Master's thesis

Master's Programme in Computer Science

Co-linear Chaining on Graphs With Cycles

Jun Ma

May 26, 2021

Faculty of Science

University of Helsinki

Supervisor(s)

Assoc. Prof. Alexandru I. Tomescu
MSc. Manuel Caćeres

Examiner(s)

Prof. Veli Mäkinen
Assoc. Prof. Alexandru I. Tomescu

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Jun Ma

Työn nimi — Arbetets titel — Title

Co-linear Chaining on Graphs With Cycles

Ohjaajat — Handledare — Supervisors

Assoc. Prof. Alexandru I. Tomescu
MSc. Manuel Cáceres

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	May 26, 2021	43 pages

Tiivistelmä — Referat — Abstract

Sequence alignment by exact or approximate string matching is one of the fundamental problems in bioinformatics. As the volume of sequenced genomes grows rapidly, pairwise sequence alignment becomes inefficient for pan-genomic analyses involving multiple sequences. The graph representation of multiple genomes has been an increasingly useful tool in pan-genomics research. Therefore, sequence-to-graph alignment becomes an important and challenging problem.

For pairwise approximate sequence alignment under Levenshtein (edit) distance, subquadratic algorithms for finding an optimal solution are unknown. As a result, aligning sequences of millions of characters optimally is too challenging and impractical. Thus, many heuristics and techniques are developed for possibly suboptimal alignments. Among them, co-linear chaining (CLC) is a powerful and popular technique that approximates the alignment by finding a chain of short aligned fragments that may come from exact matching. The optimal solution to CLC on sequences can be found efficiently in subquadratic time. For sequence-to-graph alignment, the CLC problem has been solved theoretically on a special class of graphs that are narrow and have no cycles, i.e. directed acyclic graphs (DAGs) with small width, by Mäkinen et al. (ACM Transactions on Algorithms, 2019). Pan-genome graphs such as variation graphs satisfy these restrictions but allowing cycles may enable more general applications of the algorithm.

In this thesis, we introduce an efficient algorithm to solve the CLC problem on general graphs with small width that may have cycles, by reducing it to a slightly modified CLC problem on DAGs. We implemented an initial version of the new algorithm on DAGs as a sequence-to-graph aligner `GraphChainer`. The aligner is evaluated and compared to an existing state-of-the-art aligner `GraphAligner` (Genome Biology, 2020) in experiments using both simulated and real genome assembly data on variation graphs. Our method improves the quality of alignments significantly in the task of aligning real human PacBio data. `GraphChainer` is freely available as an open source tool at https://github.com/algbio/GraphChainer.

ACM Computing Classification System (CCS)
Applied computing → Life and medical → Genomics → Computational genomics

Avainsanat — Nyckelord — Keywords

pan-genome, graph algorithm

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms study track

Contents

1 Introduction

Sequence alignment by exact or approximate string matching is one of the fundamental problems in bioinformatics. The basic alignment problem asks for an exact match of one string in another reference string. A more general problem asks for an approximate alignment that allows differences in the matching. One of such approximate alignment method uses the Levenshtein (edit) distance [18], which equals the minimal number of insertions, deletions and mismatches between the two sequences. The edit distance between two sequences of length n can be computed exactly in $O(n^2)$ time [16], but algorithms to compute this distance exactly in wrost-case subquadratic time are unknown.

As a result, computation of the optimal alignments between long genome sequences that usually have millions of characters is too challenging and impractical. Thus, many heuristics and techniques are developed for suboptimal alignments. Among them, co-linear chaining (CLC) is a powerful and popular technique that approximates the alignment by finding a chain of short aligned fragments that may come from exact matching. For CLC on sequences, the optimal solution can be found efficiently in subquadratic time.

On the other hand, as the volume of sequenced genomes grows rapidly, pairwise sequence alignment becomes inefficient for pan-genomic analyses involving multiple sequences. The graph representation of multiple genomes has been an increasingly useful tool in pan-genomics research. One of the common practices is to use directed labelled graphs to represent multiple genomes. Therefore, sequence-to-graph alignment becomes an important and challenging problem.

In this thesis, we introduce an efficient algorithm to solve the CLC problem on general graphs with small width that may have cycles. This algorithm is based on the *sparse dynamic programming framework* of [21], which solves the CLC problem efficiently on a special class of graphs that are narrow and have no cycles, i.e. directed acyclic graphs (DAGs) with small width. Our algorithm reduces the CLC problem on general graphs to CLC on DAGs, under the same assumption that the graphs are narrow.

We also implemented an initial version of the new algorithm on DAGs as a sequence-to-graph aligner `GraphChainer`. For now, this implementation assumes that input graphs are DAGs, but has the potential to be modified to accept graphs with cycles in the future. The aligner is evaluated and compared to an existing state-of-the-art aligner `GraphAligner` [27]

in experiments using both simulated and real genome assembly data on variation graphs. Our method improves the quality of alignments significantly in the task of aligning real human PacBio data.

In the following sections, we will introduce the backgrounds of the problem. In Section 1.1, we examine the approximate sequence alignment problems and the co-linear chaining technique on sequences. In Section 1.2, we introduce the concepts of *pan-genome graphs* and the properties of such graphs. In Section 1.3, we review the *sparse dynamic programming framework* of [21] which is the an important ingredient in our algorithm. At last, in Section 1.4, the contributions of this thesis are outlined.

1.1 Co-linear Chaining

In bioinformatics *approximate sequence alignment* has more applications than *exact sequence alignment*. For example, when mapping short read sequences against a reference sequence in genome assembly, it is critical to allow errors in the alignments since the reads are not always accurate and the sequence to reconstruct may have mutations so that no exact match exists. Approximate sequence alignment searches for an alignment between the two sequences with the smallest distance such as *Levenshtein distance* (edit distance) [18]. Figure 1.1 shows an example of approximate alignment under edit distance, defined as the minimum number of operations to change one sequence to another by inserting, deleting and changing one character at a time.

```
--TGGGGTTTCGC--TGGTCGCAGTACT-CAGACATAACATTTC
  ||||||||||    ||||||||  |||  |  |  |||||||||
GGTGGGGTTTCGGTTTGGTCGCAA-ACTTC-G--ATAACATTTA
```

Figure 1.1: An example of global edit distance alignment. A dash symbol represents a gap. There are 5 insertions (green), 4 deletions (blue) and 3 mismatches (red) in the shown alignment. This alignment is optimal but not unique. The edit distance is $5 + 4 + 3 = 12$.

The pairwise approximate sequence alignment can be computed by dynamic programming in $O(n^2)$ for two sequences of length n [16]. However, an algorithm faster than the worst-case quadratic complexity is unknown and might be unlikely to exist. As proven in [3], the optimal edit distance alignment between two sequences cannot be computed in time $O(n^{2-\epsilon})$ for any $\epsilon > 0$ unless the *Strong Exponential Time Hypothesis* (SETH) is false . In the context of bioinformatics, where the sequence may consist of millions or billions of

base pairs, the quadratic complexity in time and especially in space is often not practical.

Therefore many heuristics arise to speed up sequence alignment in practice, such as seeding by minimizers [28, 29], maximal unique matches (MUMs) as in MUMmer [8, 9]. These short alignments are not very informative on their own but can be extended to an alignment. Instead of computing alignments by extending from a single seed, we can also chain several seeds, which leads to the *co-linear chaining* (CLC) problem.

In the CLC problem, the input is a set of anchors. An *anchor* is a matching pair of intervals on each of the two sequences, and the anchor may have a weight. The goal is to compute an ordered subset of anchors so that adjacent anchors satisfy some co-linear conditions and the chain is optimal under some target evaluation function. For example, we can look for the longest chain, where adjacent anchors have no overlaps, and both of their intervals appear in increasing order. Depending on the various definitions of the co-linear condition or the precedence relation and the target function, CLC is very flexible. As shown in Figure 1.2, CLC can rule out anchors that cannot be chained with others if overlapping is not allowed.

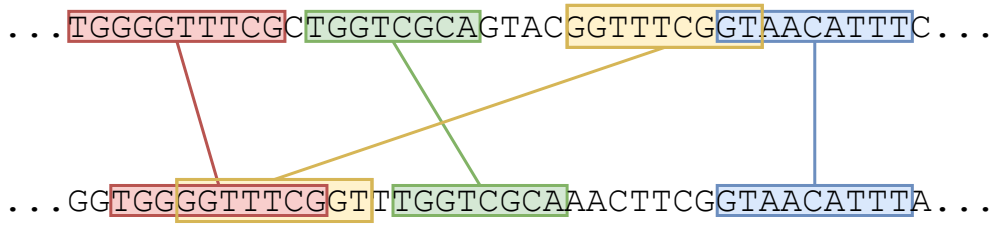

Figure 1.2: An example of co-linear chaining on two sequences. Each anchor is a matching pair of intervals. The three anchors colored in red, greed and blue can be chained together in order. The yellow anchor cannot be chained with any other anchor. Note that the blue anchor can extend to the left so it is not maximal.

The advantage of CLC is that a CLC problem usually has efficient algorithms for the optimal solutions under the definition, unlike other heuristic alignment strategies. The chaining algorithm in [11] only requires $O(N \log N)$ time, where N is the number of anchors.

CLC can be easily extended to higher dimensions to support approximate multiple sequence alignment. An anchor is now a tuple of matching intervals on each sequence. Two anchors to be chained together need to have all of their parts appearing in the order. For k sequences, CLC can be solved in $O(N \log^k N)$ time and $O(N \log^{k-1} N)$ space [23].

In addition, CLC can also be extended to the sequence-to-graph alignment with anchors

modified accordingly. On graphs, an anchor is a pair of a path on the graph, and an interval on the sequence, indicating that the sequence of this path matches the interval sequence.

1.2 Pan-Genome Graphs

As a natural extension of sequences, labelled graphs are widely used in bioinformatics such as in genome assembly [2, 4] and error correction [30].

A *pan-genome graph* also called a *variation graph*, is a reference graph containing all variants of a population, so that each path in the graph is a potential genome. Compared to a single reference sequence, a pan-genome graph considers the variants between different individuals. In addition, it enables the combinations of several variants at different sites which do not necessarily appear in any of the sequences to build the graph, which further increases the robustness. Another advantage of aligning sequences against graphs is the efficiency. The sequences of interest usually are similar to each other, for example, the genetic difference between individual humans is about 20 millions of base pairs, which is about 0.6% of the total of 3.2 billion base pairs [6]. As a result, aligning against a merged graph of these sequences can be hundreds of times faster than aligning against each sequence separately. A variation graph in practice is usually a labelled DAG, which is a directed labeled graph that has no cycles.

Aligning sequences against graphs is fundamental in applications utilizing graph representations. Similar to the sequence only cases, approximate matching under edit distance or affine cost would be useful in practice. The algorithms to exact solution such as the bit-parallel algorithm [26] extended from the sequence version [22], has a worst case complexity of $O(|V|+m|E|\log w)$ for a general graph with $|V|$ nodes and $|E|$ edges and a sequence of length m, or $O(|V|+\left\lceil\frac{m}{w}\right\rceil|E|\log w)$ for a DAG, where w is the word size of machine (commonly 32 or 64). In areas other than bioinformatics, approximate sequence-to-graph alignment also has broad applications and many solutions, such as in the hypertext searching [24], but the scale of pan-genome graphs are usually too large for these algorithms to be applied.

