Chaining for Accurate Alignment of Erroneous Long Reads to Acyclic 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 erroneous 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 a string labeled acyclic graph, together with the first efficient implementation of such a co-linear chaining algorithm into a new aligner of long reads to acyclic variation graphs, `GraphChainer`. Compared to `GraphAligner`, `GraphChainer` aligns 12% to 17% more reads, and 21% to 28% more total read length, on real PacBio reads from human chromosomes 1 and 22. On both simulated and real data, `GraphChainer` aligns between 95% and 99% of all reads, and of total read length.

`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, 32]. 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] can build a variation graph from a reference genome and a set of variants, or from a set of reference genomes, while also supporting alignment of reads to the graph.

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, 45, 17, 18, 42]. 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 decreases the reference bias). The alignment method of `vg` was developed for short reads, and while it can run also on long reads, it decreases its performance significantly. To the best of our knowledge, the only sequence-to-graph aligners tailored to long reads are `SPAligner` [10], designed for assembly graphs, `GraphAligner` [35], designed for both assembly and variation graphs, `PaSGAL` [22], designed

*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.

for both short and long reads on acyclic graphs, and the recent extension of `AStarix` for long reads [20], designed for low-error reads and relatively small graphs. `GraphAligner` is the state-of-the-art aligner for more-erroneous 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 [23] and various practical optimizations [20, 19, 34, 22] 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 [26], as in the case of `vg`, seed-and-extend based on minimizers [36], as in the case of `GraphAligner`, parallel computation techniques, as in the case of `PaSGAL`, and heuristic graph search [16] as in the case of `AStarix`. In the case of `GraphAligner`, the aligner finds the minimizers of the read that have occurrences in the graph, clusters and ranks these occurrences, similarly to `minimap` [28], 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 [34]. This strategy was shown to be effective in mapping erroneous 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 through the *co-linear chaining* problem [33], 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, 40, 30], where N is the number of anchors. Co-linear chaining is successfully used by the popular read-to-reference sequence aligner `minimap2` [29] and also in `uLTRA` [30], an aligner of RNA-Seq reads. Moreover, recent results show connections between co-linear chaining and classical distance metrics [30, 21].

The co-linear chaining problem can be naturally extended to a sequence and a labeled graph and has been previously studied for DAGs [31, 25], 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 [35, 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 [27] 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 **character**-labeled 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 [31] 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 [25], 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 5 in the Appendix), further motivating the idea of parameterizing the running time of the co-linear chaining problem on the width of the variation graph.

We present the first algorithm solving co-linear chaining on **string**-labeled DAGs, when allowing one-node suffix-prefix overlaps between anchor paths. The reasons to consider such variation of the problem arise from practice. First, variation graphs used in applications usually compress unary paths, also known as unitigs [24], by storing the concatenation of their labels in a unique node representing them all. This not only allows to compress graph zones without variations, but also reduces the graph size, and therefore the running time of the algorithms run on them. In Tables 1 and 5 we show that typical variation graphs have 10 times more nodes in their character based representation (columns #Nodes and Labels bps). Second, as discussed before, allowing suffix-prefix overlaps of arbitrary length significantly increases the running time of the algorithm, and also requires the use of advanced data structures incurring in an additional penalty in practice. However, the important special non-overlapping case in the character based representation translates into overlaps of at most one node in the corresponding string based representation (see Figure 1), which turns out to be easier to solve. Additionally, 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).

Our solution builds on the previous $O(k(|V| + |E|) \log |V| + kN \log N)$ time solution [31], 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, 31], and $O(kN \log kN)$ for solving co-linear chaining (Appendix). 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.

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 erroneous long reads than the state-of-art aligner `GraphAligner`. On simulated reads, we classify a read as *correctly aligned* if the reported path overlaps $100 \cdot \delta\%$ of the simulated region. We show that `GraphChainer` aligns approximately 3% more reads than `GraphAligner` in all graphs tested and for every criterion $0 < \delta < 1$. Moreover, if the criterion matches that of the average identity between the read and their ground truth, this difference increases to approximately 6% on average. On real reads, where the ground truth is not available, we classify a read as *correctly aligned* if the edit distance between the read and the reported sequence (without edits applied) is at most $100 \cdot \sigma\%$ of the read length. For criterion $\sigma = 0.3$, `GraphChainer` aligns between 12% to 17% more real reads, of 21% to 28% more total length, on human chromosomes 1 and 22. At the same criterion, `GraphChainer` produces a good alignment of 95% of all reads or of total read length, while `GraphAligner` worsens to less that 85% and 78%, respectively.

While this increase in accuracy comes with an increase in computational resources 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

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

the future other methods for finding anchors could be explored, such as using a fast short-read graph aligner or recent developments in MEMs [37].

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*) is a sequence of anchors $A_{i_1}, \ldots, A_{i_q}$ of $\mathcal{A}$, such that for all $j \in \{1, \ldots, q-1\}$, A_{i_j} *precedes* $A_{i_{j+1}}$, where the precedence relation 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 final objective is to align a read to the variation graph, co-linear chaining can be defined (and solved) independent of the labels in these objects. However, in the case of string labeled graphs we will also need the exact position $A.p$ in the label of $A.v$, where the string represented by the anchor path finishes (see Figure 1).

Co-linear chaining has been solved [31] 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 on string labeled DAGs when path precedence allows a one-node suffix-prefix overlap. More precisely, we solve the following problem (see Figure 1):

Problem 2 (One-node suffix-prefix overlap CLC)**.** *Same as Problem 1, but restricting G to be a string labeled 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 $A'.u$, but if $A.v = A'.v$ we also require that $A.p < A.p'$.*

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 [31] 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:

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

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 $C[j]$

[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.

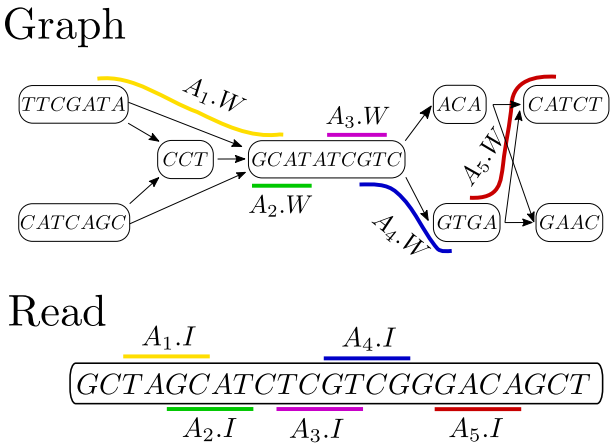

Figure 1: An illustrative example of Problem 2 consisting of a string labeled DAG, a read and a set of five anchors (paths in the graphs, paired with intervals in the read), shown here in different colors. The sequence $\mathcal{C} = A_1, A_2, A_3, A_4, A_5$ is a chain with $cov(\mathcal{C}) = 16$, and it is optimal since every other chain is subsequence of $\mathcal{C}$. In particular, note that A_2 precedes A_3 because $A_2.y = 8 < A_3.y = 13$ and $A_2.v$ reaches $A_3.u = A_3.v$, but $A_2.p = 4 < A_3.p = 9$. If one-node suffix-prefix overlaps are not allowed, $\mathcal{C}$ is not a chain and the optimal chain would have coverage 4.

