105–121, Cham, 2018. Springer International Publishing.

[23] Christopher Lee, Catherine Grasso, and Mark F Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.

[24] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*, 2013.

[25] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016.

[26] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 05 2018. `doi:10.1093/bioinformatics/bty191`.

[27] 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.

[28] Karen H Miga and Ting Wang. The need for a human pangenome reference sequence. *Annual Review of Genomics and Human Genetics*, 22, 2021.

[29] Gene Myers and Webb Miller. Chaining Multiple-Alignment Fragments in Sub-Quadratic Time. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pages 38–47, USA, 1995. Society for Industrial and Applied Mathematics.

[30] Mikko Rautiainen, Veli Mäkinen, and Tobias Marschall. Bit-parallel sequence-to-graph alignment. *Bioinformatics*, 35(19):3599–3607, 03 2019. `doi:10.1093/bioinformatics/btz162`.

[31] Mikko Rautiainen and Tobias Marschall. GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome biology*, 21(1):1–28, 2020.

[32] 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.

[33] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: a pangenomics index for finding MEMs. *bioRxiv*, 2021.

[34] Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

[35] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[36] Tetsuo Shibuya and Igor Kurochkin. Match chaining algorithms for cDNA mapping. In *International Workshop on Algorithms in Bioinformatics*, pages 462–475. Springer, 2003.

[37] Jonas Andreas Sibbesen, Lasse Maretty, and Anders Krogh. Accurate genotyping across variant classes and lengths using variant graphs. *Nature genetics*, 50(7):1054–1059, 2018.

[38] Jouni Sirén, Jean Monlong, Xian Chang, Adam M Novak, Jordan M Eizenga, Charles Markello, Jonas A Sibbesen, Glenn Hickey, Pi-Chuan Chang, Andrew Carroll, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, 2021.

[39] Martin Šošić and Mile Šikić. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 01 2017. doi:10.1093/bioinformatics/btw753.

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

[41] Daniel Valenzuela, Tuukka Norri, Niko Välimäki, Esa Pitkänen, and Veli Mäkinen. Towards pan-genome read alignment to improve variation calling. *BMC genomics*, 19(2):123–130, 2018.

[42] Ryan R. Wick. Badread: simulation of error-prone long reads. *Journal of Open Source Software*, 4(36):1316, 2019. doi:10.21105/joss.01316.

A Statistics on variation graphs of human genome

Table 4: Statistics of variation graphs of human chromosomes built with the `vg toolkit` using GRCh37 as reference, and variants from the Thousand Genomes Project phase 3 release [5]. The number of nodes is from the compact representation where a non-branching path is merged into a single node. Each graph is a DAG, and the numbers refer only to one of the two weakly connected components (one being the reverse complement of the other).

Chr	#Nodes	Node labels bps	Width
1	18 807 963	255 754 179	9
2	20 597 735	250 312 064	6
3	16 965 471	203 883 122	7
4	16 662 965	196 912 161	6
5	15 313 396	186 204 491	6
6	14 596 952	176 169 819	9
7	13 707 868	163 880 288	8
8	13 370 501	150 986 669	8
9	10 355 761	144 794 206	7
10	11 595 921	139 553 125	6
11	11 760 609	139 076 341	7
12	11 239 568	137 745 335	6
13	8 304 603	118 040 865	6
14	7 714 611	110 019 784	7
15	7 045 787	104 968 212	6
16	7 837 615	93 074 017	8
17	6 758 004	83 545 050	6
18	6 592 253	80 357 608	7
19	5 306 144	60 981 512	6
20	5 268 137	64 854 544	6
21	3 207 166	49 243 683	6
22	3 197 160	52 423 213	7
X	10 011 934	158 748 581	9
Y	184 003	59 435 025	2

B CLC in general graphs

Our algorithm for Problem 2 allows us to efficiently solve a formulation of co-linear chaining in general graphs, namely:

Problem 3 (Reachability path precedence CLC)**.** *Same as Problem 1, such that for every $A, A' \in \mathcal{A}$, we define A* precedes *A' iff $A.y < A'.y$ and $A.v$ reaches $A'.u$.*

Note that in this version we do not restrict the graph to be acyclic. We solve Problem 3 by solving Problem 2 in the condensation of G. The *condensation* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of a graph $G = (V, E)$ is the DAG of its strongly connected connected components, which can be computed in $O(|V| + |E|)$ time [40, 7, 35]. More precisely, $\mathcal{V} = \{S \mid S \text{ is a strongly connected component of } G\}$, and $\mathcal{E} = \{(S, S') \mid \exists u \in S, v \in S', (u, v) \in E\}$. Since the strongly connected components of a graph form a partition of the vertices, every vertex of G can be mapped to its corresponding vertex (component) in $\mathcal{G}$. As such, a walk in G can be mapped to its corresponding path in $\mathcal{G}$ (the ordered sequence of components it visits in the walk). Therefore, any path cover of G can be mapped into a path cover of $\mathcal{G}$, thus $\mathsf{width}(G) = k \geq k' = \mathsf{width}(\mathcal{G})$. To finish our reduction we map the input anchors $\mathcal{A}$ to $\mathcal{A}'$ by mapping the corresponding anchor paths to paths in $\mathcal{G}$. Let us call