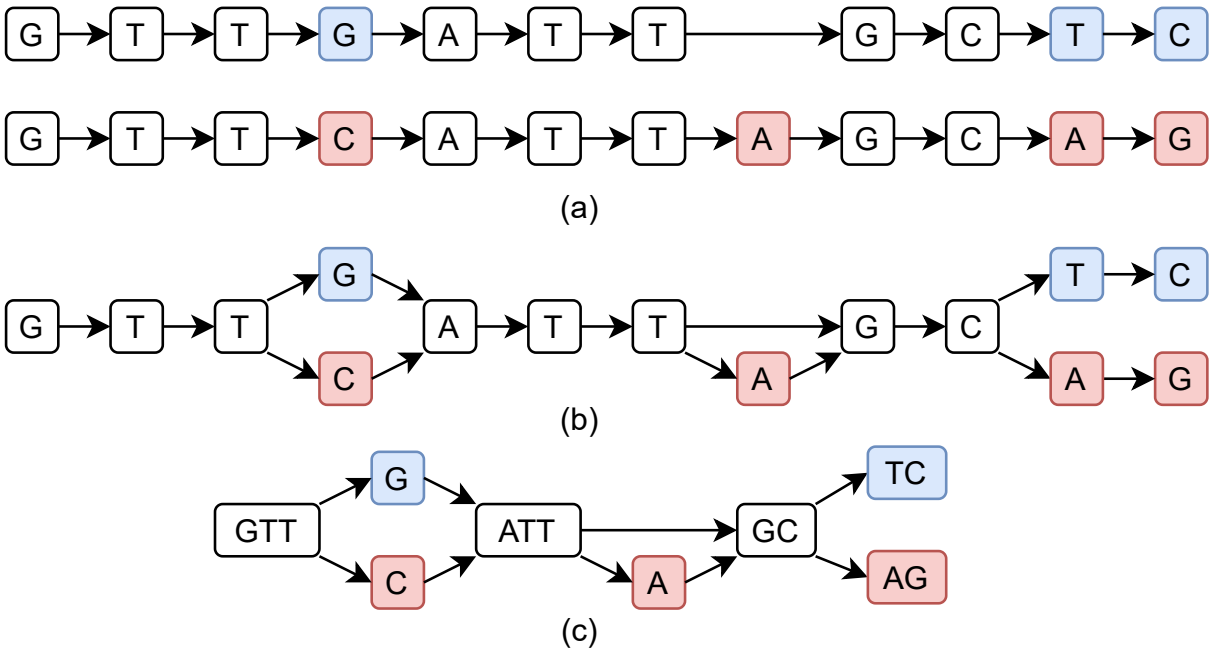

Figure 1.3: A simple example of a variation graph. (a) The two linear graphs on the top represent two sequences. (b) The two linear graphs can be merged into a variation graph. The merged graph is a DAG, and its width is 2. (c) In practice, the graph may have a compact representation by merging non-branching paths.

Therefore heuristics are widely used in practical applications of sequence-to-graph alignment. Techniques such as minimizers can extend to graphs by considering only the non-branching regions. There are many softwares to build variation graphs, `vg` [12] for an example. Figure 1.3 shows a variation graph built from two sequences. There are even more aligners on graphs for various purposes, such as `PaSGAL` [15], `GraphAligner` [27], and `Astarix` [13].

The co-linear chaining technique has also been proposed for sequence-to-graph alignment as in [17]. The algorithm is designed for DAGs and needs the graph to be of small width. Here the *width* of a DAG is defined as the minimum number of paths needed to cover all nodes in the graph. Although pan-genome graphs may have billions of nodes, the width is usually smaller. For example, a variation graph of chromosome 22 may have a width less than 10. Table 1.1 shows widths of such variation graphs on human chromosomes, which are built with whole genome sequencing (wgs) variants from Thousand Genomes project phase 3 release [5], which consists of variants from 2504 individuals. For this specific set of variants, the widths are all smaller than 10.

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

Table 1.1: Statistics of our variation graphs on human genomes built with vg. The number of nodes is from the compact representation where a non-branching path is merged as a single node. The number of base pairs is identical to the number of nodes in the single-letter-labelled graph. All graphs have only one connected component.

1.3 Sparse Dynamic Programming Framework

Although pan-genome graphs have a huge number of nodes, the width of the graph is usually small. For a linear graph representing a single sequence, the width is 1. For a variation graph built from k sequences, the width is at most k. In practice the width is even smaller, as the number of variants in a single site is usually smaller than the number of individuals in a population. As a result, for a graph with small width, some algorithms may achieve an efficiency that is close to the sequence cases.

The *sparse dynamic programming framework* of [21] utilises this small-width property. If the nodes of a DAG can be covered by k paths, the framework can extend some sequence algorithms to graphs with a similar time complexity multiplied by just k. The framework first computes a *minimum path cover* (MPC) of the graph, and builds an index of it. This preprocessing step takes $O(k|E|\log|V|)$ time for a DAG of $|V|$ nodes and $|E|$ edges and width k and is independent from the later dynamic programming algorithms for specific problems. This MPC index can be computed once and stored. The framework then maintains a data structure for each path in the MPC while processing nodes in topological

order. Hence, problems such as the longest increasing subsequences (LIS), longest common subsequences (LCS) on graphs and co-linear chaining without overlapping can be efficiently solved.

1.4 Our Contributions

In the sparse dynamic programming framework of [21], the graph is required to be a DAG. By slightly modifying the definition of co-linear chaining, we obtain an algorithm to accept general graphs while maintaining the same efficiency under the same assumption that the graph has a small width. Our algorithm solves the CLC problem on general graphs by reducing it to a slightly modified CLC problem on DAGs, and then apply the sparse dynamic programming framework. The new algorithm is discussed in Chapter 2.

We implement a version of the new algorithm on DAGs as a sequence-to-graph aligner `GraphChainer`. The code is based on an existing state-of-the-art aligner `GraphAligner` [27]. To reduce the alignment problem to co-linear chaining problem, we take short alignments from output of `GraphAligner` as anchors and connect the final chain to obtain an alignment. The implemented pipeline is demonstrated in Chapter 3. `GraphChainer` is freely available as an open source tool at https://github.com/algbio/GraphChainer.

We also evaluate `GraphChainer` in experiments and compare it with `GraphAligner`. Both simulated data and real sequencing data are used. On all data sets, `GraphChainer` has a significantly better alignment accuracy than `GraphAligner` assuming that smaller edit distance indicates a more accurate alignment. As for time and space usage, the co-linear chaining module takes only a small fraction of the overall aligning time, but the preprocessing by `GraphAligner` to obtain anchors is already slower than direct aligning with `GraphAligner`. The experiment design and results are included in Chapter 4.

Finally, we discuss the possible future development directions of our algorithm and implentation in Chapter 5.

2 Theoretical Results

We first focus on DAGs and discuss three definitions of co-linear chaining, where the overlapping between anchor paths are strictly forbidden, or only allowed for one-node overlapping, or allowed for all suffix-prefix overlapping.

Then we present a model for co-linear chaining problems in general and algorithms for the above variants and propose an efficient sparse dynamic programming algorithm for the one-node overlapping co-linear chaining. The algorithm is based on Algorithm 1 of [21] which solves non-overlapping co-linear chaining. The key component in both algorithms is the efficient computation of a minimum path cover and of an index on it, so a review of this part is also included.

Finally, we analyze the co-linear chaining problem on general graphs, possibly with cycles, and the correctness of applying the same algorithm on the one-node overlapping problem.

2.1 Definitions for Co-linear Chaining on Graphs

Given a directed graph $G = (V, E)$, where $V = \{1, 2, \ldots, |V|\}$ is the set of vertices, and $E \subset V^2$ is the set of edges, we use following notions:

- A *walk* in G is a sequence of nodes $P = p_1, p_2, \ldots, p_{|P|}$ where for all $i \in [1, |P|-1]$, $(p_i, p_{i+1}) \in E$ holds. In addition, we denote p_1 by $P.start$ and $p_{|P|}$ by $P.end$.
- We say that a node $u \in V$ can *reach* another node $v \in V$ if and only if there is a walk from u to v.
- A *path* in G is a walk in G which does not visit the same node twice.
- A *cycle* in G is a path $P = p_1, p_2, \ldots, p_{|P|}$ such that $(p_{|P|}, p_1) \in E$.
- If a directed graph has no cycles, then it is called a *directed acyclic graph (DAG)*. In a DAG, each walk is a path.
- An *anchor* is a pair $(P, [x \ldots y])$ of a path and an interval, where $P = p_1, p_2, \ldots, p_{|P|}$ is a path in G, and $[x \ldots y] = \{x, x+1, \ldots, y\}$ is an interval of integer indices. We

denote by $\mathcal{A}$ a given set of $N = |\mathcal{A}|$ *anchors*. We use $A.P$ to indicate the path of anchor A, and $A.x$, $A.y$ for the interval endpoints of A.

- An anchor A *precedes* another anchor B if that $A.x \leq B.x \wedge A.y \leq B.y$ and either $A.P.end$ can reach $B.P.start$ or there is a suffix-prefix overlap between $A.P$ and $B.P$, i.e. there exists $k > 0$ that $A.P.p_{|A.P|-k+i} = B.P.p_i$ holds for all $1 \leq i \leq k$. We denote this by $A \prec B$.
- A *chain* is defined as an ordered sequence of anchors $\mathcal{C} = s_1, s_2, \ldots, s_p$ such that the corresponding paths and intervals appears "from left to right", or more formally, for all $i \in [1 \ldots p-1], s_i \prec s_{i+1}$.

The *Graph Co-linear Chaining* (GCLC) problem introduced as Problem 3 in [21] asks to find a chain $\mathcal{C} = s_1 \ldots s_p$ of a subset of the anchors $\mathcal{A}$ maximizing the number of elements covered by the intervals in the chain, namely

$$Coverage(\mathcal{C}) = |\bigcup_{i=1}^{p} [s_i.x \ldots s_i.y]|.$$

Instead of a max-coverage chain, we can also use other target functions such as the size of the chain. More generally, as analyzed in [1], the problem of co-linear chaining function should have the form of

$$Target(\mathcal{C}) = \sum_{i=1}^{s} f(s_i) + \sum_{i=1}^{p-1} g(s_i, s_{i+1}),$$

where $f(\cdot)$ and $g(\cdot,\cdot)$ are some score functions. The algorithms searching for the optimal chain will be the same as long as the target function is a sum of values that only depends on one or two anchors locally, so that the optimal chain is always an extension of the optimal subchains. The above $Coverage(\mathcal{C})$ can be factored as $f(s) = 0$, $g(s_1, s_2) = s_2.y - max\{s_1.y, s_2.x - 1\}$ plus an initial score $s_1.y - s_1.x + 1$. For ease of discussion, we focus on the max-coverage chain in the following sections.

Chaining Without Overlaps With the precedence defined with strictly no overlapping anchors, we have the following problem:

Problem 1. *(CLC without Overlaps) Given a labeled DAG $G = (V, E)$, and N anchors $A_1, A_2, \ldots, A_N$ of the form $(P, [x \ldots y])$ where P is a path on G, and $x \leq y$ are non-negative integers, find an ordered subset $\mathcal{C} = s_1 \ldots s_p$ of the anchors such that:*

- *for all* $2 \leq i \leq p$*, it holds that* $s_{i-1}.y < s_i.y$*,* $s_{i-1}.P.end \neq s_i.P.start$*, and* $s_{i-1}.P.end$ *can reach* $s_i.P.start$*.*
- $\mathcal{C}$ *maximizes the coverage of intervals, that is,* $Coverage(\mathcal{C}) = |\bigcup_{i=1}^{p}[s_i.x \dots s_i.y]|$ *is maximized.*

Figure 2.1 shows an example of the co-linear chaining problem. For Problem 1, $A_1A_4A_5$ is a valid chain, while A_2A_3 is not valid because of overlapping paths, and A_2A_5 is not valid because their interval are not in order. A_1 and A_2 cannot be chained because $A_1.P.end$ cannot reach $A_2.P.start$.

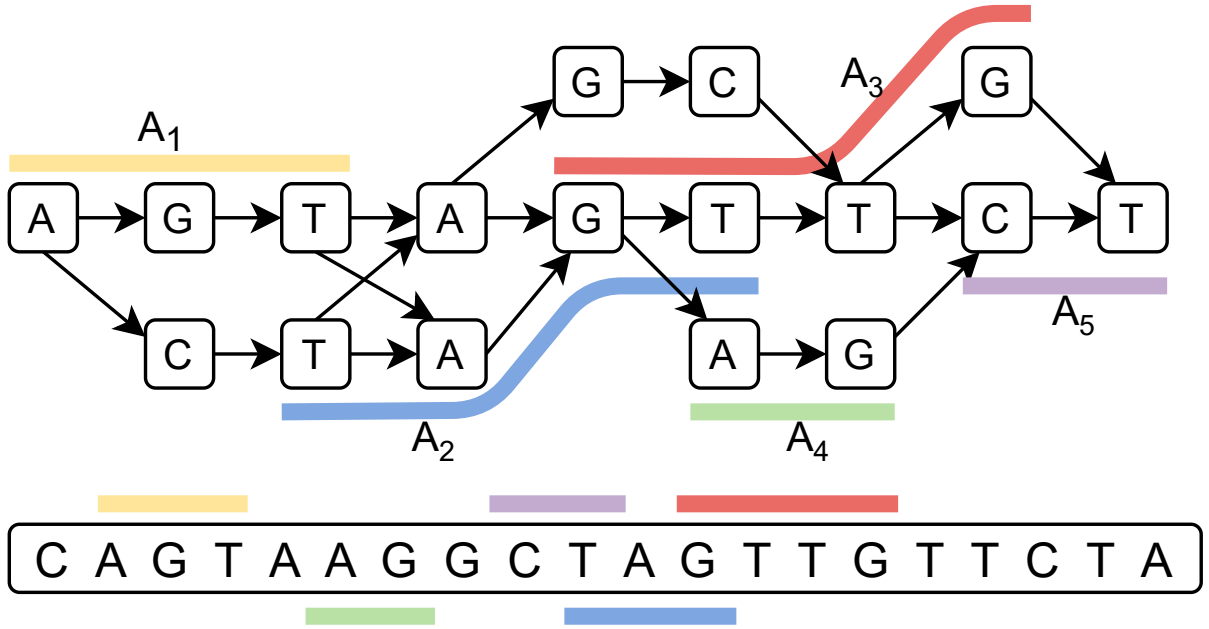

Figure 2.1: An example of the co-linear chaining problem. Each pair of a path and an interval of the same color indicates an anchor.

