Chaining of Maximal Exact Matches in Graphs

Nicola Rizzo
nicola.rizzo@helsinki.fi

Manuel Cáceres
manuel.caceres@helsinki.fi

Veli Mäkinen
veli.makinen@helsinki.fi

Department of Computer Science, University of Helsinki, Finland

Abstract

We show how to chain *maximal exact matches* (MEMs) between a query string Q and a labeled directed acyclic graph (DAG) $G = (V, E)$ to solve the *longest common subsequence* (LCS) problem between Q and G. We obtain our result via a new symmetric formulation of chaining in DAGs that we solve in $O(m + n + k^2|V| + |E| + kN \log N)$ time, where $m = |Q|$, n is the total length of node labels, k is the minimum number of paths covering the nodes of G and N is the number of MEMs between Q and node labels, which we show encode full MEMs.

Keywords sequence to graph alignment, longest common subsequence, sparse dynamic programming

Funding This project received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 956229, and from the Academy of Finland grants No. 352821 and 328877.

1 Introduction

Due to recent developments in *pangenomics* [6] there is a high interest to extend the notion of string alignments to graphs. A common pangenome representation is a node-labeled directed acyclic graph (DAG), whose paths represent plausible individual genomes from a species. Unfortunately, even finding an exact occurrence of a query string as a subpath in a graph is a conditionally hard problem [11, 10]: only quadratic time dynamic programming solutions are known and faster algorithms would contradict the Strong Exponential Time Hypothesis (SETH). Due to this theoretical barrier, parameterized solutions have been developed [2, 8, 7, 18], and/or the task has been separated into finding short exact occurrences (anchors) and then *chaining* them into longer matches [16, 13, 14, 5]. Although the chaining algorithms provide exact solutions to their internal chaining formulations and their solutions can be interpreted as alignments of queries to a graph with edit operations, so far they have not been shown to provide exact solutions to the corresponding alignment formulation.

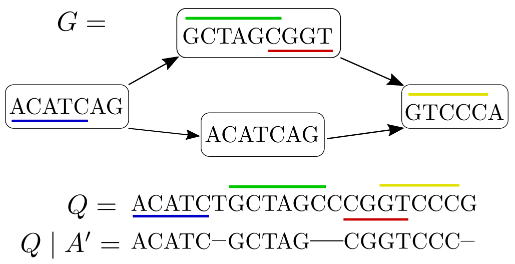
In this paper, we integrate a symmetric formulation from string chaining [19, 15] to graph chaining [16] yielding the first chaining-based parameterized exact alignment algorithm between a query string and a graph. Namely, we obtain an $O(m+n+k^2|V|+|E|+kN\log N)$ time algorithm for computing the length of a *longest common subsequence* (LCS) between a query string Q and a path of G, where $m=|Q|$, n is the total length of node labels, k is the width (minimum number of paths covering the nodes) of G, and N is the number of *maximal exact matches* (MEMs) between Q and the node labels (node MEMs).

The paper is structured as follows. The preliminaries in Section 2 and the basic concepts in Section 3 follow the notions developed in our recent work [17], where we introduce the definition of a MEM between a string and a graph, and study the non-trivial problem of finding graph MEMs with a length threshold; for the purposes of this paper, we observe that node MEMs are sufficient. In Section 4.1, we revise the solution for an asymmetric chaining formulation in DAGs [16] for the case of node MEMs. Then, in Section 4.2, we tailor the string to string symmetric chaining algorithm [19, 15] to use MEM anchors. In Section 4.3, we show how to integrate these two approaches to obtain our main result. Finally, in Section 5 we discuss the length threshold setting and cyclic graphs.