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 $A_j.u$ [31, 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 (for example A_1, A_2, A_3 in Figure 1 could have appear in another order). To solve this issue we first sort $\mathcal{A}$ by path endpoints $(A.v)$, and in case of ties by endpoint in the string label of the path endpoint $(A.p)$. As such, anchors appearing after cannot precede anchors appearing before in the order, and the optimal chain can be obtained by computing the maximum $C[j]$.

[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.

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 [31]. Data structures can answer `update`, and range maximum queries (`rMq`).

Input : A path cover $\mathcal{P} = P_1, \ldots, P_k$ of G, anchors $\mathcal{A} = A_1, \ldots, A_N$ sorted by $A_i.v, A_i.p$.
Output: A chain $\mathcal{C}$ of maximum coverage.
`initialize`($\mathcal{D}_i^{\nearrow}, \mathcal{D}_i^{\cap}$), for every $i \in \{1, \ldots, k\}$
for $s \in V$ *in topological order* **do**
  `initialize`($\mathcal{U}^{\nearrow}$,$\mathcal{U}^{\cap}$)
  for $p \in \{A_j.p \mid A_j \in \mathcal{A}^s_s\}$ *in increasing order* **do**
    $\mathcal{A}_p \leftarrow \{A_j \mid A_j.p = p\}$
    for $A_j \in \mathcal{A}_p \cap \mathcal{A}_s$ **do**
      `C[j]` $\leftarrow \max($`C[j]`$, A_j.y - A_j.x + 1 + \mathcal{U}^{\nearrow}$.`rMq`$(0, A_j.x - 1), A_j.y + \mathcal{U}^{\cap}$.`rMq`$(A_j.x, A_j.y))$
    for $A_j \in \mathcal{A}_p \cap \mathcal{A}^s$ **do**
      $\mathcal{U}^{\nearrow}$.`update`$(A_j.y,$ `C[j]`$)$
      $\mathcal{U}^{\cap}$.`update`$(A_j.y,$ `C[j]`$ - A_j.y)$
  for $A_j \in \mathcal{A}^s$ **do**
    for $P_i \in \{P \in \mathcal{P} \mid s \in P\}$ **do**
      $\mathcal{D}_i^{\nearrow}$.`update`$(A_j.y,$ `C[j]`$)$
      $\mathcal{D}_i^{\cap}$.`update`$(A_j.y,$ `C[j]`$ - A_j.y)$
  for $(t, P_i) \in$ `forward[s]` **do**
    for $A_j \in \mathcal{A}_t$ **do**
      `C[j]` $\leftarrow \max($`C[j]`$, A_j.y - A_j.x + 1 + \mathcal{D}_i^{\nearrow}$.`rMq`$(0, A_j.x - 1), A_j.y + \mathcal{D}_i^{\cap}$.`rMq`$(A_j.x, A_j.y))$
return Maximum coverage chain $\mathcal{C}$ recovered from `C`

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 $A_j.p$, 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. Moreover, to avoid chaining anchors with the same $A.v$ and $A.p$, we process all anchors with the same $A.p$ value together, that is, we first apply

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

Step 0.1 to all such anchors and then **Step 0.2** to all such anchors. Algorithm 1 shows the pseudocode of the algorithm. Since every anchor requires at most two 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 $O(k(|V| + |E|) \log |V| + kN \log N)$.

In the Appendix we explain how to adapt Algorithm 1 to obtain $O(k^3|V| + k|E|)$ time for pre-processing and $O(kN \log kN)$ time for solving co-linear chaining. Also in the Appendix, we show how to solve Problem 2 but removing the restriction of G to be acyclic.

2.4 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 few fragments to align increases the speed, but decreases the accuracy. These fragments are then aligned with `GraphAligner` to the variation graph[6], with default parameters; for each alignment reported by `GraphAligner`, we obtain an anchor made up of the reported path in the graph (including the offset in the last node $A.p$), and the fragment interval in the long read.

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 [31] for its simplicity, where for its max-flow routine we implemented Dinic's algorithm [9]. The (range maximum query) data structures that our algorithm maintains per MPC path are supported by our own thread-safe implementation of a treap [38].

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 $A_i.v$. A more principled approach would be to connect consecutive anchors by a path minimizing edit distance, or to consider such distances as part of the optimization function of co-linear chaining. We use BFS for performance reasons.

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

Finally, `GraphChainer` uses `edlib` [43] to compute an alignment between the read and the path found, and to decide if this alignment is better than the (best) alignment reported by `GraphAligner`. Therefore, `GraphChainer` can be also be viewed as a refinement of `GraphAligner`'s alignment results.

3 Experiments

We run several erroneous 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. We also excluded `PaSGAL` [22] and the recent extension of `AStarix` for long reads [20], since these aligners are tailored to find optimal alignments and they were three orders of magnitude slower than `GraphChainer` in our smallest dataset.

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

Table 1: Statistics of every dataset used in our experiments. Every graph has a read set simulated on a random path of the graph. Graphs Chr22 and Chr1 were built using GRCh37 as reference and variants from the Thousand Genomes Project, these graphs have an additional PacBio read set (rows "real") obtained from SRA. Column k is the width of the graph.

Graph	#Nodes	Labels bps	k	#Reads	Total read bps	Cov
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	52 464	769 238 818	15×
real				136 494	2 858 621 416	56×
Chr1	18 807 963	255 754 179	9	254 251	3 736 803 386	15×
real				907 572	19 617 046 919	79×

3.1 Datasets

Variation graphs. We use two (relatively) small variation graphs and two chromosome-level variation graphs. The small graphs, LRC and MHC1 [22], correspond to two of the most diverse variation regions in the human genome [8, 41]. The chromosome-level graphs, Chr22 and Chr1 (human chromosomes 22 and 1, respectively), were built with the `vg toolkit` using GRCh37 as reference, and variants from the Thousand Genomes Project phase 3 release [5]. We use Chr22 to replicate GraphAligner's results [35], and consider Chr1 since it is one of the most complex variation graphs of the human chromosomes (see Table 5 in the Appendix).

Note that acyclic variation graphs (built as above) span all genomic positions, i.e., do not collapse repeats like, e.g., de Bruijn graphs. As such, the aligner does not perform extra steps to identify the corresponding repeat of an alignment but instead they are solved by the alignment task itself.

Simulated reads. For each of the previous graphs, we sample a reference sequence and use it to simulate a PacBio read dataset of 15x coverage and average error rate of 15% with the package `Badread` [46]. 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 location on the graph.

Real reads on chromosome-level graphs. For the chromosome-level grpahs, we also used the whole human genome PacBio Sequel data from HG00733 (SRA accession SRX4480530)[7]. 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. 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.