Chaining With Suffix-Prefix Overlaps Since in Problem 1 the intervals in the chain can overlap in the sequence, allowing a similar overlap of the paths of selected anchors might be preferred in some applications. Therefore, in addition to the above conditions on the paths, two paths appear "in order" if some suffix of the first path is a prefix of the second path. More formally, we have the following problem:

Problem 2. *(CLC With Suffix-Prefix Overlaps) Given a labeled DAG* $G = (V, E)$*, and* N *anchors* $A_1, A_2, \dots, A_N$ *of the form* $(P, [x \dots y])$ *where* P *is a path on* G*, and* $x \leq y$ *are non-negative integers, find an ordered subset* $\mathcal{C} = s_1 \dots s_p$ *such that:*

- *for all* $2 \leq i \leq p$*, it holds that* $s_{i-1}.y < s_i.y$ *and that either*
  - $s_{i-1}.P.end \neq s_i.P.start$ *and* $s_{i-1}.P.end$ *can reach* $s_i.P.start$*, or*
  - *there exists* $L \in [1 \dots \min\{|s_{i-1}.P|, |s_i.P|\}]$*, such that for all* $1 \leq j \leq L$*,* $s_{i-1}.P.p_{|s_{i-1}.P|-L+j} = s_i.P.p_j$*.*
- $\mathcal{C}$ *maximizes the coverage of intervals, that is,* $Coverage(\mathcal{C}) = |\bigcup_{i=1}^{p}[s_i.x \dots s_i.y]|$ *is maximized.*

This is a more general co-linear chaining definition compared to the non-overlapping one, and it can extend to the graphs with cycles easily. In Figure 2.1, under the definition of Problem 2, A_2A_3 is now a valid chain.

However, the suffix-prefix overlaps are more difficult to check for two given paths. In [21] it takes additional time $O(L \log^2 |V|)$ or $O(L + \#overlaps)$ where $L = \sum_i |A_i.P|$ is sum of anchor path lengths and $\#overlaps$ is the number of overlaps between the input paths. So we give another definition of co-linear chaining that is general enough for the later extension on graphs with cycles, but still has an efficient solution.

Chaining With One-node Overlaps The most basic overlapping case is when the two paths share exactly one node. This is equivalent to allowing node u to reach u itself.

Problem 3. *(CLC with One-node Overlaps) Given a labeled DAG* $G = (V, E)$*, and* N *anchors* $A_1, A_2, \dots, A_N$ *of the form* $(P, [x \dots y])$ *where* P *is a path on* G*, and* $x \leq y$ *are non-negative integers, find an ordered subset* $\mathcal{C} = s_1 \dots s_p$ *such that:*

- *for all* $2 \leq i \leq p$*, it holds that* $s_{i-1}.y < s_i.y$ *that either* $s_{i-1}.P.end = s_i.P.start$ *or* $s_{i-1}.P.end$ *can reach* $s_i.P.start$*.*
- S *maximizes the coverage of intervals, that is,* $Coverage(\mathcal{C}) = |\bigcup_{i=1}^{p}[s_i.x \dots s_i.y]|$ *is maximized.*

2.2 Algorithms

We first review the algorithm to compute the minimum path cover index from [21] which solves our Problem 1 and 2 and then propose our algorithm for solving Problem 3.

2.2.1 Minimum Path Cover

A *path cover* of a graph $G = (V, E)$ is a set of paths that each node $u \in V$ is included in at least one path in the set. A *minimum path cover* (MPC) is a path cover with the smallest number of paths. The *width* of a graph is the size of its minimum path cover.

To solve the minimum path cover on DAGs, we can reduce the problem to a lower-bounded network flow problem [25]. First, we construct a new graph $G' = (V', E')$. A source S and a sink T are added to V'. We split each node $u \in V$ into two nodes u_{in} and u_{out} that are added to V'. So $V' = \{S, T\} \cup \{u_{in}, u_{out} | u \in V\}$ where S and T are two new nodes. For each edge $(u, v) \in E$ we add edge (u_{out}, v_{in}) to E'. In addition, we add (u_{in}, u_{out}) and $(S, u_{in}), (u_{out}, T)$ for all $u \in V$. All edges have infinite capacity. Figure 2.2 shows an example DAG with 5 nodes.

There is a mapping between a set of paths on G and a flow on G'. On the one hand, each path in G is mapped to a path in G' from S to T following the same nodes. In the other direction, each flow can be decomposed into a set of several paths from S to T which each has a unique path in G with the same nodes.

A flow that is at least one for all edges (u_{in}, u_{out}) clearly maps to a path cover in G. By requiring a lower bound of 1 on these edges, a minimum flow on G' gives a minimum path cover on G. This is a well-studied problem that can be reduced to the basic max flow problem by shrinking from a satisfying flow. Such flow always exists as there is always path covers on G with $|V|$ single-node paths. In addition, the time complexity is proportional to the amount of flow shrinks. We can first compute a greedy path cover by iteratively selecting a path that covers the most uncovered nodes. If the width of G is k, it was proved that the greedy path cover has $O(k \log |V|)$ paths [21]. Therefore, the entire shrinking procedure takes $O((k \log |V| - k)|E|) = O(k|E| \log |V|)$ time.

Given a path cover $\mathcal{P}$ of size k, where $\mathcal{P} = P_1, P_2, \ldots, P_k$, the set of nodes that can reach a given node u is the union of nodes on each path that can reach u and on each path such nodes form a prefix of the path. Therefore, as in [21] we can efficiently find this set by precomputing the array $last2reach[i, u]$, defined as the last node on path P_i other than u that can reach u, or $\emptyset$ if no such node exists. Further, we compute a reversed set $forward[u]$ for all $u \in V$ such that $(v, i) \in forward[u]$ if and only if $last2reach[i, v] = u$.

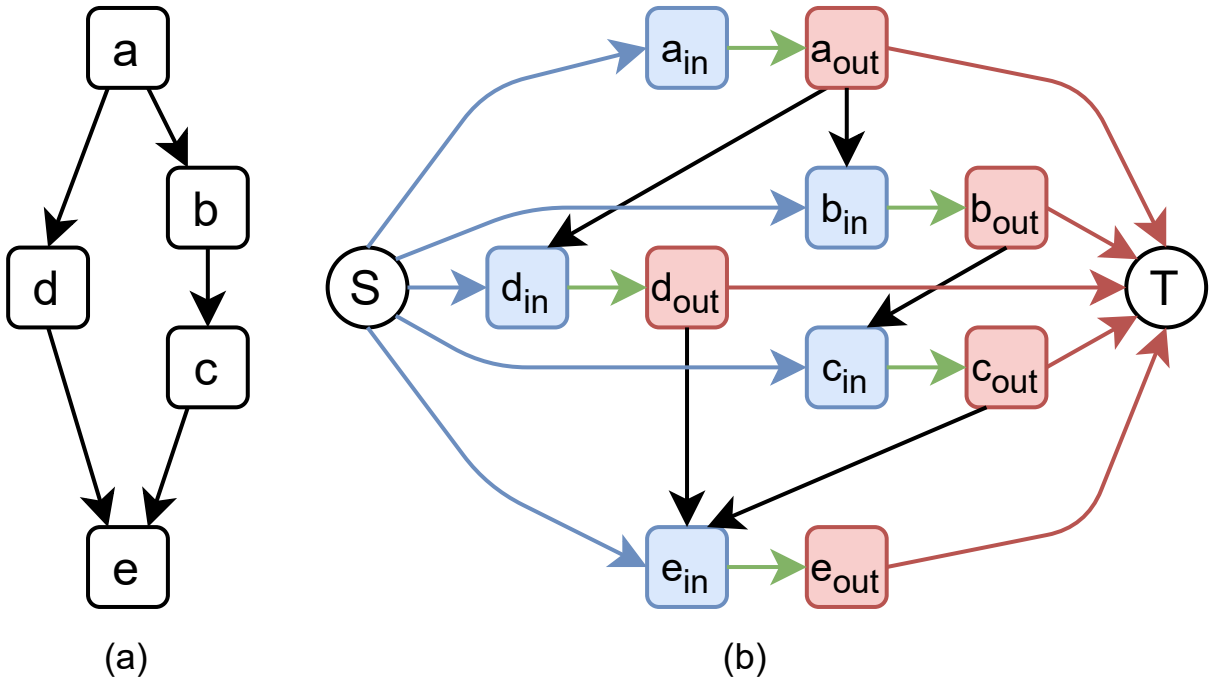

Figure 2.2: A example of the reduction from G to G'. (a) The original DAG of 5 nodes. For clarity we use $V = \{a, b, c, d, e\}$ instead of integers. (b) The new graph G'. Green edges have a requirement on the flow of a lower bound of 1, and all other edges have no limits.

2.2.2 Sparse Co-linear Chaining

We propose a new algorithm for solving Problem 3 on a DAG of small width. It is similar to Algorithm 1 in [21]. Both algorithms require a path cover index built on the DAG, and process the nodes in topological order. While Algorithm 1 solves Problem 1 that requires strictly non-overlapping paths, the new algorithm solves Problem 3 by explicitly adding the one-node overlapping updates in the dynamic programming.

The algorithm computes an array $C[1 \dots N]$ so that $C[i]$ is the maximum coverage of chains that has the anchor A_i as the last anchor and consists of only anchors A_j that $j < i$. The previous algorithm, for each node in topological order, first updates the index structure on each path with all the anchors whose paths end at the current node, then updates all the anchors whose paths start from a node that is "one step" from the current node by following the *forward* links from the path cover index.

The index structure needs to maintain a set in the form of $D = \{(x_i, c_i) \mid 1 \leq i \leq |D|\}$ and to support two operations:

- $D.update(x, c)$ adds a new pair (x, c) to the set D.
- $D.RMaxQ(l, r)$ returns the maximum c_i of all pairs in D with x_i in the range $[l, r]$, that is, $D.RMaxQ(l, r) := max\{c_i \mid (x_i, c_i) \in D \wedge l \leq x_i \leq r\}$.

There are many efficient data structures for this purpose. For example, it is sufficient to use a *balanced binary search tree* with a worst-case time complexity $O(\log |D|)$ for each operation, as shown in Chapter 3 of [20].

The one-node overlapping has tricky cases where the above dynamic programming order no longer works. Several anchors with a single-node path on the same nodes, can update each other's $C[]$ array values in an interweaving way. We can sort these anchors first by their sequence interval ending positions, and process one anchor at a time by first updating its $C[i]$ and then inserting this value into the data structure. In addition, we include the anchors with a path starting or ending at the current node as well. We show the pseudo-code of this algorithm in Algorithm 1.

In the new algorithm, we still process nodes one by one in topological order. Initially we set $C[j] := A[j].y - A[j].x + 1$ which is the $Coverage(A[j])$ for a single-anchor chain. Also we initialize two index structures T_i and I_i for each path P_i in the path cover. For each node v, we first consider the one-node overlapping cases. We find all anchors A_i that has either $A_i.P.start = v$ or $A_i.P.end = v$ and sort them in the order of increasing $A_i.y$. Two temporary index structures $TmpT$ and $TmpI$ are initialized. For each of these anchors, if $A_i.P.start = v$ then $C[i]$ is updated with values in $TmpT$ and $TmpI$, and if $A_i.P.end = v$ we add its value $C[i]$ to $TmpT$ and $TmpI$. Note that for an anchor A_i, it is possible that $A_i.P.start = A_i.P.end = v$ so that $C[i]$ is immediately added to the index structures after it is updated. After all of these anchors processed, we follow the same steps in the previous algorithm: add the value $C[i]$ of each anchor A_i with $A_i.P.end = v$ to all index structures T_i and I_i if path P_i contains v, then compute $C[i]$ for all anchors A_i that $A_i.P.start$ occurs in the forward links of v.