2 Preliminaries

Strings. We work with strings coming from a finite alphabet $\Sigma=[1..\sigma]$ and assume that σ is at most the length of the strings we work with. For two integers x and y we use $[x..y]$ to denote the integer interval $\{x,x+1,\dots,y\}$ or the empty set $\emptyset$ when $x>y$. A *string* T is an element of Σ^n for a non-negative integer n, that is sequence of n symbols from Σ, where $n=|T|$ is the *length* of the string. We denote ε to the only string of length zero. We also denote $\Sigma^+=\Sigma^*\setminus\{\varepsilon\}$. For two strings T_1 and T_2 we denote their *concatenation* as $T_1\cdot T_2$, or just T_1T_2. For a set of integers I and a string T, we use $T[I]$ to denote the *subsequence* of T made of the concatenation of the characters indicated by I in increasing order. If I is an integer interval $[x..y]$, then $T[x..y]$ is a *substring*: if $x=y$ then we also use $T[x]$, if $y<x$ then $T[x..y]=\varepsilon$, if $x\le y=n$ we call it a *suffix* (*proper suffix* when $x>1$) and if $1=x\le y$ we call it a *prefix* (*proper prefix* when $y<n$). A length-κ' substring $Q[x..x+\kappa'-1]$ *occurs* in T if $Q[x..x+\kappa'-1]=T[i..i+\kappa'-1]$; in this case, we say that (x,i,κ') is an *(exact) match* between Q and T, and *maximal* (a MEM) if the match cannot be extended to the left (*left-maximality*), that is, $x_1=1$ or $x_2=1$ or $Q[x_1-1]\neq T[x_2-1]$ nor it can be extended to the right (*right-maximality*) $x_1+\ell=|Q|$ or $x_2+\ell=|T|$ or $Q[x_1+\ell]\neq T[x_2+\ell]$.

Labeled graphs. We work with labeled directed acyclic graphs (DAGs) $G=(V,E,\ell)$, where V is the vertex set, E the edge set, and $\ell: V\to\Sigma^+$ a *labeling* function on the vertices. A length-k *path* P from v_1 to v_k is a sequence of nodes $v_1,\dots,v_k$ such that $(v_1,v_2),(v_2,v_3),\dots,(v_{k-1},v_k)\in E$, in this case we say that v_1 *reaches* v_k. We extend the labeled function to paths by concatenating the corresponding node labels, that is, $\ell(P):=\ell(v_1)\cdots\ell(v_k)$. For a node v and a path P we use $\|\cdot\|$ to denote its *string length*, that is $\|v\|=|\ell(v)|$ and $\|P\|=|\ell(P)|$. We say that a length-κ' substring $Q[x..x+\kappa'-1]$ *occurs* in G if $Q[x..x+\kappa'-1]$ occurs in $\ell(P)$ for some path P. In this case, we say that $([x..x+\kappa'-1],(i,P=v_1\dots v_k,j))$ is an *(exact) match* between Q and G, where $Q[x..x+\kappa'-1]=\ell(v_1)[i..]\cdot\ell(v_2)\cdots\ell(v_{k-1})\cdot\ell(v_k)[..j]$, with $1\le i\le\|v_1\|$ and $1\le j\le\|v_k\|$. We call the triple (i,P,j) a *substring* of G



$G =$
GCTAGCGGT
ACATCAG
ACATCAG
GTCCCA
$Q =$ ACATCTGCTAGCCCGGTCCCG
$Q \mid A' =$ ACATC–GCTAG—CGGTCCC–

Figure 1: Co-linear chaining setting between a string Q and a labeled graph G. If v is the last node to the right, then $([16..20], (1, v, 5))$ is a match, with $\mathrm{lext}(1, v, 5) = \{\mathtt{G}, \mathtt{T}\}$ and $\mathrm{rext}(1, v, 5) = \{\mathtt{A}\}$. It is a MEM, since $|\mathrm{lext}(i, P, j)| \geq 2$ and it cannot be extended to the right (Definition 1). In fact, all exact matches are MEMs and they form a symmetric chain A' (blue-green-red-yellow) inducing the subsequence $Q \mid A'$ (the last `C` of green match is omitted due to overlap with the red match).