3.2 Evaluation metrics and experimental setup

For each read, the aligners output an alignment consisting of a path in the graph and a sequence of edits to perform on the node sequences. We call this path the *reported path* and the concatenation of the node

[7]This is the same read set used by `GraphAligner`'s experiments [35, p.7] but without the subsampling to 15× coverage.

Table 2: Correctly aligned reads with respect to the overlap for $\delta \in \{0.1, 0.85\}$ (i.e., the overlap between the reported path and the ground truth is at least 10% or 85% of the length of the ground truth sequence, respectively) for the simulated read sets. Percentages in parentheses are relative improvements w.r.t. `GraphAligner`.

Graph	Aligner	Correctly aligned	
		$\delta = 0.1$	$\delta = 0.85$
LRC	`GraphAligner`	95.15%	91.22%
	`GraphChainer`	98.72% (+3.75%)	97.90% (+7.32%)
MHC1	`GraphAligner`	96.15%	92.52%
	`GraphChainer`	98.98% (+2.94%)	98.17% (+6.11%)
Chr22	`GraphAligner`	95.78%	92.56%
	`GraphChainer`	99.06% (+3.42%)	98.00% (+5.88%)
Chr1	`GraphAligner`	95.44%	92.26%
	`GraphChainer`	98.61% (+3.32%)	96.68% (+4.79%)

Table 3: Correctly aligned reads with respect to the distance, and percentage of read length in correctly aligned reads, for $\sigma_{read} = 0.3$ (i.e., the edit distance between the read and the reported sequence can be up to 30% of the read length) for real PacBio read sets. Percentages in parentheses are relative improvements w.r.t. `GraphAligner`.

Graph	Aligner	Correctly aligned	Good length
Chr22 (real)	`GraphAligner`	82.45%	74.84%
	`GraphChainer`	96.48% (+17.02%)	96.30% (+28.68%)
	`--speed 2`	95.31% (+15.59%)	95.01% (+26.95%)
	`--speed 3`	94.34% (+14.41%)	93.82% (+25.36%)
Chr1 (real)	`GraphAligner`	84.27%	77.69%
	`GraphChainer`	95.01% (+12.74%)	94.63% (+21.80%)
	`--speed 2`	93.12% (+10.50%)	92.08% (+18.51%)
	`--speed 3`	91.75% (+8.87%)	89.95% (+15.77%)

sequences without the edits the *reported sequence*[8].

On simulated data, we classify a read as *correctly aligned* if the overlap (in genomic sequences) between the reported path and the ground truth path is at least $(100 \cdot \delta)\%$ of the length of the ground truth sequence, where $0 < \delta \leq 1$ is a given threshold. As another criterion, we consider a read correctly aligned if the edit distance between the reported sequence and ground truth sequence is at most $(100 \cdot \sigma)\%$ of the length of the ground truth sequence, where $0 < \sigma \leq 1$ is another given threshold. On real data[9], where the ground truth is not available, we consider a read correctly aligned if the edit distance between the reported sequence and read is at most $(100 \cdot \sigma)\%$ of the read length.

Since the reads have varying sizes, we also computed the *good length*, defined as the total length of correctly aligned reads divided by the total read length, for every criterion and threshold considered.

All experiments were conducted on a server with AMD Ryzen Threadripper PRO 3975WX CPU with 32 cores and 504GB of RAM. Both 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

Simulated reads. For criterion $\delta = 0.85$, that is matching the average identity of the simulated reads[10], `GraphChainer` has at least 4–5% more correctly aligned reads, and at the weaker criterion $\delta = 0.1$, used in `GraphAligner`'s evaluation [35, p.3], `GraphChainer` correctly aligns 3–4% more reads, see Table 2, which is true for every criterion $\delta > 0$, see Figure 2. Moreover, we observe that the precision of `GraphAligner` drops below 95% for $\delta > 0.6$, whereas this does not happen to `GraphChainer` until $\delta > 0.98$. Similar results

[8] Both objects, as well as the ground truth path/sequence, exclude the offsets of the first and last nodes.
[9] Also on simulated data to show the relation between the criteria. This can be found in the Appendix.
[10] Recall that the average error rate is set to 85%.

Table 4: Running time (in format hh:mm:ss) and peak memory (Mem, in GBs) when aligning 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
Chr22 (real)	`GraphAligner`	7.92	02:00:03	00:04:12
	`GraphChainer`	12.58	04:48:09	00:09:56
	`--speed 2`	12.94	03:48:41	00:07:57
	`--speed 3`	13.14	03:30:07	00:07:20
Chr1 (real)	`GraphAligner`	18.66	13:48:46	00:28:42
	`GraphChainer`	58.47	73:05:00	02:28:12
	`--speed 2`	58.59	46:10:51	01:34:21
	`--speed 3`	58.78	37:32:20	01:17:05

are obtained when measuring good length and using edit distance criteria, see Tables 8 to 10 and Figures 5 to 12 in the Appendix.

Real reads on chromosome-level graphs (Chr22 and Chr1). In this case the difference between `GraphAligner` and `GraphChainer` is even clearer, see Figure 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 3, for criterion $\sigma = 0.3$, `GraphChainer`'s improvements in correctly aligned reads, and in total length of correctly aligned reads, are up to 17.02%, and 28.68%, respectively for Chr 22. For Chr1, the improvement in the two metrics is smaller, up to 12.74%, and 21.80%, but still significant. We also note that `GraphChainer` reaches a precision of 95% for $\sigma < 0.3$, whereas this happens at $\sigma > 0.5$ for `GraphAligner`.

Performance. Table 4 shows the running time and peak memory of both aligners on real reads and Table 7 (Appendix) on simulated reads. 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. Running times are larger on real reads 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, while the running time shrinks by up to half. Table 6 in the Appendix shows that one bottleneck of `GraphChainer` is obtaining the anchors, which could be improved by using a fast short-read graph aligner or MEMs [37] instead.

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 the more-erroneous long reads and much less work has been published around this problem. `GraphAligner` [35] 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 string labeled DAG and applied it to aligning erroneous long reads to acyclic 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 [29], is a useful technique also in the context of variation graphs. Moreover, the fact that human variation graphs have small width (Table 5), combined with our co-linear chaining algorithm allowing one-node overlaps to handle string labeled nodes, and 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.