Algorithm 1: Co-linear Chaining on DAG using path cover

Input: DAG $G = (V, E)$, a path cover $P = P_1, \ldots, P_k$ of G, Anchors $\mathcal{A} = \{A[1], \ldots, A[N]\}$

Output: index j giving max $C[j]$

for $i \leftarrow 1$ **to** l **do**
  T_i.initialize();
  I_i.initialize();
for $j \leftarrow 1$ **to** N **do**
  $C[j] = A[j].y - A[j].x + 1$;
  start$[A[j].P.start]$.push(j);
  end$[A[j].P.end]$.push(j);
for $v \in V$ *in topological order* **do**
  TmpT.initialize(); TmpI.initialize();
  for $j \in$ *end[v]*$\cup$*start[v] in the order of increasing* y **do**
    if $A[j].P.start == v$ **then**
      $C^a[j] \leftarrow max(C^a[j], A[j].y - A[j].x + 1 + TmpT.RMaxQ(0, A[j].x - 1))$;
      $C^b[j] \leftarrow max(C^b[j], A[j].y + TmpI.RMaxQ(A[j].x, A[j].y))$;
      $C[j] \leftarrow max(C[j], C^a[j], C^b[j])$;
    if $A[j].P.end == v$ **then**
      $TmpT.update(A[j].y, C[j])$;
      $TmpI.update(A[j].y, C[j] - A[j].y)$;
  for $j \in$ *end[v]* **do**
    for $i \in$ *paths[v]* **do**
      $T_i.update(A[j].y, C[j])$;
      $I_i.update(A[j].y, C[j] - A[j].y)$;
  for $(u, i) \in$ *forward[v]* **do**
    for $j \in$ *start[u]* **do**
      $C^a[j] \leftarrow max(C^a[j], A[j].y - A[j].x + 1 + T_i.RMaxQ(0, A[j].x - 1))$;
      $C^b[j] \leftarrow max(C^b[j], A[j].y + I_i.RMaxQ(A[j].x, A[j].y))$;
      $C[j] \leftarrow max(C[j], C^a[j], C^b[j])$;
return $argmax_j C[j]$

2.2.3 Proof of Correctness

Assume that the anchors $A[1] \dots A[n]$ are sorted so that we assume the endpoints of $A[1].P, A[2].P, \dots, A[N].P$ are in topological order, breaking ties by smaller $A[i].y$ first. Therefore any valid chain will be a subsequence of $A[1] \dots A[N]$ in the same order. We denote by $C[j]$ the maximum coverage using the anchor $A[j]$ as the last anchor in the chain and using only a subset of anchors $A[1] \dots A[j]$. There are 3 cases for computing $C[j]$.

(a) There is more than one anchor in the chain, and the intervals of the last two anchors have no overlap.

(b) There is more than one anchor in the chain, and the intervals of the last two anchors have non-empty overlap.

(c) $A[j]$ is the only anchor in the chain.

So the recursion for $C[j]$ is:

$$C[j] = max \begin{cases} C[j'] + A[j].y - A[j].x + 1 & \forall 1 \le j' < j, A[j'].y < A[j].x \wedge A[j'] \prec A[j] \quad (a) \\ C[j'] + A[j].y - A[j'].y & \forall 1 \le j' < j, A[j].x \le A[j'].y \wedge A[j'] \prec A[j] \quad (b) \\ A[j].y - A[j].x + 1 & (c) \end{cases}$$

Note that in case (b) if $A[j].x < A[j'].x$, i.e. the interval of $A[j']$ contained in the interval of $A[j]$, the coverage is computed incorrectly, but it is not better than the chain that has no $A[j']$ and thus fixed by the another chain that is computed correctly.

Denote the intermediate value for case (a) by $C^a[j]$ and for case (b) by $C^b[j]$. By our definition, an anchor $A[j']$ can contribute to $C[j]$ only if $A[j'].y < A[j].y$ and $A[j'].P.end$ can reach $A[j].P.start$. Denote by $R^-(u)$ the set of nodes that can reach node u, including node u itself. Then $A[j'] \prec A[j]$ if and only if $A[j'].y < A[j].y \wedge A[j'].P.end \in R^-(A[j].P.start)$

Given a path cover of size K: $P_1, \dots P_K$, and the array $last2reach[u, i]$ and $forward[i]$, define $R_i^-(u)$ as the set of nodes on P_i that can reach u, including u itself. For any node u, we have that

$$R^-(u) = \bigcup_{1 \le i \le K} R_i^-(last2reach[u, i]) \cup \{u\}.$$

Therefore, $C^a[j]$ can also be computed by combining results from all the paths, and the additional updates from node $A[j].P.start$.

Also define

$$C_i^a[j] = max\{\, C[j'] + A[j].y - A[j'].y \mid \quad A[j'].P.end \in R_i^-(last2reach[A[j].P.start, i]) \wedge A[j'].y < A[j].x \,\}$$

$$C_u^a[j] = \max\{\, C[j'] + A[j].y - A[j'].y \mid \quad A[j'].P.end = A[j].P.start \wedge A[j'].y < A[j].x \,\}$$

where $C_i^a[j]$ is the updates from path P_i and anchors with a path that are not overlapping with anchor $A[j]$, and $C_u^a[j]$ is the update from anchors that has one-node overlaps with $A[j]$. Then

$$C^a[j] = \max\{\max_{1 \le i \le K} C_i^a[j], C_u^a[j]\}.$$

And similarly we can define $C_i^b[j]$ and $C_u^b[j]$.

We prove the correctness by induction. Assume that after the first M nodes $Visited = \{v[1] \dots v[M]\}$ in topological order are processed, $C[j]$ should be computed with the correct values if $A[j].P.start \in Visited$, and the array $C^a[j]$ and $C^b[j]$ holds that

$$C^a[j] = \max\{C_i^a[j] \mid last2reach[A[j].P.start, i] \in Visited\}$$

$$C^b[j] = \max\{C_i^b[j] \mid last2reach[A[j].P.start, i] \in Visited\}$$

and T_i contains $\{(A[j].y, C[j]) \mid A[j].P.end \in Visited \cap P_i\}$, I_i contains $\{(A[j].y, C[j] - A[j].y) \mid A[j].P.end \in Visited \cap P_i\}$.

This is trivially true for $M = 0$ when $Visited = \emptyset$. Suppose that the assumption is true for $M \ge 0$, and now $v[M+1] = u$. For all paths i, $last2reach[u, i]$ should be visited if exists, so $C^a[j] = \max\{C_i^a[j]\}$.

First we prove that $C[j]$ are computed correctly if $A[j].P.start = u$. For all $1 \le i \le K$ we have $last2reach[u, i] \in Visited$ so $C^a[j]$ holds the $\max\{C_i^a[j] \mid 1 \le i \le K\}$. Suppose the anchors in $end[u] \cup start[u]$ are sorted as $j_1, \dots j_W$, we have $C_u^a[j_k] = \max\{C[j_{k'}] + A[j].y - A[j].x + 1 \mid 1 \le k' < k \wedge A[j_{k'}].y < A[j_k].x\}$ for $k \in [1 \dots W]$. In the first inner loop, when the first k anchors are processed, their values are added to the temporary search trees, and $C_u^a[j_k]$ can be computed with range maximum query, then $C^a[j_k]$ is computed and can be added to the trees. The proof for $C^b[j]$ is similar.

Then the search trees T_i and I_i are updated as stated in the assumption, to include the values of anchor $A[j]$ if $A[j].P.end = u$ and P_i contains node u.

Finally, we prove that the related $C_i^a[j]$ are computed as in the assumption. For an anchor $A[j]$, the value $C^a[j]$ remains the same if $u \neq last2reach[A[j].P.start, i]$ for all

$i \in [1 \dots K]$. Otherwise if $u = last2reach[A[j].P.start, i]$ for some anchor $A[j]$ and P_i, we have exactly all the anchors with end points from $R_i^-(u)$ added to the search tree T_i, so a range maximum query on range $[0, A[j].x - 1]$ gives $C_i^a[j]$. By following the forward links, the array $C^a[j]$ for all anchors j where $u \neq last2reach[A[j].P.start, i]$ for some P_i, is updated with $C_i^a[j]$.

Therefore, the assumption holds for $M+1$. After the main loop in the algorithm, we have $Visited = V$ and $C[j]$ has the right values for all anchors. $\max C[j]$ gives the maximum coverage of all the chains.

2.2.4 Time Complexity

The minimium path cover can be computed as in [21] with time complexity $O(k|E|\log|V|)$, where k is width of graph. In the dynamic programming part, each anchor will insert at most one point in each data structure. The peak size of these data structures is therefore N, so each insertion or range maximum query can be done in $O(\log N)$ with balanced search tree. Each $C[i]$ is updated at most once from each data structure, so the total number of range maximum queries is at most $(2k+2)N$. Therefore the algorithm takes $O(k|E|\log|V|+kN\log N)$ time which is the same as in the non-overlappint case in [21].

2.3 Co-linear Chaining on Graphs with Cycles

With a more general definition of precedence, we can define the co-linear chaining problems similarly on graphs with cycles. The motivation of co-linear chaining is to recover a long match by chaining a set of short matches in order. Two anchors can be chained if there is a long match containing both of the anchors and they appear in it in order. For the DAG case, a match is between a path and a sequence. When the graph has cycles, walks and paths can be different, since a walk might contain some node multiple times, leading to a confusing definition of "overlapping". For example, given two different nodes $u, v \in V$ such that u can reach v, if the graph is a DAG, this implies that every path ending at u has no common nodes with any path starting at v and that the paths ending at u are "before" paths starting at v. But when the graph is just a cycle of all nodes, v can also reach u, and any path appears "before" another path including itself. So we look for a relaxation of the precedence relationship.

2.3.1 Relaxation of Precedence

In the relation $A.P \prec B.P$ we remove the condition that "$A.P$ and $B.P$ have no common nodes" and simply require that "$A.P.end = B.P.start$ or $A.P.end$ can reach $B.P.start$". The case when $A.P$ and $B.P$ have a suffix-prefix overlap is similar to the case when they have no common nodes. Suppose $A.P.end$ can reach $B.P.start$, there is a walk that contains $A.P$ and $B.P$ in order as two disjoint parts. With this new definition for $A.P \prec B.P$ we obtain the co-linear chaining problem on general graphs:

Problem 4. *(CLC with One-node Overlaps On General Graphs) Given a labeled graph $G = (V, E)$, and N anchors of the form $(P, [x \dots y])$ where P is a walk on G, and $x \leq y$ are non-negative integers, find an ordered subset $\mathcal{C} = s_1 \dots s_p$ such that:*

- *for all $2 \leq i \leq p$, it holds that $s_{i-1}.y < s_i.y$ and $s_{i-1}.P \prec s_i.P$, that $s_{i-1}.P.end = s_i.P.start$ or $s_{i-1}.P.end$ can reach $s_i.P.start$.*
- *S maximizes the coverage, that is, $Coverage(\mathcal{C}) = |\bigcup_{i=1}^{p}[s_i.x \dots s_i.y]|$ is maximized.*

2.3.2 Reduction to DAGs

A straightforward solution for Problem 4 is to consider a reduction to Problem 3 on DAG. Here we take the *condensation* of a graph, which is a DAG, and discuss how to apply the algorithm for Problem 4.

Consider the *condensation* or *strongly connected components graph* $G_c = (V_c, E_c)$ of a graph G possibly with cycles. A *strongly connected component* is a set of nodes $S = \{v_1, v_2, \dots, v_{|S|}\}$ such that any $u \in S$ can reach any $v \in S$, and is maximal with this property. In particular, a set of just one node is a strongly connected component, and so is a cycle. Each node u is in the same component with all the nodes that are reachable from u and can reach u. This component is represented as a single node in V_c. Let the mapping $S : V \to V_c$ mas the node u from G to its component node in G_c. Furthermore, for a walk $P = p_1, p_2, \dots, p_{|P|}$, let $S(P)$ be the walk $S(P) = S(p_1), S(p_2), \dots, S(p_{|P|})$ and remove adjacent duplicate nodes. $S(P)$ is a walk on the DAG G_c so it is a path with no duplicate nodes.

G_c will be a DAG with a width smaller or equal to the width of G. This is because any path cover $\mathcal{P} = P_1, P_2, \dots, P_k$ of G gives a path cover $S(\mathcal{P}) = S(P_1), S(P_2), \dots, S(P_k)$ on G_c of size k.