and we define its *left-extension* $\mathrm{lext}(i, P, j)$ as the singleton $\{\ell(v_1)[i-1]\}$ if $i > 1$ and $\{\ell(u)[\|u\|] \mid (u, v_1) \in E\}$ otherwise. Analogously, the *right-extension* $\mathrm{rext}(i, P, j)$ is $\{\ell(v_k)[j+1]\}$ if $j < \|v_k\|$ and $\{\ell(v)[1] \mid (v_k, v) \in E\}$ otherwise. Note that the left (right) extension can be equal to the empty set $\emptyset$, if the start (end) node of P does not have incoming (outgoing) edges. See Figure 1.

Chaining of matches. An *asymmetric chain* $A'[1..N']$ is an ordered subset of a set A of N exact matches between a labeled DAG $G = (V, E, \ell)$ and a query string Q, with the ordering $A'[l] < A'[l+1]$ for $1 \leq l < N'$ defined as $([x'..x' + \kappa'' - 1], (i', P_l, j')) < ([x..x + \kappa' - 1], (i, P_{l+1}, j))$ iff the start of path P_{l+1} is strictly reachable from the end of path P_l and $x' \leq x$. The asymmetry comes from the fact that overlaps are not allowed in G, but they are allowed in Q. We are interested in chains that maximize the length of an induced subsequence Q', denoted $Q' = Q \mid A'$, that is obtained by deleting all parts of Q that are not covered by chain A'. For example, consider $Q =$ `ACATTCAGTA` and $A' = ([2..4], (i_1, P_1, j_1)), ([3..6], (i_2, P_2, j_2)), ([9..10], (i_3, P_3, j_3))$. Then $Q' = Q \mid A' =$ `CATTCTA`; anchors cover the underlined part of $Q =$ `A`$\underline{\texttt{CATTC}}$`AG`$\underline{\texttt{TA}}$.

We could define symmetric chains by considering overlaps of paths, but for the purposes of this paper it will be sufficient to consider overlaps of exact matches inside the nodes of G: A *symmetric chain* $A'[1..N']$ is an ordered subset of a set A of N exact matches between the nodes of a labeled DAG $G = (V, E, \ell)$ and a query string Q, with the ordering $A'[l] < A'[l+1]$ for $1 \leq l < N'$ defined as $([x'..x' + \kappa'' - 1], (i', v, j')) < ([x..x + \kappa' - 1], (i, w, j))$ iff (i) w is strictly reachable from v or $v = w$ and $i' \leq i$ and (ii) $x' \leq x$. We extend the notation $Q' = Q \mid A'$ to cover symmetric chains A' so that Q' is obtained by deleting all parts of Q that are not *mutually* covered by chain A'. We define mutual coverage in Section 4.2: Informally, Q' is formed by concatenating the prefixes of exact matches until reaching the overlap between the next exact match in the chain. Figure 1 illustrates the concept.

3 Finding MEMs in labeled DAGs

We now consider the problem of finding all *maximal exact matches* (MEMs) between a labeled graph G and a query string Q for the purpose of chaining.

Definition 1 (MEM between a pattern and a graph [17])**.** *Let $G = (V, E, \ell)$ be a labeled graph, with $\ell \colon V \to \Sigma^+$, and $Q \in \Sigma^+$. We say that a match $([x..y], (i, P, j))$ between Q and G is* left-maximal *(*right-maximal*) if it cannot be extended to the left (right) in both Q and G, that is,*

$$\begin{array}{rll} (\mathsf{LeftMax}) & x = 1 \vee \operatorname{lext}(i, P, j) = \emptyset \vee Q[x-1] \notin \operatorname{lext}(i, P, j) & \textit{and} \\ (\mathsf{RightMax}) & y = |Q| \vee \operatorname{rext}(i, P, j) = \emptyset \vee Q[y+1] \notin \operatorname{rext}(i, P, j). & \end{array}$$

The pair $([x..y], (i, P, j))$ is a MEM if it is left-maximal or its left (graph) extension is not a singleton, and right-maximal or its right (graph) extension is not a singleton, that is $\mathsf{LeftMax} \vee |\operatorname{lext}(i, P, j)| \geq 2$ *and* $\mathsf{RightMax} \vee |\operatorname{rext}(i, P, j)| \geq 2$.