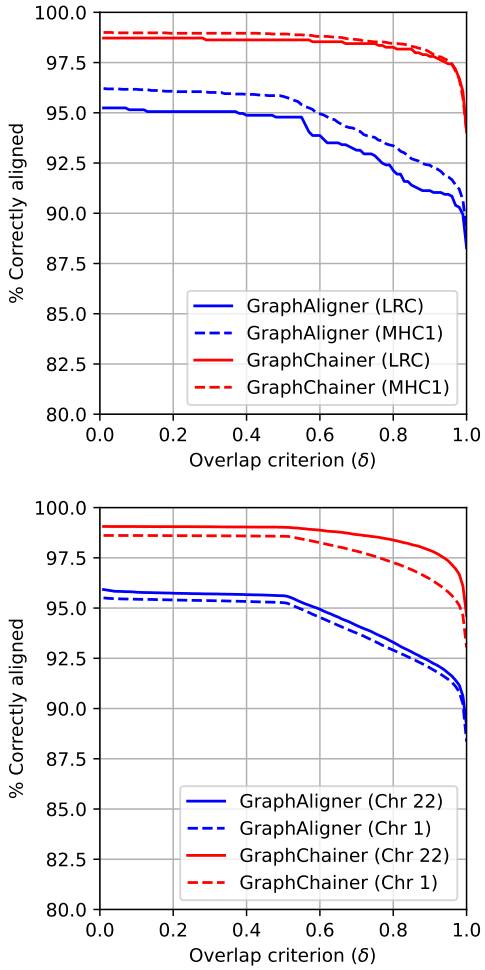

Figure 2: Correctly aligned reads w.r.t overlap with ground truth on the simulated read sets for LRC (top solid), MHC1 (top dashed), Chr22 (bottom solid) and Chr1 (bottom dashed).

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).

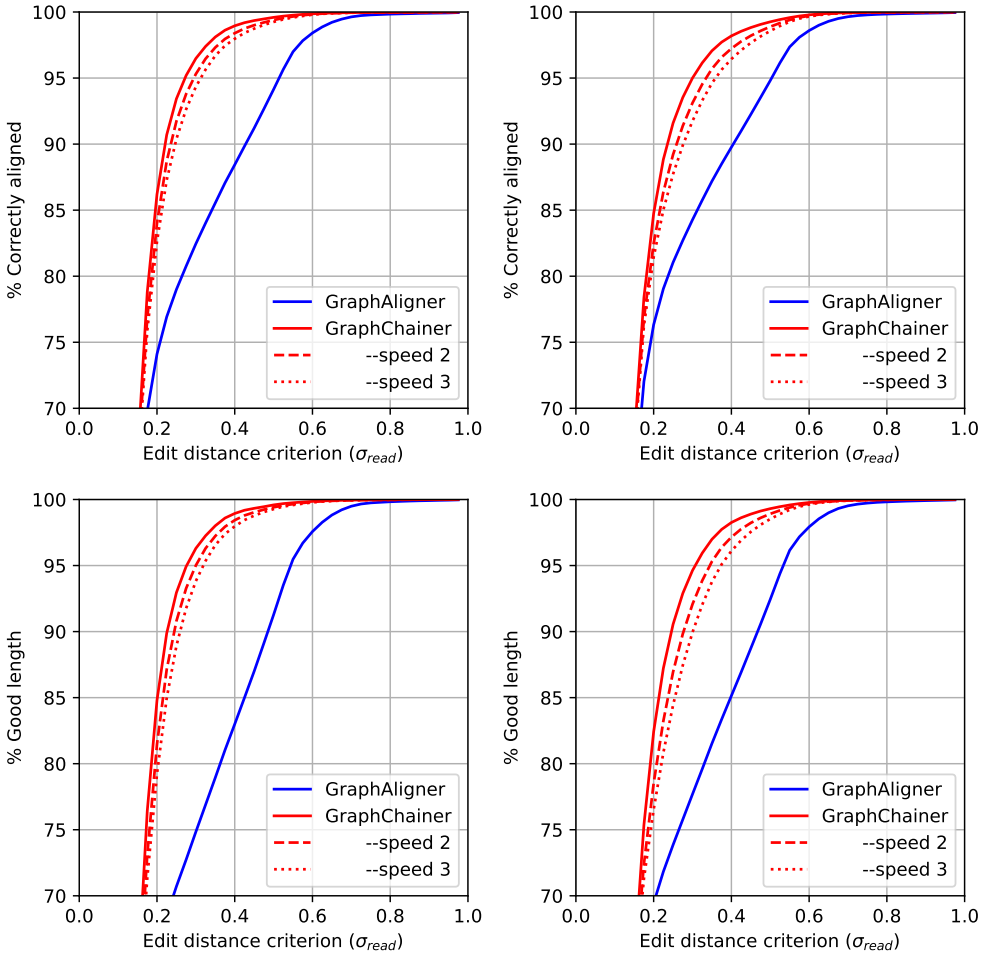

Figure 3: Correctly aligned reads w.r.t. the read distance (top), and read length in correctly aligned reads (bottom), on Chr22 (left) and Chr1 (right).

[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] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[17] 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.

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

[19] 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.

[20] 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.

[21] Chirag Jain, Daniel Gibney, and Sharma V Thankachan. Co-linear chaining with overlaps and gap costs. *bioRxiv*, 2021. To appear in RECOMB 2022.

[22] 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.

[23] 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.

[24] John D Kececioglu and Eugene W Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1):7–51, 1995.

[25] 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.

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

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

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

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

[30] Veli Mäkinen and Kristoffer Sahlin. Chaining with Overlaps Revisited. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:12, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2020/12150, doi:10.4230/LIPIcs.CPM.2020.25.

[31] 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.

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

[33] 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.

[34] 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.

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

[36] 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.

[37] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A Pangenomic Index for Finding Maximal Exact Matches. *Journal of Computational Biology*, 2022.

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

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

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

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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 5: 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	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 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, we decided to divide the algorithm into pre-processing the DAG, that we solve in $O(k^3|V|+k|E|)$ time by using a recent result to compute a minimum path cover of a DAG [4], and chaining that we solve in time $O(kN \log kN)$[11].

To remove the chaining's dependency on the graph size we get rid of the for loops of Algorithm 1 by instead processing 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

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

path endpoints[12], which can be done in $O(N \log N)$ time (by previously computing the topological ordering on pre-processing 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 $(A_j.u, P_i) \in$ `forward[s]`) by topological order of s. Since there can be $O(kN)$ such tuples, this takes $O(kN \log kN)$ time.

C 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 2 but we do not restrict G 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 [44, 7, 39]. 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 $f_{\mathcal{G}}$, 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 $A_{i_{j+1}}.u$. □

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 $O(|V| + |E| + N + k'(|\mathcal{V}| + |\mathcal{E}|) \log |\mathcal{V}| + k'N \log N) = O(k(|V| + |E|) \log |V| + kN \log N)$.