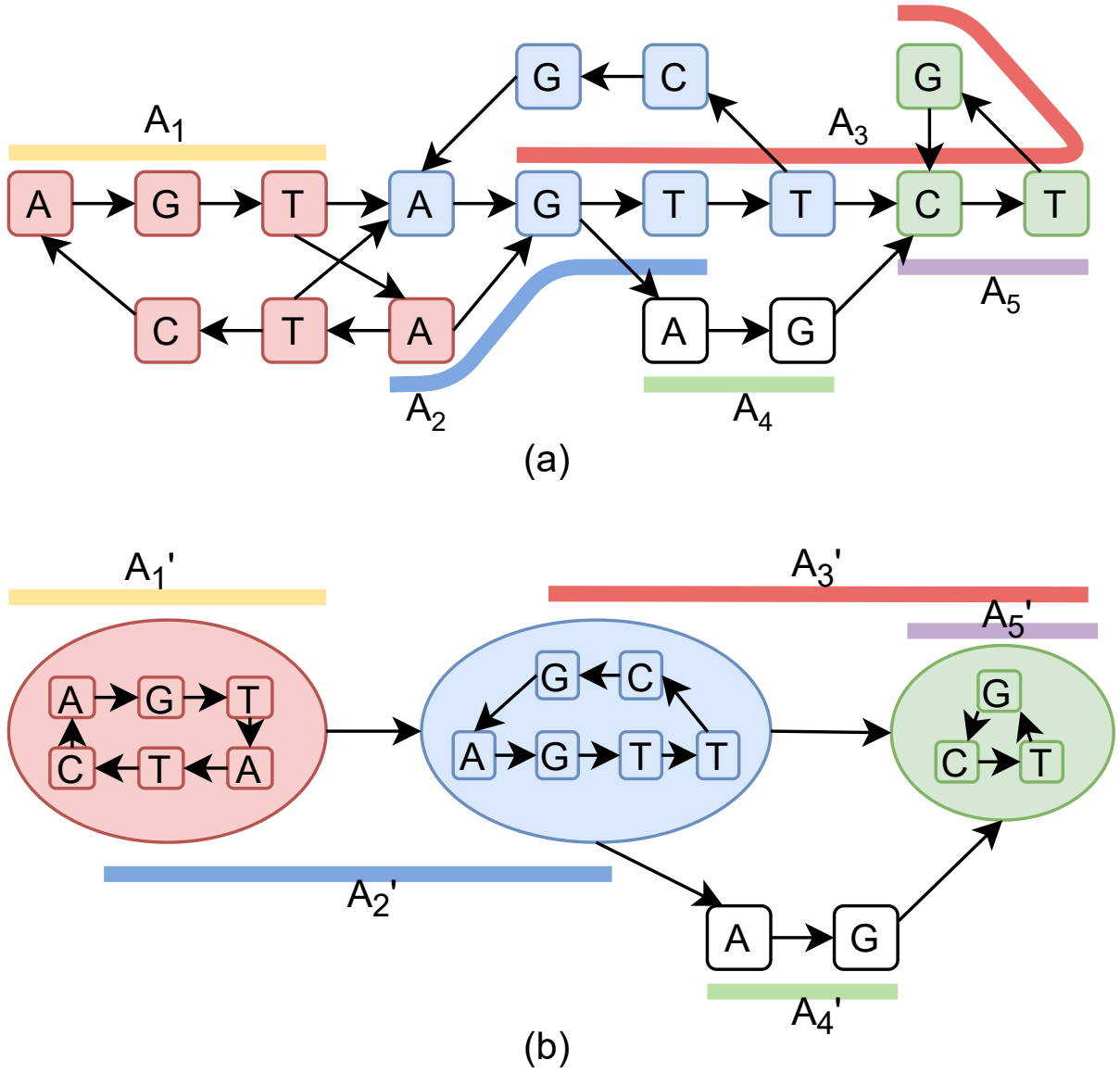

Figure 2.3: An example of the reduction to DAGs. (a) The original graph and 5 anchors. Only the path of each anchor is shown, since the intervals are not affected. (b) The reduced DAG and mapped anchors.

Figure 2.3 shows an example reduction from a graph with cycles to a DAG. There are 3 strongly connected components with more than one node and two single-node components. After reduction, many anchors now have one-node overlaps with each other (e.g. A'_1 and A'_2, A'_3 and A'_5). In addition, anchor A'_1 and A'_5 now have only one node in their paths. A_1A_2 is a valid chain without overlaps, while $A'_1A'_2$ has an one-node overlap.

To apply the sparse dynamic programming algorithm for solving non-overlapping co-linear chaining on this DAG G_c, we need to modify it for anchors that overlap within the same strongly connected component. For anchors with a path completely included in one com-

ponent, any two of them can be chained if their intervals allow. For two anchors that have a suffix-prefix overlap of a single node, they can also be chained.

Given N anchors $\mathcal{A} = \{A[1], \ldots, A[N]\}$, we map them to $\mathcal{A}' = \{A'[1], \ldots, A'[N]\}$ where $A'[i].P = S(A[i].P)$. This gives an instance of Problem 3 with $\mathcal{A}'$ and G_c.

Now we need to prove that the maximum-coverage chain $\mathcal{C}' = A'[i_1], A'[i_2], \ldots, A'[i_{|\mathcal{C}'|}]$ of $\mathcal{A}'$ on G_c gives also a maximum-coverage chain $\mathcal{C} = A[i_1], A[i_2], \ldots, A[i_{|\mathcal{C}|}]$ for G. To do this, we prove that $\mathcal{C}$ is a chain on G if and only if $\mathcal{C}'$ is a chain on G_c.

For two anchors $A, B \in \mathcal{A}$, we prove that A can be chained before B if and only if A' can be chained before B'. The condition that $A.y < B.y$ holds since $A'.y = A.y$ and $B'.y = B.y$. If $A.P.end = B.P.start$ or $A.P.end$ can reach $A.P.start$, with the above mapping S, we have that $A'.P.end = S(A.P.end)$ and $B'.P.start = S(B.P.start)$ so either $A'.P.end = B'.P.start$ ($A.P.end$ and $B.P.start$ are in the same component) or $A'.P.end$ can reach $A'.P.start$ ($A.P.end$ and $B.P.start$ are not in the same component). In the other direction, if $A'.P.end = B'.P.start$ or $A'.P.end$ can reach $A'.P.start$, there is a walk W in G from some node u to node v that $S(u) = S(A.P.end)$ and $S(v) = S(B.P.start)$. So there is a walk L from $A.P.end$ to u, and a walk R from $B.P.start$ to v. The concatenation of L, W, R is a walk from $A.P.end$ to $B.P.start$ so A can be chained before B. Therefore there is a bijection between the chains on G with $\mathcal{A}$ and on G_c with $\mathcal{A}'$, and the maximum-coverage chain on G_c gives the optimal chain on G.

Note that the same proof does not work if one-node overlapping is not allowed. A pair of anchors that can be chained without common nodes might have a one-node overlap in G_c if the nodes are in the same strongly component in G. So there is no such bijection between chains on G and on G_c unless we allow one-node overlapping.

3 Implementation into a Pan-Genomic Graph Aligner

We present here how the co-linear chaining algorithms are implemented efficiently on DAGs and how they are applied in sequence-to-graph alignment. The pipeline is described with technical details, in the hope that one can reproduce the experimental results from Chapter 4 reliably, and that as a software in practice, the internal mechanism is as transparent as possible. Therefore, one can use our implementation directly or modify the code easily whenever necessary.

Although the algorithm can be applied on graphs with cycles, we implemented it only on DAGs. One reason is that the reduction from general graphs to DAGs is relatively independent from the co-linear chaining problem on DAGs. Another reason is that all the variation graphs created by `vg` in our experiments are DAGs. One can easily modify the implementation to work on graphs with cycles by adding the reduction procedures.

First, in Section 3.1 we describe the application scenario where an erroneous long read is being aligned to a pre-built pan-genome reference graph. The pipeline is demonstrated briefly. Then in Section 3.2 the necessary steps to prepare the input graph are listed, together with the pre-processing needed to extract anchors from the long read, and the procedures to build a minimum path cover index with an efficient maxflow solver. In Section 3.3 we show the implementation details of the algorithm solving the one-node overlapping co-linear chaining problem described in Chapter 2 Problem 3, and review a method to support dynamic range maximum queries on a balanced binary search tree. Finally, in Section 3.4 we demonstrate the method to convert a chain of anchors to an alignment between the input read and a matching path on the graph with the help of `edlib` [31].

3.1 Overview

One of the most common applications in pan-genome graphs is aligning a read sequence against a reference graph. The task is to find a path of the graph, such that its concatenation of node labels has a small edit distance to the input read, or to its reverse

complement. Both the sequence and the graph may have ambiguous characters such as `N` that can match multiple characters from `ACGT`. In particular, a long read, such as the ones produced from third-generation sequencing, is difficult to align due to its length and high error rate. For example, one long read may have 10k to 40k base pairs, which can exceed 100 times the length of short reads. In addition, the previous algorithms and techniques for short reads might not be directly applicable to long reads because of the high error rate. The error rate of these long reads is usually around 15%, which is much higher than that of short reads (about 0.1% for Illumina). As a result, it means that the edit distance between the long read and the original ground truth sequence can easily reach 6000. Based on observation, these errors (insertion, deletion and mismatch) in a read are not uniformly distributed, but appear in a pattern of small clusters. So techniques such as seeding with the minimizer scheme may still work well when some continuous parts in the long reads have a relatively small number of errors. One can extend the seed-and-grow strategy from sequence-to-sequence alignment to obtain a long read sequence-to-graph aligner, such as GraphAligner [27].

However, there are two drawbacks to the direct adoption of extension algorithms on long reads. One is that the seeds may be clustered in several regions of the graph and are separated in distance. Extending from one seed through the erroneous zone to reach the next relatively accurate region is hard. Hence, for a long read, such aligner may find several short alignments covering different parts of the read, but not a long alignment of the entire read. On the other hand, a short part may have many falsely aligned fragments that are not interesting for the alignment of the long read. If the sequence is randomly generated, there is always a small chance to have duplication paths on the graph with the same labels accidentally. Furthermore, in genome sequences, some chromosomes share some sequences with other chromosomes.

So, among the many short alignments, one can connect a subset of them to form a long alignment. This is where co-linear chaining can be applied.

Pipeline The co-linear chaining aligner pipeline is shown in Figure 3.1.

Our idea is to split one long read into many short ones, covering the entire read, and we search for alignments of these short reads using other tools. Then we take all these alignments as anchors and compute the max coverage chain. This chain is a sequence of matching pairs of a path on the graph and an interval in the long read, and by our definition of co-linear chaining, we can connect these paths to form a long path. The

intervals are also merged as one. To connect one path to another, we perform a breadth-first search (BFS) from the ending node of the first path to reach the starting node of the second path. This connected path from BFS is not optimal as an alignment, but already yields good alignments for the long read.

At last, we search an alignment between the long path and the long read and map it back to a sequence-to-graph alignment. This sequence-to-sequence edit distance alignment task is well-studied and has many efficient tools. We use `edlib` [31] with the global edit distance setting, and pass an additional matching matrix to allow ambiguous characters.

3.2 Pre-processing for Co-linear Chaining

The input for a sequence-to-graph aligner is a graph and a set of read sequences. For the co-linear chaining with sparse dynamic programming, the inputs are a pan-genome graph, a set of anchors and a minimum path cover index. The MPC index only needs to be built once for the same graph, and can be stored on disk together with the graph file. To use GraphAligner correctly for anchor preparation, the input graph has to satisfy a few conditions.

Pan-Genome Graph Although the graph is given as input to the aligner, we require that the graph is a directed acyclic graph. This is naturally satisfied for variation graphs produced by `vg` from a reference sequence plus a set of variants in Variant Call Format (`.vcf`). As shown in Chapter 2, by computing the condensation of a graph with cycles, the algorithm can solve the co-linear chaining problem on general graphs. However, it is rare to find such graphs that are large enough that co-linear chaining on general graphs is necessary in the context of pan-genome graphs.