this anchor transformation f, such that, $f_{\mathcal{G}}(A) = (A.I, P)$, where P is the corresponding path of $A.W$ in $\mathcal{G}$, and thus $\mathcal{A}' = f_{\mathcal{G}}(\mathcal{A})$. The following lemma shows that solving Problem 3 for G and $\mathcal{A}$ is equivalent to solve Problem 2 for $\mathcal{G}$ and $\mathcal{A}'$.

Lemma 1. *$A_{i_1}, \ldots, A_{i_q}$ is a chain of $(G, \mathcal{A})$ iff $f_{\mathcal{G}}(A_{i_1}), \ldots, f_{\mathcal{G}}(A_{i_q})$ is a chain of $(\mathcal{G}, f_{\mathcal{G}}(\mathcal{A}))$.*

Proof. Since the anchor intervals do not change it suffices to check the precedence of anchor paths in both directions of the equivalence. ($\Rightarrow$): For all $j \in \{1, \ldots, q-1\}$, if $v = A_{i_j}.v$ reaches $u = A_{i_{j+1}}.u$, they are either in the same strongly connected component, or the component of v reaches the component of u, thus $f_{\mathcal{G}}(A_{i_j}).v$ reaches $f_{\mathcal{G}}(A_{i_{j+1}}).v$. ($\Leftarrow$): For all $j \in \{1, \ldots, q-1\}$, if $S = f_{\mathcal{G}}(A_{i_j}).v$ reaches $T = f_{\mathcal{G}}(A_{i_{j+1}}).u$, then every vertex in S reaches every vertex of T, in particular, $A_{i_j}.v$ reaches $A_{i_{j+1}}.u$. $\square$

Finally, since the anchor intervals do not change, optimal chains are shared in both problems. Therefore, since $\mathcal{G}$ is a DAG, we can solve Problem 3 with our algorithm for Problem 2, in time $O(|V| + |E| + N + k'(|\mathcal{V}| + |\mathcal{E}|) \log |\mathcal{V}| + k'N \log N) = O(k(|V| + |E|) \log |V| + kN \log N)$.

C Commands for running the tools

Read simulation with BadReads:

```
badread simulate --seed {seed} --reference {Ref}
--quantity 15x --length 15000,10000
--error_model pacbio2016 --identity 85,95,5
```

Variation graph construction with vg toolkit (same parameters as those used for `GraphAligner`'s experiments [31, p.24]):

```
vg construct -t 30 -a -r {ref} -v {vcf} -R 22 -p
-m 3000000
```

Read alignment (same parameters for `GraphAligner` as used on its variation graph experiments [31, p.24] but using 30 threads, same parameters for the extension of `AStarix` on its experiments for long reads [19, p.13]):

```
GraphAligner -t 30 -x vg -f {Reads} -g {Graph}
-a {long_gam}

GraphChainer -t 30 -f {Reads} -g {Graph}
-a {clc_gam}

astarix align-optimal -a astar-seeds -g {Graph}
-q {Reads} --fixed_trie_depth 1 --seeds_len 150
-D 14 -G 1 -S 1
```

D Further experimental results

Table 5 shows the effect of allowing one-node overlapping in co-linear chaining. It can be seen that over 90% of the optimal chains contain one-node overlapped anchors. For the real PacBio data on chromosome 22 this ratio is 97%.

Table 6 shows the running times of each phase of the `GraphChainer`'s alignment: further improvements are clearly possible by improving the method for obtaining anchors, e.g. by using a fast short-read graph aligner or by using recent developments in MEMs [33].

Table 5: Impact of allowing one-node suffix-prefix overlaps. The column "Overlapping optimal chains" shows the number of reads whose maximum-coverage chain has at least one pair of anchors with a one-node suffix-prefix overlap.

Graph	`--speed`	#Reads	Overlapping optimal chains
LRC	1	1 093	1 007 (92.1%)
MHC1	1	5 091	4 731 (92.9%)
Chr22	1	136 494	133 290 (97.65%)
Chr22	2		116 065 (85.03%)
Chr22	3		14 468 (10.60%)
Chr1	1	907 572	860 934 (94.86%)
Chr1	2		740 645 (81.61%)
Chr1	3		54 257 (5.98%)

Table 6: CPU time spent in the main phases of `GraphChainer` on Chr22 and Chr1 (in format hh:mm:ss). The column `edlib` is the time needed to run `edlib` between the read and the path found by co-linear chaining to finally report the best solution.