D 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
```

[12] If there are ties, they are sorted by endpoint in the string label of the path endpoint. This is done in this case as well as the other sortings, but we remove it from the phrasing by ease of explanation.

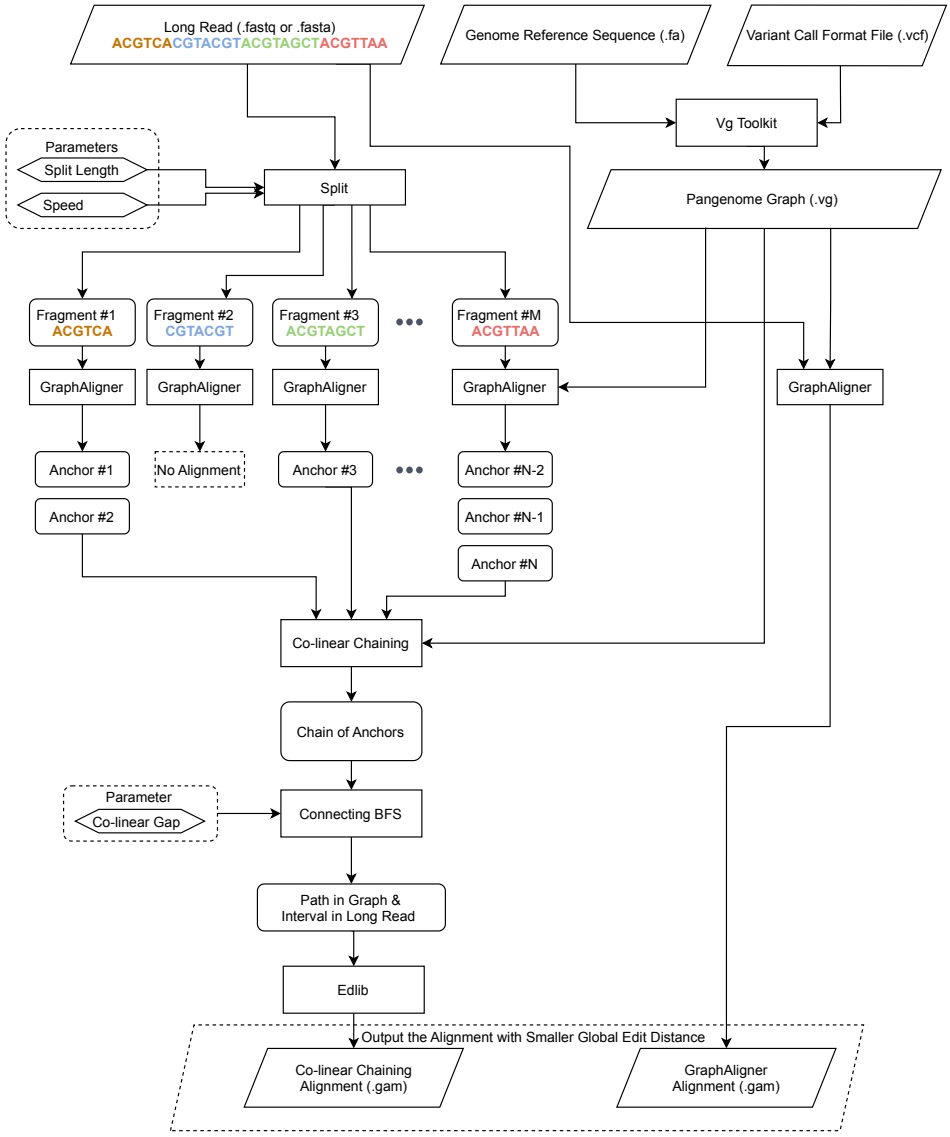

Figure 4: The flow diagram of `GraphChainer`: a long-read is split into fragments (of default length 35), which are aligned with `GraphAligner` (though any other method to extract anchors can be used). Anchors are created for each alignment of a fragment. The anchors are chained with our algorithm from 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.

Variation graph construction with vg toolkit (same parameters as those used for `GraphAligner`'s experiments [35, 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 [35, p.24] but using 30 threads, default parameters for `PaSGAL` [22], same parameters for the extension of `AStarix` on its experiments for long reads [20, p.13]):

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

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

PaSGAL -m vg -r {Graph} -q {Reads} -t 30
-o {output_file}

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

E Further experimental results

Table 6: CPU time spent in the main phases of `GraphChainer` on Chr22 and Chr1 (in format hh:mm:ss) on the real PacBio reads. 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