Furthermore, to let GraphAligner work properly, the nodes in non-branching regions should be merged into a single node with a label obtained by concatenating all node labels. This is because GraphAligner searches for seeds that are completely contained in one node label only. Tools such as `vg construct` might split long node labels into 32-base-pair chunks and cause very few seeds reported by GraphAligner. Once the graph is loaded inside GraphAligner, the long node labels are split into shorter nodes in a similar manner. In our implementation, we consider the split node that might have a node label longer than 1 base pair as if they are just nodes described in Chapter 2. As a result, a one-node overlap can be in fact a short suffix-prefix overlap. This fine for our applications,

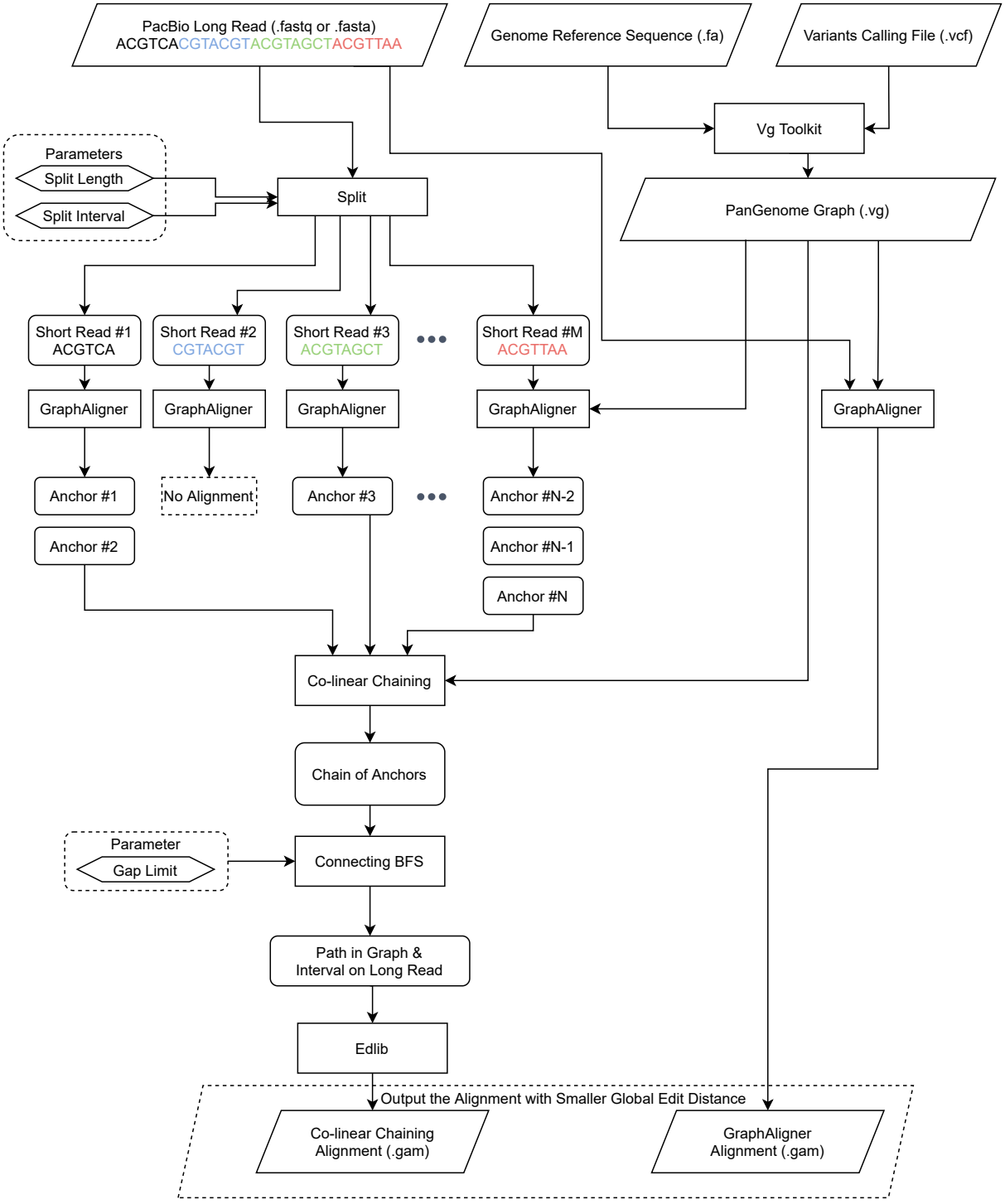

Figure 3.1: The pipeline of our long read aligner with co-linear chaining. GraphAligner, vg and edlib are used as tools. Although GraphAligner is drawn as an external tool in the chart, in the final software the pipeline is intensively integrated with GraphAligner.

since the default parameter 32 is much smaller compared to the read lengths and most nodes have shorter labels.

The graph can contain ambiguous characters such as `N`. When such are characters present, GraphAligner will ignore them when searching for seeds as if the node labels are split before and after these ambiguous ones.

Anchors To obtain anchors, we split and align the long read with GraphAligner. The input read sequence is partitioned into short reads of L base pairs each. Then we run GraphAligner for each short reads separately. For each alignment reported for each short read, we extract the path of this alignment, and the original interval in the long read of this split, to form an anchor. Here length L of each short read is defined as a parameter `colinear-split-len`.

This method might miss some good short alignments crossing the splitting points between short reads. So we also give a parameter `colinear-split-gap` or S to indicate the step size, which is the number of base pairs between the starting positions of adjacent short reads. By default $S = L$ and the short reads forms 1x coverage of the long read. When S is smaller, more anchors may be reported since more short reads are created. This may drastically increase the pre-processing time for computing anchors, but we only observed a tiny improvement in final alignment quality.

MPC Index Here we implement the algorithm to compute a minimum path cover as described in [21]. The graph is not necessarily connected. When the graph has m connected components $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \ldots, G_m = (V_m, E_m)$, each with widths $k_1, k_2, \ldots, k_m$, the width of the entire graph is $\sum_{1 \leq i \leq m} k_i$, which can be relatively huge. Since both the co-linear chaining and the aligning are independent within each connected component, we can build the MPC index for each component separately, and partition the anchors based on which component the anchor path belongs to. The time spent is the sum of time for building the MPC index for each component separately, i.e. $O(\sum_{1 \leq i \leq m} k_i |E_i| \log |V_i|)$. This essentially reduces the complexity from $O(\sum_{1 \leq i \leq m} k_i |E| \log |V|)$ to $O(\max_{1 \leq i \leq m} k_i |E| \log |V|)$ because

$$O(\sum_{1 \leq i \leq m} k_i |E_i| \log |V_i|) \leq O((\max_{1 \leq i \leq m} k_i \sum_{1 \leq i \leq m} |E_i| \log |V|) = O(\max_{1 \leq i \leq m} k_i |E| \log |V|).$$

In addition, since the sequence can be matched in the reverse complemented form, the graph is first copied once to build a reverse complement graph, where each edge is re-

versed, and node labels are also changed. Merging this copy and the original graph will double the width. With the above optimization with connected components, this issue is automatically addressed.

Computing Greedy Path Cover The minimum path cover is computed in two stages. First we find a greedy path cover, as shown in Algorithm 2, by iteratively adding one path that covers the most number of uncovered nodes. In each iteration, the graph is scanned in topological order. For node v we update $maxcover[v]$ with the maximum number of uncovered nodes in a path that ends at v, and $pre[v]$ records which node is the previous one in that path. At last, a path traced back from $argmax_v\ maxcover[v]$ is added to current collection of paths. Uncovered nodes in this path are marked. We keep track of total number of covered nodes, and iterate until all nodes are covered.

Shrinking to Minimum Path Cover The second step is to shrink the above path cover to a minimum one. This is done by converting the path cover to a flow on the split graph. Then this minimum flow problem on a graph where each edge has only lower bounds is converted to a standard maximum flow problem with only upper capacity bounds. The maxflow here is exactly the maximum number of paths that can be removed from the path cover. In the residual network, we can factor the flow left back to a path cover which is minimum.

To address the memory issue, we implemented a simple maxflow solver to avoid copying the pan-genome graph. The solver uses Dinic's algorithm [10] with a few optimizations.

Although the computation of minimum path cover can be paralleled by covering each connected component separately and simultaneously, this is not implemented since the current solver is already fast enough in a single thread, and most of our test graphs have only one connected component (or two if the reverse complement graph counts).

Index Array We compute the MPC index slightly different from [21]. The index stores two parts the are necessary to stay in resident memory to support the later sparse dynamic programming framework. One is $path[v]$ which is a list for each node $v \in V$ containing the index of paths that contain v. Another is $backwards[v]$, similar to the `forward` links but is a list of pairs (u, k) to indicate that u is the last node on path P_k that can reach v. Both of these two arrays are computed separately for each connected component and its minimum path cover.

Algorithm 2: Greedy path cover

Input: Graph $G = (V, E)$

Output: A greedy path cover $P = P_1 \dots P_K$ (where K is not necessarily minimum but within $O(k \log|V|)$)

```
path_cover ← ∅;
covered_count ← 0;
for j ← 1 to |V| do
    covered[j] ← 0;
while covered_count < |V| do
    for j ← 1 to N do
        maxcover[j] ← 0;
        pre[j] ← None;
    for v ∈ V in topological order do
        if covered[v] == 0 then
            maxcover[v] ← maxcover[v] + 1;
        for (v, u) ∈ E do
            if maxcover[u] < maxcover[v] then
                maxcover[u] ← maxcover[v];
                pre[u] ← v;
    T ← argmax_v maxcover[v];
    path ← ∅;
    while covered_count < |V| do
        if covered[T] == 0 then
            covered_count ← covered_count + 1;
            covered[T] ← 1;
        path ← {T} ∪ path;
        T ← pre[T];
    path_cover ← path_cover ∪ {path};
return path_cover
```

3.3 Implementation of Co-linear Chaining

When implementing Algorithm 1, a key difference is that in practice, even the long read is very short compared to the entire pan-genome graph. Therefore some operations are more expensive than their theoretical complexity. For example, the `for` loop that enumerates node in topological order will take $O(|V|)$ even as an empty loop. In fact the algorithm only cares about nodes where $start[v]$ and $end[v]$ are not empty, or some (u, k) in the forward links of v has non-empty $start[u]$. These are essentially $O(N)$ nodes which is much smaller in practice.

Therefore we first compute the topological order as part of the index, along with the index in it for each node. When given a set of anchors, we extract all the start and end nodes in anchors, and nodes whose forward links reach some starts, which are exactly those in the array `backwards[v]`. Then we sort these nodes according to their appearance order in the precomputed topological order.

Data Structure There is a similar issue with data structures supporting range maximum queries. Some simple choices for such data structure such as segment tree [7] will take a huge amount of memory that is proportional to the maximum range of the anchor intervals, which can be $|V|$. Especially when running co-linear chaining for a set of reads in parallel, a 30-core machine might need space as big as $30 \times |V| \times k$.

Our choice is a balanced binary search tree. The one used here is `treap`, but the method to support range maximum queries can extend to any binary search trees as in [20]. Here the key for each insertion is a structure such as $(x, (c, j))$ where x is key in the search tree, c is the main value to compare with during range maximum queries, and j is the index of the previous anchors for tracebacks. For each node in the binary search tree, we store the $(key, value)$ pair and also a $max = (max_c, max_j)$ which is the maximum value in the subtree rooted at this node. This can be maintained in each insertion and self-balancing rotations. For a range maximum query at (l, r), we search for l and r simultaneously and record a temporary maximum value of the nodes and sub-tree `max` that fall between l and r. The size of a single binary search tree is only proportional to the distinct value of inserted keys, which is only possibly $A[j].y$ for some anchor $A[j]$. Although each split short read may have many anchors and thus the total number of anchors N may be large, the total number of distinct y where the splits end is much smaller. Hence the data structure is extremely space-efficient and enables parallel execution of this pipeline.

3.4 Transforming a Chain to an Alignment

The above algorithm only returns a chain with maximum coverage, which is an ordered list of anchors, but not an alignment between the sequence and the graph yet. Suppose the chain obtained is $\mathcal{C} = A_1, A_2, \ldots, A_p$; we need to find a path on the graph from $A_i.end$ to $A_{i+1}.start$ for each $1 \leq i \leq p-1$. In addition, we wish that the long path formed by the paths of the anchors and these connecting paths has a small edit distance with the read sequence. Ideally, we can use something similar to bit-parallel algorithms on graphs to extend from one end on the graph until the other end is reached, but the readily available tools in GraphAligner only provides a single-direction extension, and implementing such tool with both efficiency and alignment quality might be well beyond the focus of co-linear chaining algorithms. Besides, the extension may find paths that are optimal connecting adjacent anchors, but anchors themselves are already not optimal. The final long path cannot match the read sequence well if the anchors have poor qualities, even if the connecting paths are unnecessarily optimal. In addition, the need to connect these parts originates from the lack of anchors in the region that can be chained, which likely indicates 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. In the end, we test the simplest way to connect two nodes on the graph, by performing breadth-first search (BFS). The paths by BFS match the read surprisingly well in our experiments.

Gap Another problem in converting a chain to an alignment is that the connected path might be several times longer than the read. This is because the first and the last anchor in a maximum coverage chain is replaceable if there is another anchor with the same interval and a path that is distant but still able to be chained. Since anchors are obtained by aligning the same short split sequence, there are usually several false-positive anchors for each interval. If some chain accidentally starts or ends with one of such anchors, the connected path may easily span over nearly half of the graph. To fix this drawback of finding anchors by splitting, we introduce another parameter $G =$`colinear-gap`. If some connecting path has or will have more than G base pairs, the chain is split and only the longer one is kept. This may lead to multiple gaps when connecting a chain. This way the remaining part of the chain is less likely to include false-positive anchors. Figure 3.2 shows an example of a chain of 4 anchors.

Figure 3.2: An example for gap breaking. Each green block indicates an anchor, and a blue block indicates the BFS connecting path. Depending on the parameters, the first anchor might be ignored due to the large gap between the first and the second anchor.

Edit Distance Alignment With the long path obtained and the input read, the alignment task is reduced to sequence-to-sequence alignment. We use a package `edlib` to compute the global edit distance alignment between the two sequences, and map it back to the path as an alignment on the graph.