Graph	`--speed`	Getting anchors	Co-linear chaining	`edlib`
Chr22	1	01:24:58	00:30:20	00:47:39
Chr22	2	00:42:12	00:12:14	00:49:11
Chr22	3	00:27:55	00:07:05	00:50:03
Chr1	1	31:49:44	19:36:05	05:49:54
Chr1	2	15:56:58	08:22:24	06:02:22
Chr1	3	10:37:14	04:58:55	06:07:00

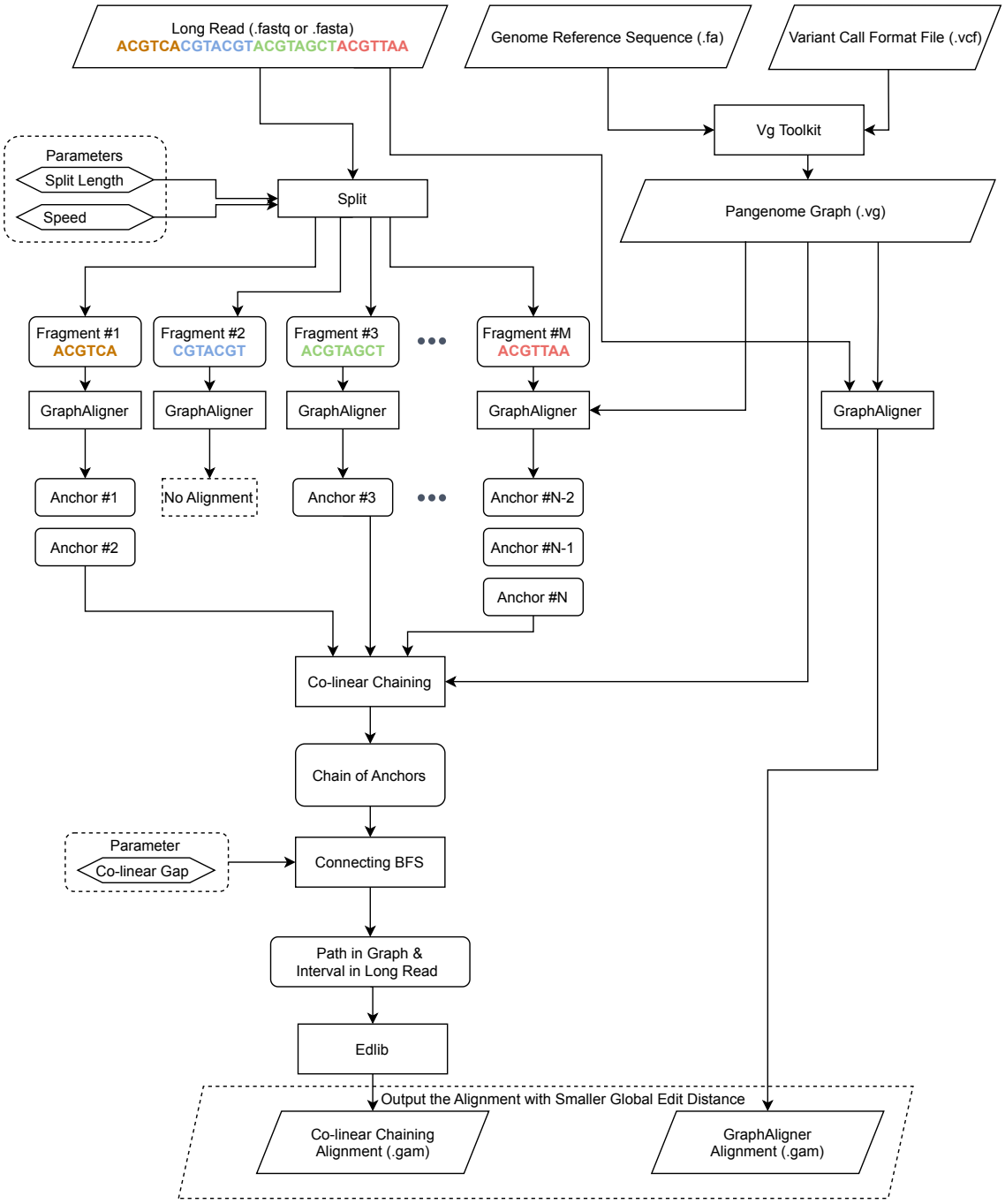

Figure 4: The flow diagram of `GraphChainer`: a long-read is split into fragments (of default length 35), which are aligned with `GraphAligner` (though any other method to extract anchors can be used). Anchors are created for each alignment of a fragment. The anchors are chained with our algorithm from Section 2.3. The optimal chain is split whenever the BFS shortest path between consecutive anchor paths is longer than a co-linear gap limit parameter (default 10 000), and the longest resulting path is kept. If the edit distance between the long read and this path is better than the one obtained by aligning the entire long read with `GraphAligner`, then this alignment is output; otherwise `GraphAligner`'s alignment is reported.