Table 7: Running time (in format hh:mm:ss) and peak memory (Mem, in GBs) when aligning simulated 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`	1.19	00:00:37	00:00:01
	`GraphChainer`	1.54	00:01:01	00:00:02
MHC1	`GraphAligner`	2.43	00:02:56	00:00:07
	`GraphChainer`	3.33	00:04:43	00:00:12
Chr22	`GraphAligner`	5.81	00:28:20	00:01:08
	`GraphChainer`	11.38	00:55:43	00:02:11
Chr1	`GraphAligner`	16.95	02:29:08	00:06:03
	`GraphChainer`	58.14	11:38:42	00:25:16

Table 8: Read length in correctly aligned reads with respect to the overlap for $\delta \in \{0.1, 0.85\}$ (i.e., the overlap between the reported path and the ground truth is at least 10% or 85% of the length of the ground truth sequence, respectively) for the simulated read sets. Percentages in parentheses are relative improvements w.r.t. `GraphAligner`.

Graph	Aligner	Good length	
		$\delta = 0.1$	$\delta = 0.85$
LRC	`GraphAligner`	95.67%	90.60%
	`GraphChainer`	99.52% (+4.02%)	98.18% (+8.37%)
MHC1	`GraphAligner`	96.80%	91.70%
	`GraphChainer`	99.50% (+2.79%)	98.44% (+7.36%)
Chr22	`GraphAligner`	96.26%	91.91%
	`GraphChainer`	99.46% (+3.33%)	98.19% (+6.83%)
Chr1	`GraphAligner`	96.00%	91.58%
	`GraphChainer`	99.17% (+3.30%)	96.73% (+5.62%)

Table 9: Correctly aligned reads with respect to the distance, for $\sigma_{truth} = 0.3$ (i.e., the edit distance between the truth sequence and the reported sequence can be up to 30% of the truth sequence length), and $\sigma_{read} = 0.3$ (i.e., the edit distance between the read and the reported sequence can be up to 30% of the read length) for simulated read sets. Percentages in parentheses are relative improvements w.r.t. `GraphAligner`.