The alignment by co-linear chaining is compared with the one from GraphAligner in terms of global edit distance. The better alignment is selected first. This way the aligner can successfully align most of the reads. For example, when the errors in reads are many but clustered, so that some anchor are accurate but scattered in the graph, co-linear chaining will likely find an alignment containing all these clusters; on the other hand, when the anchors are of poor quality, it is impossible for co-linear chaining to find anything better than seed-and-extend alignments, so the alignments from GraphAligner will be selected.

4 Experimental Results

We present here several experiments to demonstrate the practical performance of our co-linear chaining aligner. First, we introduce the design of the experiments and the data used. Then, we show results under various combinations of parameters, and compare our tool `GraphChainer` with `GraphAligner` [27], which is a state-of-the-art aligner of long reads to a pan-genome graph.

4.1 Experiment Design

Our first experiment consists of aligning simulated PacBio long reads. Given a variation graph which is a DAG, we sample a long path from the graph by walking randomly on the graph and concatenate the node labels on the path to obtain a reference sequence. The ambiguous characters on the simulated reference sequence will be randomly replaced by one of its indicated characters. We simulate a set of 15x coverage PacBio long reads with package `Badread` [32]. Badread is used with parameters:

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

For each simulated read, we know its original path on the graph as the ground truth by mapping its interval on the reference. We use `GraphAligner` to align the reads directly as a baseline, and to compute the anchors for the co-linear chaining algorithm. GraphAligner is used as baseline with parameters:

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

and `GraphChainer` is used with parameters:

```
GraphChainer -t 30 -x vg -f {Reads} -g {Graph} -a {clc_gam}
   --short-verbose
```

The `-short-verbose` enables collecting statistics such as the number of chains that contain one-node overlaps and the CPU time spent in each phase for each read.

For an aligned set of reads, we compute the global edit distance between the read and the path labels with package `edlib`. If this distance divided by the length of the read is smaller than a certain threshold, we say that this read is aligned correctly. For example, the error rate of simulated data is about 25%, so if a read of length 10000 is aligned with edit distance 2500, the alignment is good enough. The simulated errors will make the ground-truth path indistinguishable from this alignment. The co-linear chaining alignment is compared to the baseline in terms of the number of well-aligned reads under various thresholds. We also compute the global edit distance between the path sequence of this alignment and the ground truth path of the read. This indicates the overlapping of the alignment and the ground truth. Ideally, this distance is zero when the alignment path is exactly the ground-truth path.

We also test the algorithm on real PacBio long reads. For real reads, the ground truth path is not available or even not existing if the variants in reads are not present in the graph. So we only compute the edit distance between the read and the alignment.

All experiments are conducted on a server with AMD Ryzen Threadripper PRO 3975WX 32-Cores and 504Gb of RAM. All scripts and programs are given `-t 30` in the arguments to run with 30 threads. The time and peak memory usage of each program are measured with command `/usr/bin/time -v`.

4.2 Data

Graphs We use two graphs from [14]: LRC.vg and MHC1.vg. We also build a variation graph of chromosome 22 using vg toolkit with GRCh37 as the reference, and variants from the Thousand Genomes Project phase 3 release [5]. We replaced all ambiguous characters in GRCh37. GraphAligner searches for seeds within continuous node labels, and therefore, the non-branching path nodes in the graph should be grouped into a single node with long labels. The command for vg is

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

To test the case when the graph is extremely large, we also build the variation graph of chromosome 1.

Reads For the real PacBio reads, we use the same dataset as in [27] with SRA accession SRX4480530. We first aligned all the reads against GRCh37 with `minimap2` [19] and

selected only the reads that are aligned to chromosome 22 with at least 70% of their length, and no longer alignments to other chromosomes. This leads to $136,494$ reads with $2,858,621,416$ total base pairs, which is roughly 56x coverage on chromosome 22. For chromosome 1, we filtered a similar data set of $907,572$ reads which is roughly 79x coverage.

Table 4.1 shows statistics of graphs and the reads used in our experiments.

Graph	nodes	splitted nodes	node label bps	width	total reads	total read bps
LRC	117,787	122,227	1,099,856	4	1,093	15,872,214
MHC1	479,531	504,883	5,138,362	4	5,091	74,524,274
Chr22 sim	3,197,160	3,632,307	52,423,213	7	52,464	769,238,818
Chr22 real					136,494	2,858,621,416
Chr1 real	18,807,963	20,428,949	255,754,179	9	907,572	19,617,046,919

Table 4.1: Statistics of each data set. In our experiments, all graphs have 2 connected components that have the same number of nodes, number of edges and width, as one component is the reverse complement of the other one. So we show the statistics of one component, and the statistics will be doubled on the entire graph.

4.3 Results

Metrics To compare the overall quality of alignments, we count the number of reads that are aligned with an edit distance smaller than a given threshold $0 < \sigma \leq 1$ multiplied by the read length, which are called good alignments. For example, given a read of length 10000, an alignment with edit distance 2000 is consider good for a threshold $\sigma > 0.2$ and not good for $\sigma \leq 0.2$. This threshold is related to the error rate of the long read sequences.

LRC and MHC1 On average, when the threshold is between 20% and 50%, our co-linear chaining algorithm has aligned 2% more reads compared to GraphAligner. At threshold 35%, which is slightly above the error rate of most of the reads, co-linear chaining reduced the number of unaligned reads by half. When comparing in terms of ground truth overlapping, the advantage of co-linear chaining is more obvious. Figure 4.1 shows the results on LRC.vg. Figure 4.2 shows the results on MHC1.vg.

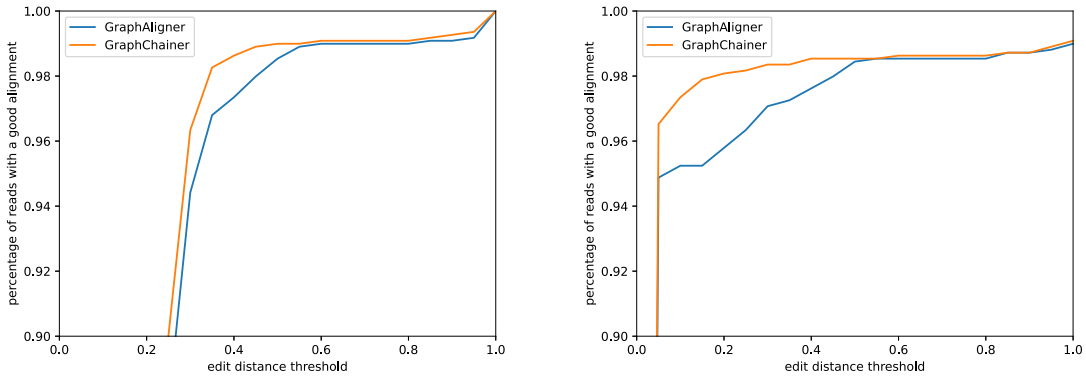

Figure 4.1: The number of good alignments at each threshold, for simulated data on LRC.vg.

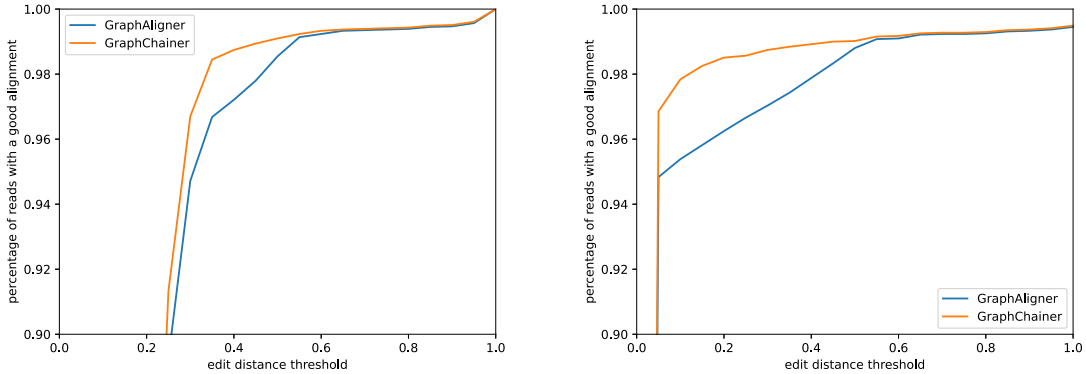

Figure 4.2: The number of well-aligned reads at each threshold, for simulated data on MHC1.vg.

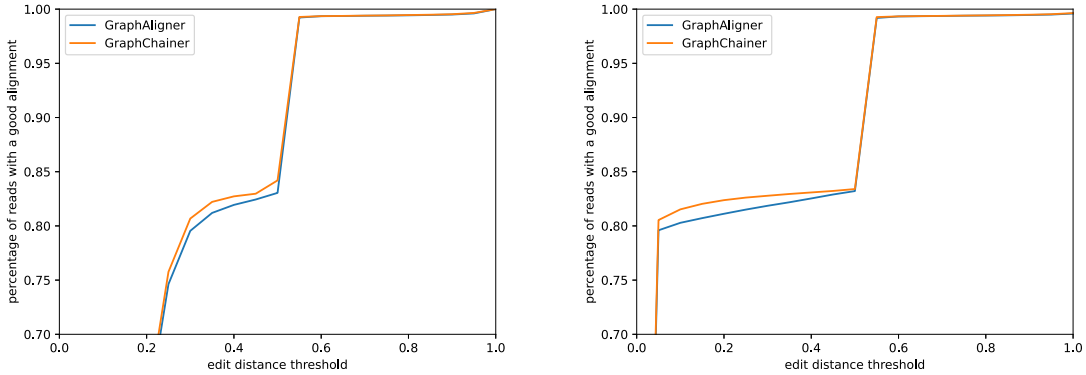

Figure 4.3: The number of well-aligned reads at each threshold, for simulated data on chr22.vg

Chromosome 22 For chromosome 22, the graph size is much larger than the average length of simulated reads. The results are shown in Figure 4.3.

The curve in Figure 4.3 looks very different from the curves in Figure 4.1 and Figure 4.2. One reason might be that randomly replacing the ambiguous character indepently violates some assumption of the simulator, so that reads simulated from the region which originally has only ambiguous characters follows a different pattern.

Analysis on Simulated Data Table 4.2 shows the number of good reads for each graph. Here we use 0.4 as the threshold for good alignments with edit distance between input reads, and 0.2 as the threshold for overlapping alignments with edit distance between the ground truth path. On all graphs, `GraphChainer` find a good alignment for 1% more of the reads under threshold 0.4, and an overlapping alignment for 2% more of the reads under threshold 0.2.

Graph	Aligner	Good Reads by edit distance	Good Reads bps by edit distance	Overlap Reads by true path	Overlap Reads bps by true path
LRC	GraphAligner	1064(97.35%)	15402957(97.04%)	1047(95.79%)	15177005(95.62%)
	GraphChainer	1078(98.63%)	15712725(99.00%)	1072(98.08%)	15668530(98.72%)
MHC1	GraphAligner	4949(97.21%)	71933675(96.52%)	4900(96.25%)	71028183(95.31%)
	GraphChainer	5027(98.74%)	73735165(98.94%)	5015(98.51%)	73534431(98.67%)
Chr22 simulated	GraphAligner	42989(81.94%)	630159099(81.92%)	42559(81.12%)	622012110(80.86%)
	GraphChainer	43402(82.73%)	638983670(83.07%)	43218(82.38%)	636460703(82.74%)
Chr22 real	GraphAligner	123353(90.37%)	2450869693(85.74%)		
	GraphChainer	134771(98.74%)	2825239847(98.83%)		
Chr1 real	GraphAligner	830989(91.56%)	17209566360(87.73%)		
	GraphChainer	880549(97.02%)	19065996572(97.19%)		

Table 4.2: Quality of alignments with each aligner on simulated and real PacBio long reads.

Real Data For the real PacBio reads on chromosome 22, the the difference between `GraphAligner` and `GraphChainer` is more obvious than that on simulated data. As shown in Table 4.2, `GraphAligner` can align only 90.37% of the reads within threshold 40%, while co-linear chaining aligned 98.74% within the same threshold. Although if we use the threshold 90%, which is similar to the 10% overlapping standard in [27], the difference is insignificant.

For chromosome 1, the improvement is slightly smaller but still significant. For the 8.44% reads that are not aligned within 40% threshold, `GraphChainer` found a good alignment

for 64.7% of them, and reduced this ratio to 2.98% of the total number of reads. Figure 4.4 and Figure 4.5 shows the number of good alignments for chromosome 22 and 1.