See Figure 1 for an example. We use this particular extension of MEMs to graphs—with the additional conditions on non-singletons lext and rext—as it captures all MEMs between Q and $\ell(P)$, where P is a source-to-sink path in G. Moreover, we will show that this MEM formulation captures LCS through co-linear chaining, whereas avoiding the additional conditions would fail. Indeed, consider Q, G, and match $([16..20], (1, v, 5))$ from Figure 1: the match is not left-maximal, since $Q[15] = \texttt{G}$ and $\texttt{G} \in \operatorname{lext}(1, v, 5)$, but extending it would impose any chain using it as an anchor to go through the bottom suboptimal path, that in this case does not capture the LCS between Q and G. Also, it turns out that we can focus on MEMs between the node labels and the query, as chaining will cover longer MEMs implicitly.

To formalize the intuition, we say that a *node MEM* is a match (i, P, j) of $Q[x..y]$ in G such that $P = v$ and it is left and right maximal w.r.t. $\ell(P)$ only in the string sense: conditions $\mathsf{LextMax} \vee i = 1$ and $\mathsf{RightMax} \vee j = \|v\|$ hold. Consider the text

$$T_{\text{nodes}} = \prod_{v \in V} \mathbf{0} \cdot \ell(v),$$

where $\mathbf{0} \notin \Sigma$ is used as a delimiter to prevent MEMs spanning more than a node label. Running the MEM finding algorithm [1] on Q and T_{nodes} will retrieve exactly the node MEMs we are looking for [17] (a more involved problem of finding graph MEMs with a length threshold is studied in [17], but here a simplified result without the threshold is sufficient):

Lemma 1 ([17])**.** *Given a labeled DAG $G = (V, E, \ell)$, with $\ell : V \to \Sigma^+$, and a query string Q, we can compute all node MEMs between Q and G in time $O(n + m + N)$, where n is the total length of node labels, $m = |Q|$, and N is the number of node MEMs.*

Let A be the set of node MEMs found using Lemma 1. In Appendix A, we show that any long MEM spanning two or more nodes in G can be formed by concatenating node MEMs into *perfect chains*—chains that have no gap between consecutive matches.

Theorem 1. *For every MEM $([x..y], (i, P, j))$ between G and Q, there is a perfect chain $A'[1..p] \subseteq A$ such that $A'[1] \cdots A'[p] = ([x..y], (i, P, j))$.*

Corollary 1. *The set A is a* compact representation *of the set M of MEMs between query Q and a labeled DAG $G = (V, E, \ell)$: it holds $|A| \leq \|M\|$, where $\|M\|$ is the length of the encoding of the paths in MEMs as the explicit sequence of its nodes.*

Our strategy is to use set A as the representation of MEMs: Perfect chains are implicitly covered by the chaining algorithms of next section.

4 Symmetric co-linear chaining in labeled DAGs

Mäkinen et al. [16, Theorem 6.4] gave an $O(kN \log N + k|V|)$-time algorithm to find an asymmetric chain $A'[1..N']$ of a set A of N anchors[1] between a labeled DAG $G = (V, E, \ell)$ and a query string Q maximizing the length of an induced subsequence $Q' = Q \mid A'$. Here k is the *width* of G, that is, the minimum number of paths covering nodes V of G. The algorithm assumes a minimum path cover as its input, which can be computed in $O(k^2|V| + |E|)$ time [4, 3]. A limitation of this chaining algorithm is that anchors in the solution are not allowed to overlap in the graph, which has been partially solved by considering one-node overlaps [14]. However, both of these approaches maximize the length of the sequence induced by the reported chain only on the string Q, which makes the problem formulation asymmetric.