Graph	Aligner	Correctly aligned	
		$\sigma_{truth} = 0.3$	$\sigma_{read} = 0.3$
LRC	`GraphAligner`	93.32%	91.40%
	`GraphChainer`	98.72% (+5.78%)	96.25% (+5.31%)
MHC1	`GraphAligner`	94.03%	92.73%
	`GraphChainer`	98.82% (+5.10%)	96.64 (+4.22%)
Chr22	`GraphAligner`	94.22%	92.67%
	`GraphChainer`	98.86% (+4.93%)	96.43% (+4.06%)
Chr1	`GraphAligner`	94.04%	92.57%
	`GraphChainer`	98.19% (+4.41%)	95.73% (+3.42%)

Table 10: Read length in correctly aligned reads with respect to the distance, for $\sigma_{truth} = 0.3$ (i.e., the edit distance between the truth sequence and the reported sequence can be up to 30% of the truth sequence length), and $\sigma_{read} = 0.3$ (i.e., the edit distance between the read and the reported sequence can be up to 30% of the read length) for simulated read sets. Percentages in parentheses are relative improvements w.r.t. `GraphAligner`.

Graph	Aligner	Good length	
		$\sigma_{truth} = 0.3$	$\sigma_{read} = 0.3$
LRC	`GraphAligner`	92.79%	90.47%
	`GraphChainer`	99.08% (+6.78%)	95.87% (+5.97%)
MHC1	`GraphAligner`	93.42%	92.06%
	`GraphChainer`	99.00% (+5.97%)	96.76% (+5.10%)
Chr22	`GraphAligner`	93.97%	92.03%
	`GraphChainer`	99.05% (+5.41%)	96.35% (+4.69%)
Chr1	`GraphAligner`	93.72%	91.87%
	`GraphChainer`	98.39% (+4.99%)	95.58% (+4.03%)

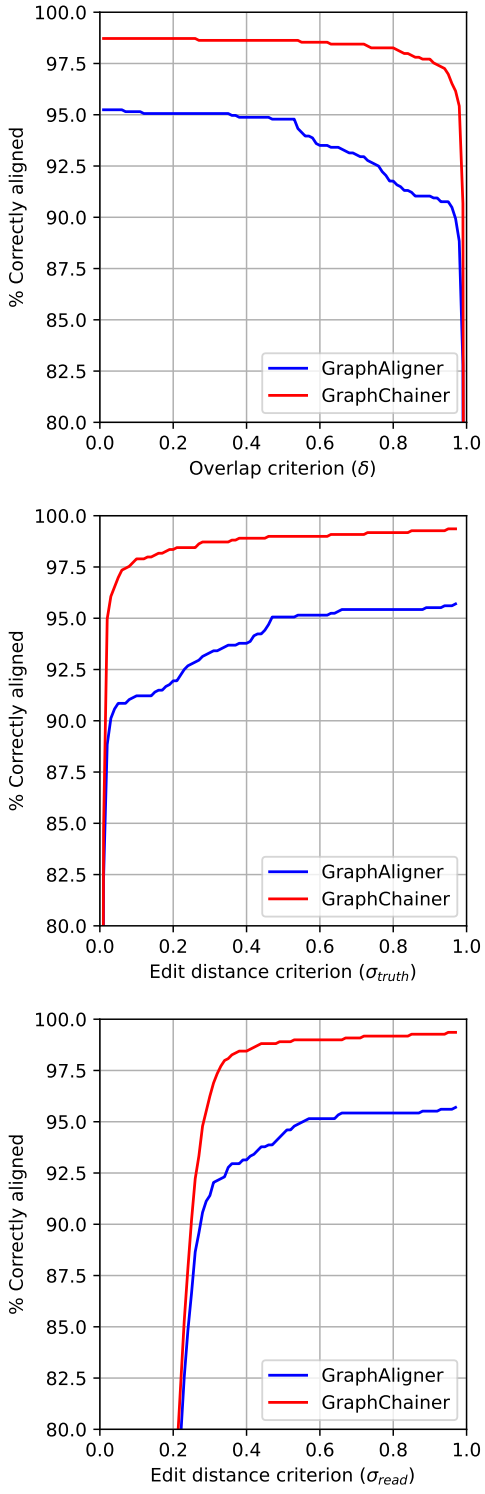