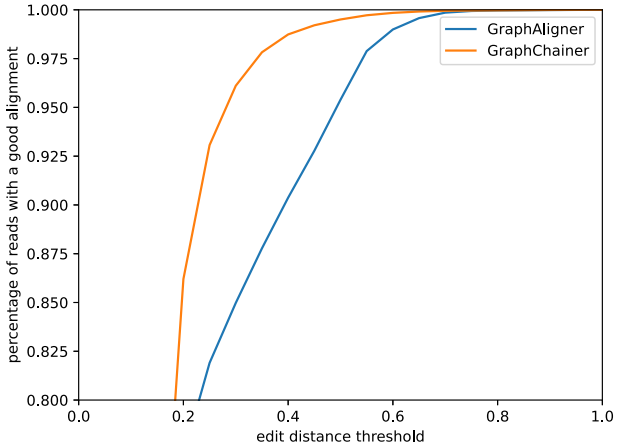

Figure 4.4: The number of good alignments at each threshold, for real PacBio data on chromosome 22.

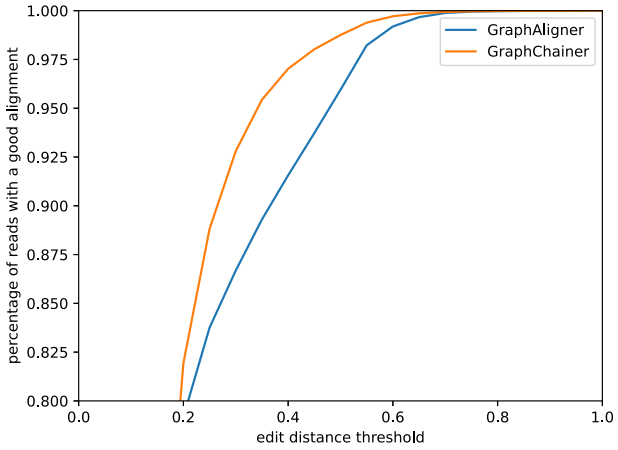

Figure 4.5: The number of good alignments at each threshold, for real PacBio data on chromosome 1.

Effect of One-node Overlapping Table 4.3 shows the effect of allowing one-node overlapping in co-linear chaining. It can be seen that over 90% of the chains contains one-node overlapped anchors. For the real data on chromosome 22 this ratio becomes 97%. Although it does not implies that there is no other maximum-coverage chains without such overlaps, it provides reasons to allow such overlaps.

Graph	Total Reads	Overlapping Optimal Chains	After Gap Splitting
LRC	1093	1007(92.1%)	998(91.3%)
MHC1	5091	4731(92.9%)	4706(92.4%)
Chr22 real	136494	133111(97.5%)	132718(97.2%)

Table 4.3: Impact of allowing one-node overlapping. The "Overlapping Optimal Chains" count the number of reads that maximum-coverage chain has at least one pair of anchors that are one-node overlapped. "After Gap Splitting" consider only the final chain after breaking large gaps.

Speed The co-linear chaining takes anchors from short GraphAligner alignments as input, so the running time can be much longer than directly aligning with GraphAligner. The short length of split reads gives rise to many false-positive anchors, which consumes extra time that is not needed when directly aligning. Table 4.4 shows the speed and memory usage of both aligners with 30 threads in the experiment on real data.

Memory The memory usage of co-linear chaining is slightly larger. The size of the MPC index is $O(k|V|)$. If the graph has more than one connected components, the path cover can be split into groups and reduces space. The data structures during the dynamic programming process take $O(N)$ each and $O(kN)$ in total, where N is the number of anchors. In addition, the data structure part is independent for each thread, while the MPC index is shared.

Chromosome	Aligner	Peak Memory(GiB)	CPU Time(hh:mm:ss)	Real Time(hh:mm:ss)
22	GraphAligner	8.92	1:30:08	0:03:16
	GraphChainer	10.46	6:08:33	0:12:44
1	GraphAligner	19.3	10:37:23	0:22:44
	GraphChainer	58.1	135:55:43	4:35:36

Table 4.4: Speed and memory usage of co-linear chaining aligner when aligning real PacBio reads.

Time in Each Phase The time spent on each part of the algorithm using 30 threads is shown in Table 4.5. The most time-consuming part is finding anchors from short read alignments. For chromosome 1, there are 7803 anchors on average for each read, which is much more than 2233 anchors in the case of chromosome 22.

We can estimate the running time of co-linear chaining using the theoretical time complexity $O(kN \log|V|)$. When comparing chromosome 1 to chromosome 22, the number of reads is 6.6x, and number of anchors is 3.5x, and there is a factor $\frac{9}{7}$ due to the widths,

multiplied by the factor from $\log|V|$ which is about 1.25x. Given that the CPU time spent in co-linear chaining for chromosome 22 is 49 minutes, an estimation of the CPU time of this part for chromosome 1 is 30 hours and 19 minutes. This matches the measured CPU time $30:11:54$ in our experiments.

The time for the co-linear chaining part is relatively short. Since building the MPC index is faster than loading the graph from disk, its time usage is omitted together with I/O time and indexing time by `GraphAligner`.

Dataset	anchor time	chaining time	edlib time	total time
chr22 simulated	0:38:24	0:06:22	0:09:27	1:03:27
chr22 real	4:02:14	0:49:06	0:46:10	6:08:33
chr1 real	95:30:45	30:11:54	5:57:42	135:55:43

Table 4.5: CPU time spent in each phase of the aligner on the chromosome 22 graph and the chromosome 1 graph. Time is shown in format hh:mm:ss.

5 Conclusions

We extended the dynamic programming framework to solve co-linear chaining on general graphs by a reduction to CLC with one-node overlaps on DAGs. This extension would enable more general applications of graphical representations of genomes. The algorithm is implemented for DAGs as a sequence-to-graph aligner `GraphChainer`. We experimentally show that `GraphChainer` significantly improves alignment quality and has competitive performance when compared with a state-of-the-art aligner `GraphAligner`. The improvement of alignment quality is significant for real PacBio long reads on human genomes. In the experiments, the time complexity of our algorithm is empirically confirmed.

Although our implementation uses `GraphAligner` to find anchors, the CLC module is relatively independent. One may compute the anchors with other methods and feed the anchors to our CLC module. This can be added as a separate software in the future.

For now, the implementation assumes that the input graph is a DAG. The reduction from graphs with cycles to DAGs and the procedure of mapping anchors between the two graphs may be added in future developments.

As an initial implementation, there are many ways to improve the performance further. One possible direction is to replace the method to connect adjacent anchors. Instead of BFS, a better way is to use alignment algorithms that can extend from both directions, where both the starting node and the ending node are determined. A modified bit-parallel algorithm might also work.

One immediate application of our aligner would be genotyping, or variant calling, by aligning a set of reads to the variants graph built from a set of known variants, as done in [27]. With improved quality of alignments, we can expect better accuracy and recall on the genotyping task.

The algorithm can extend to graphs with cycles easily, although such graphs in the application are relatively rare compared to DAGs as the variation graphs in our experiments. We hope that our aligner enables more applications on graphs with cycles.

Bibliography

[1] M. I. Abouelhoda and E. Ohlebusch. "Chaining algorithms for multiple genome comparison". In: *Journal of Discrete Algorithms* 3.2-4 (2005), pp. 321–341.

[2] D. Antipov, A. Korobeynikov, J. S. McLean, and P. A. Pevzner. "hybridSPAdes: an algorithm for hybrid assembly of short and long reads". In: *Bioinformatics* 32.7 (2016), pp. 1009–1015.

[3] A. Backurs and P. 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*. 2015, pp. 51–58.

[4] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, et al. "SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing". In: *Journal of computational biology* 19.5 (2012), pp. 455–477.

[5] L. Clarke, S. Fairley, X. Zheng-Bradley, I. Streeter, E. Perry, E. Lowy, A.-M. Tassé, and P. Flicek. "The international Genome sample resource (IGSR): A worldwide collection of genome variation incorporating the 1000 Genomes Project data". In: *Nucleic Acids Research* 45.D1 (Sept. 2016), pp. D854–D859. ISSN: 0305-1048. DOI: 10.1093/nar/gkw829.

[6] 1. G. P. Consortium et al. "A global reference for human genetic variation". In: *Nature* 526.7571 (2015), p. 68.

[7] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf. "Computational geometry". In: *Computational geometry*. Springer, 1997, pp. 1–17.

[8] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. "Alignment of whole genomes". In: *Nucleic acids research* 27.11 (1999), pp. 2369–2376.

[9] A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg. "Fast algorithms for large-scale genome alignment and comparison". In: *Nucleic acids research* 30.11 (2002), pp. 2478–2483.

[10] E. A. Dinic. "Algorithm for solution of a problem of maximum flow in networks with power estimation". In: *Soviet Math. Doklady*. Vol. 11. 1970, pp. 1277–1280.

[11] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. "Sparse dynamic programming I: linear cost functions". In: *Journal of the ACM (JACM)* 39.3 (1992), pp. 519–545.

[12] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, S. Garg, C. Markello, M. F. Lin, et al. "Variation graph toolkit improves read mapping by representing genetic variation in the reference". In: *Nature biotechnology* 36.9 (2018), pp. 875–879.

[13] P. Ivanov, B. Bichsel, H. Mustafa, A. Kahles, G. Rätsch, and M. Vechev. "Astarix: Fast and optimal sequence-to-graph alignment". In: *International Conference on Research in Computational Molecular Biology*. Springer. 2020, pp. 104–119.

[14] C. Jain, S. Misra, H. Zhang, A. Dilthey, and S. Aluru. "Accelerating Sequence Alignment to Graphs". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 451–461. DOI: 10.1109/IPDPS.2019.00055.

[15] C. Jain, S. Misra, H. Zhang, A. Dilthey, and S. Aluru. "Accelerating sequence alignment to graphs". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 451–461.

[16] K. Kukich. "Techniques for Automatically Correcting Words in Text". In: *ACM Comput. Surv.* 24.4 (Dec. 1992), pp. 377–439. ISSN: 0360-0300. DOI: 10.1145/146370.146380.

[17] A. Kuosmanen, T. Paavilainen, T. Gagie, R. Chikhi, A. Tomescu, and V. Mäkinen. "Using Minimum Path Cover to Boost Dynamic Programming on DAGs: Co-linear Chaining Extended". In: *Research in Computational Molecular Biology*. Ed. by B. J. Raphael. Cham: Springer International Publishing, 2018, pp. 105–121. ISBN: 978-3-319-89929-9.

[18] V. I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[19] H. Li. "Minimap2: pairwise alignment for nucleotide sequences". In: *Bioinformatics* 34.18 (May 2018), pp. 3094–3100. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty191.

[20] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

[21] V. Mäkinen, A. I. Tomescu, A. Kuosmanen, T. Paavilainen, T. Gagie, and R. Chikhi. "Sparse dynamic programming on DAGs with small width". In: *ACM Transactions on Algorithms (TALG)* 15.2 (2019), pp. 1–21.

[22] G. Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming". In: *Journal of the ACM (JACM)* 46.3 (1999), pp. 395–415.

[23] G. Myers and W. Miller. "Chaining Multiple-Alignment Fragments in Sub-Quadratic Time". In: *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '95. San Francisco, California, USA: Society for Industrial and Applied Mathematics, 1995, pp. 38–47. ISBN: 0898713498.

[24] G. Navarro. "Improved approximate pattern matching on hypertext". In: *Theoretical Computer Science* 237.1-2 (2000), pp. 455–463.

[25] S. C. Ntafos and S. L. Hakimi. "On Path Cover Problems in Digraphs and Applications to Program Testing". In: 5.5 (Sept. 1979), pp. 520–529. ISSN: 0098-5589. DOI: 10.1109/TSE.1979.234213.

[26] M. Rautiainen, V. Mäkinen, and T. Marschall. "Bit-parallel sequence-to-graph alignment". In: *Bioinformatics* 35.19 (Mar. 2019), pp. 3599–3607. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btz162.

[27] M. Rautiainen and T. Marschall. "GraphAligner: rapid and versatile sequence-to-graph alignment". In: *Genome biology* 21.1 (2020), pp. 1–28.

[28] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. "Reducing storage requirements for biological sequence comparison". In: *Bioinformatics* 20.18 (2004), pp. 3363–3369.

[29] M. Roberts, B. R. Hunt, J. A. Yorke, R. A. Bolanos, and A. L. Delcher. "A preprocessor for shotgun assembly of large genomes". In: *Journal of computational biology* 11.4 (2004), pp. 734–752.

[30] L. Salmela and E. Rivals. "LoRDEC: accurate and efficient long read error correction". In: *Bioinformatics* 30.24 (2014), pp. 3506–3514.

[31] M. Šošić and M. Šikić. "Edlib: a C/C++ library for fast, exact sequence alignment using edit distance". In: *Bioinformatics* 33.9 (Jan. 2017), pp. 1394–1395. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw753.

[32] R. R. Wick. "Badread: simulation of error-prone long reads". In: *Journal of Open Source Software* 4.36 (2019), p. 1316. DOI: 10.21105/joss.01316.