In the case of two strings as input, the asymmetry of the coverage metric was solved by Mäkinen and Sahlin [15] applying the technique by Shibuya and Kurochkin [19]. They provided an $O(N \log N)$-time algorithm to find a symmetric chain $A'[1..N']$ of a set A of N anchors maximizing the length of an *induced common subsequence* $C = Q \mid A' = T \mid A'$ between two input strings Q and T, that is obtained by deleting all parts of Q, or equivalently all parts of T, that are not *mutually covered* by chain A' (to be defined below). Here anchors are assumed to be exact matches (x, i, κ') (not necessarily maximal) such that $Q[x..x+\kappa'-1] = T[i..i+\kappa'-1]$, and $A'[j] < A'[j+1]$ for $1 \leq j < N'$, where the order $<$ between anchors is defined as $(x', i', \kappa'') < (x, i, \kappa')$ iff $x' \leq x$ and $i' \leq i$. For completeness, in Appendix B we include a revised proof that this algorithm computes the length of a longest common subsequence of strings Q and T if it is given all (string) MEMs between Q and T as input [15]. The concept of mutual coverage [15, Problem 1] is defined through the score

$$\texttt{coverage}(A') = \sum_{j=1}^{N'} \min_{\substack{(i,x,\kappa') := A'[j+1], \\ (i',x',\kappa'') := A'[j]}} \begin{cases} \min(i, i' + \kappa'') - i', \\ \min(x, x' + \kappa'') - x', \end{cases}$$

where $A'[N'+1] = (\infty, \infty, 0)$. Each part of the sum contributes the corresponding number of character matches from the beginning of the anchors to the induced common subsequence. These form the mutually covered part of the inputs; see Figure 1 for an illustration on an extension of this concept to graphs.

Consider now the symmetric chaining problem between a DAG and a string:

Problem 1 (Symmetric DAG chaining with overlaps)**.** *Find a symmetric chain $A'[1..N']$ of a set A of N anchors between a labeled DAG $G = (V, E, \ell)$ and*

[1]Anchors have the same representation as graph MEMs, $([x..y], (i, P, j))$, but they do not necessarily represent exact matches.

a query string Q maximizing the length of an induced common subsequence $C = P \mid A' = Q \mid A'$ for some path P of G, where $P \mid A'$ denotes the subsequence obtained by deleting the parts of $\ell(P)$ that are not mutually covered by chain A' and $Q \mid A'$ denotes the subsequence obtained by deleting the parts of Q not mutually covered by chain A'.

In this section, we will solve this problem in the special case where the anchors are all node MEMs between G and Q: thanks to Theorem 1 we know that the algorithm by Mäkinen et al. [16] solves the problem when a longest induced common subsequence C is covered by node MEMs that appear in different nodes. Since in our setting the overlaps can only occur inside node labels, we are left with what essentially is the symmetric string-to-string chaining problem [19, 15]. However, we cannot separate these subproblems and call the respective algorithms as black boxes, but instead we need to carefully interleave the computation of both techniques in one algorithm.

4.1 DAG chaining with node MEMs

Algorithm 1 shows the pseudocode of [16, Algorithm 1] simplified to take node MEMs as anchors. The original algorithm uses two arrays to store the start and the end nodes of anchor paths, but in the case of node MEMs one array suffices. We also modified [16, Lemma 3.2] below to explicitly use primary and secondary keys (the original algorithms [16, 15] implicitly assumed distinct keys). We still use primary keys to store MEM ending positions in Q to do range searches, and we use the secondary key to store the MEM identifiers to update the values of the corresponding anchors.