Figure 5: Correctly aligned reads w.r.t overlap (top), truth sequence distance (middle) (middle) and read distance (bottom) for LRC on simulated reads.

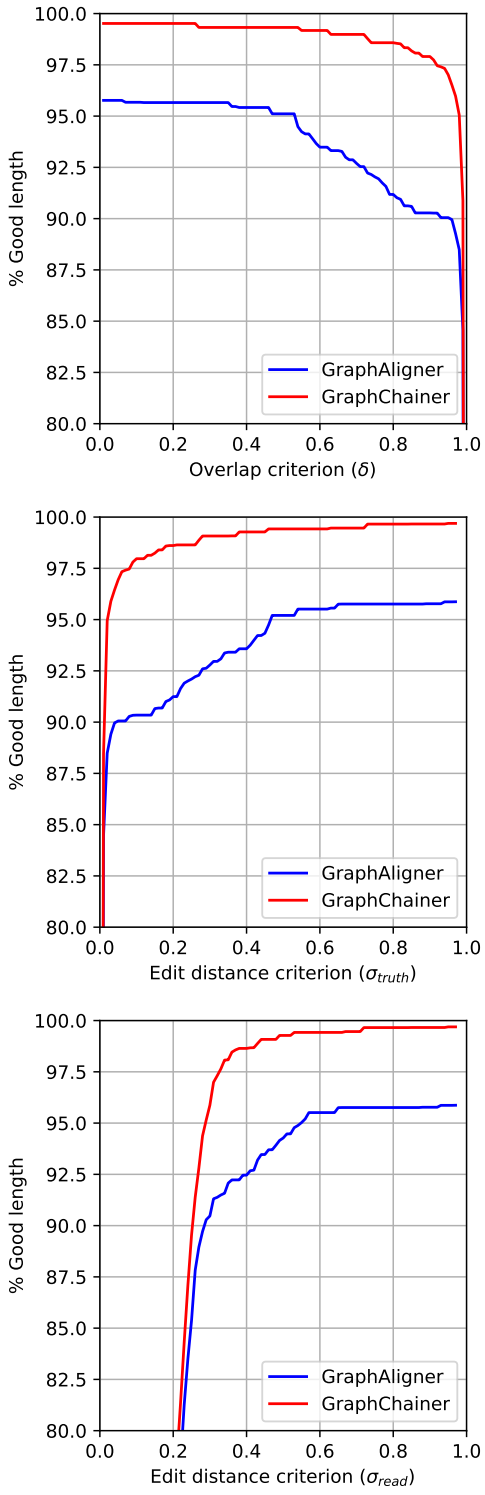

Figure 6: Read length in correctly aligned reads w.r.t overlap (top), truth sequence distance (middle) and read distance (bottom) for LRC on simulated reads.

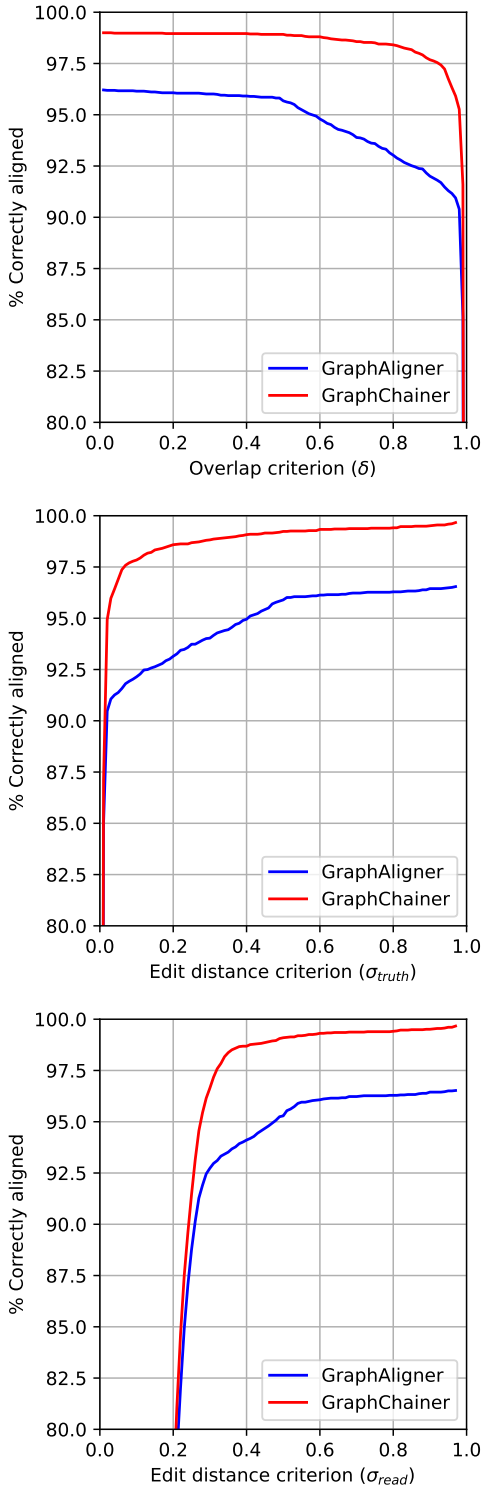

Figure 7: Correctly aligned reads w.r.t overlap (top), truth sequence distance (middle) and read distance (bottom) for MHC1 on simulated reads.

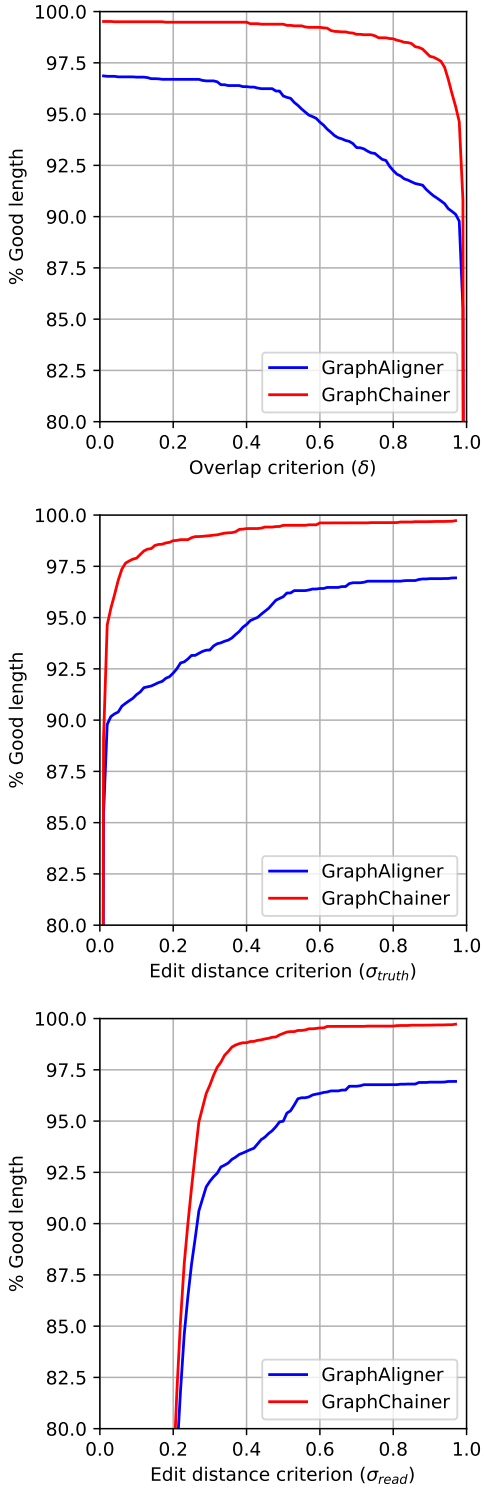

Figure 8: Read length in correctly aligned reads w.r.t overlap (top), truth sequence distance (middle) and read distance (bottom) for MHC1 on simulated reads.

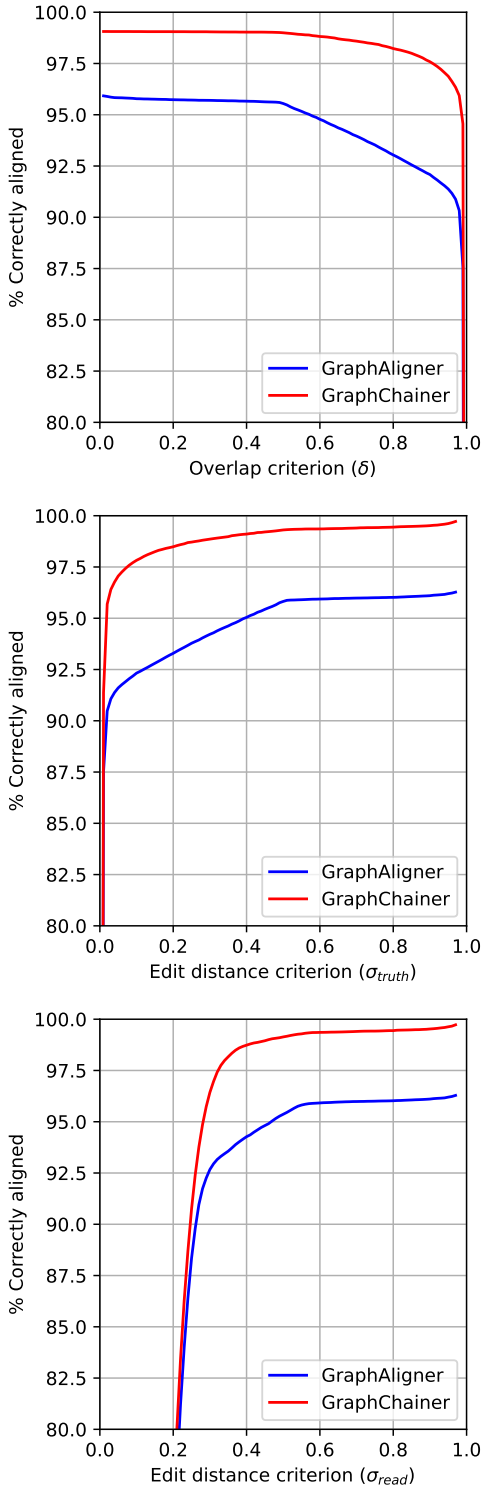

Figure 9: Correctly aligned reads w.r.t overlap (top), truth sequence distance (middle) and read distance (bottom) for Chr22 on simulated reads.

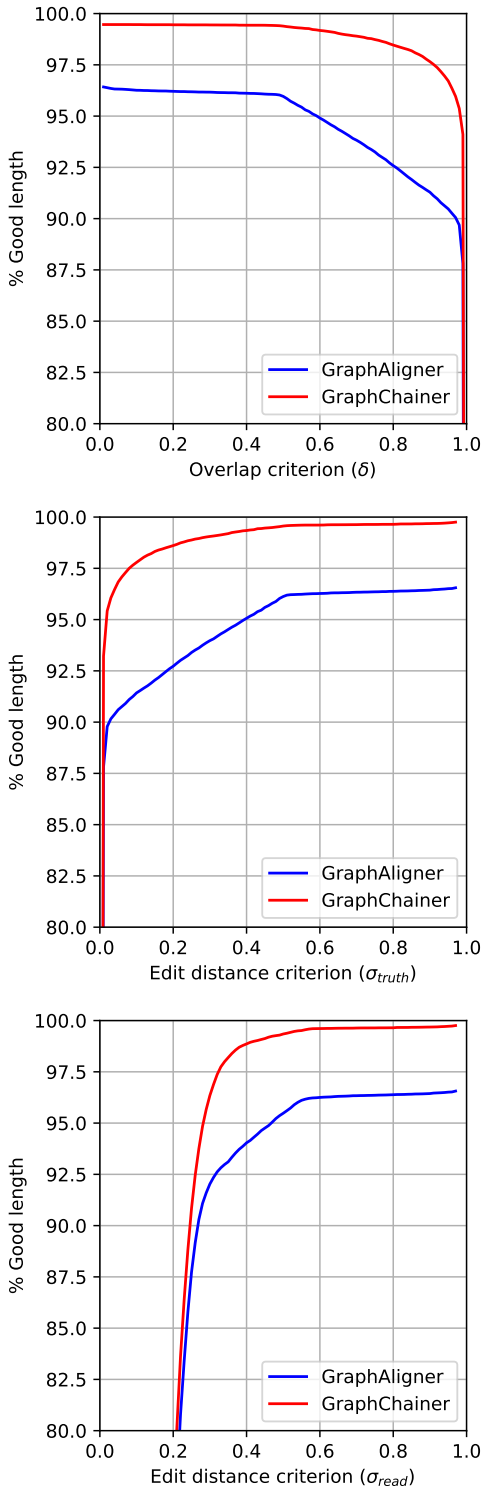

Figure 10: Read length in correctly aligned reads w.r.t overlap (top), truth sequence distance (middle) and read distance (bottom) for Chr22 on simulated reads.

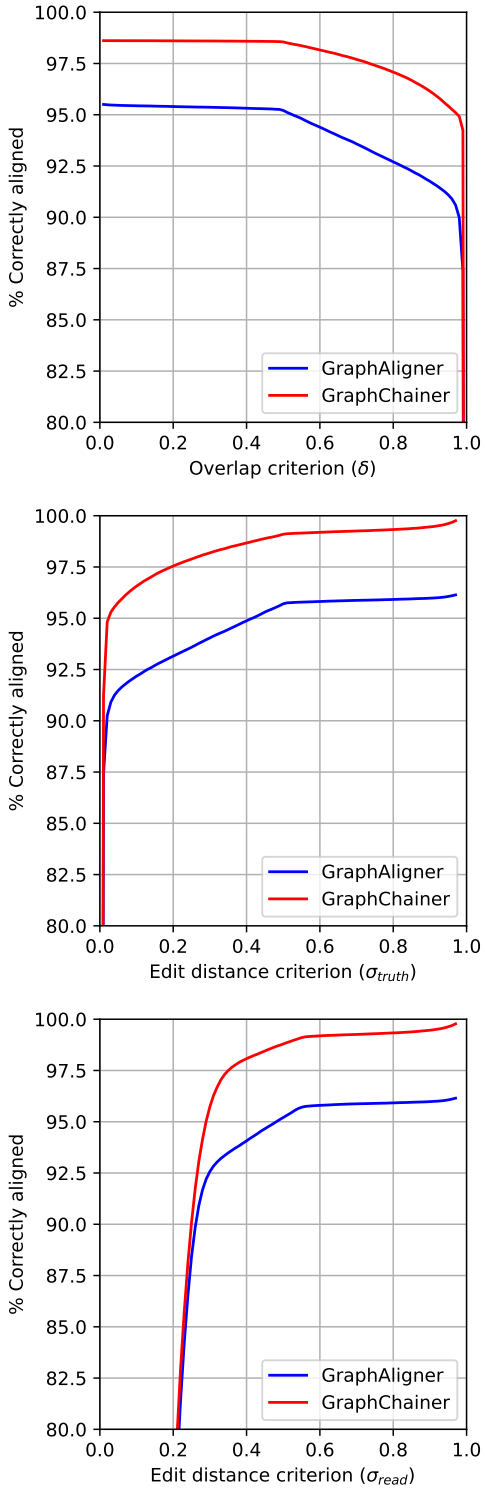

Figure 11: Correctly aligned reads w.r.t overlap (top), truth sequence distance (middle) and read distance (bottom) for Chr1 on simulated reads.

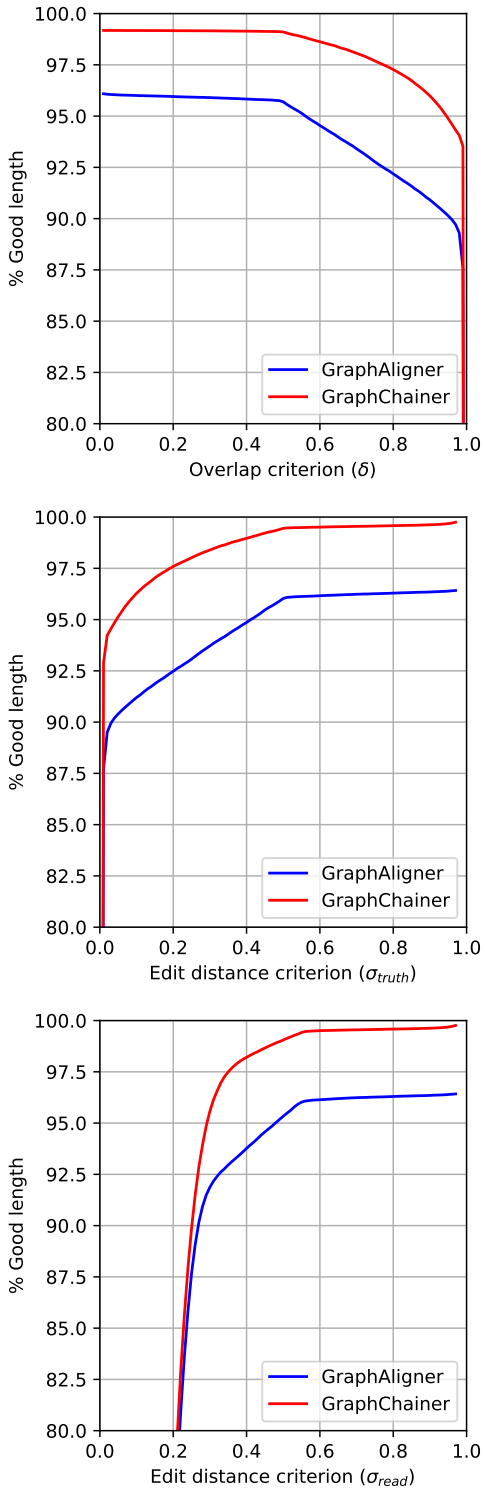

Figure 12: Read length in correctly aligned reads w.r.t overlap (top), truth sequence distance (middle) and read distance (bottom) for Chr1 on simulated reads.