Just like the original algorithm, our simplified version fills a table $C[1..N]$ such that $C[j]$ is the maximum coverage of an asymmetric chain that uses the j-th node MEM as its last item. That is, there is an asymmetric chain that induces a subsequence Q' of the query Q of length $C[j]$. In addition, our simplified version is restricted to chains that can include at most one MEM per node and includes an intermediate step to fill table $C^-[1..N]$ such that $C^-[j] = C[j] - \kappa'$, where κ' is the length of the j-th node MEM. The reason for these modifications will become clear when we integrate the algorithm with the symmetric string-to-string chaining.

To fill tables $C[1..N]$ and $C^-[1..N]$, the algorithm considers a) MEMs from different nodes without overlap in the query and b) MEMs from different nodes with overlap in the query. These cases are illustrated in the left panel of Figure 2. The algorithm maintains the following data structure for each case and for each path in a given path cover of k paths (see e.g. [9, Chapter 5]):

Lemma 2. *The following four operations can be supported with a balanced binary search tree $\mathcal{T}$ in time $O(\log n)$, where n is the number of key-value pairs $((k, j), \texttt{val})$ stored in the tree. Here k is the primary key, j is the secondary key to break ties, and $k, j, \texttt{val}$ are integers.*

- $\mathsf{value}(k, j)$*: Return the value associated to key (k, j) or $-\infty$ if (k, j) is not a proper key.*
- $\mathsf{update}((k, j), \texttt{val})$*: Associate value $\texttt{val}$ to key (k, j).*
- $\mathsf{upgrade}((k, j), \texttt{val})$*: Associate value $\max(\texttt{val}, \mathsf{value}(k, j))$ to key (k, j).*

ALGORITHM 1: Asymmetric co-linear chaining between a sequence and a DAG using a path cover and node MEMs.

```
Input: A DAG $G = (V, E, \ell)$, a query string $Q$, a path cover
       $P_1, P_2, \ldots, P_k$ of $G$, and node MEMs $A[1..N]$ of the form
       $([x..x+\kappa'-1], (i, v, i+\kappa'-1))$, where
       $\ell(v)[i..i+\kappa'-1] = Q[x..x+\kappa'-1]$.
Output: Index of a MEM ending at a chain with maximum coverage
        $\max_j C[j]$ allowing at most one MEM per node of $G$.
1  Use Lemma 3 to find all forward propagation links;
2  for $k' \leftarrow 1$ to $k$ do
3      Initialize data structures $\mathcal{T}^a_{k'}$ and $\mathcal{T}^b_{k'}$ with keys $(x+\kappa'-1, j)$ such
         that $([x..x+\kappa'-1], (i, v, i+\kappa'-1)) = A[j]$, $1 \le j \le N$, and with
         key $(0, 0)$, all keys associated with values $-\infty$;
4      $\mathcal{T}^a_{k'}$.update$((0, 0), 0)$;
5      $\mathcal{T}^b_{k'}$.update$((0, 0), 0)$;
   /* Save to anchors[$v$] all node MEMs of node $v$.                */
6  for $j \leftarrow 1$ to $N$ do
7      $([x..x+\kappa'-1], (i, v, i+\kappa'-1)) = A[j]$;
8      anchors[$v$].push($j$);
9      $C^-[j] \leftarrow 0$;
10     $C[j] \leftarrow \kappa'$;
11 for $v \in V$ in topological order do
12     for $j \in$ anchors[$v$] do
           /* Update the data structures for every path that
              covers $v$, stored in paths[$v$].                    */
13         $([x..x+\kappa'-1], (i, v, i+\kappa'-1)) = A[j]$;
14         for $k' \in$ paths[$v$] do
15             $\mathcal{T}^a_{k'}$.upgrade$((x+\kappa'-1, j), C[j])$;
16             $\mathcal{T}^b_{k'}$.upgrade$((x+\kappa'-1, j), C^-[j] - x)$;
       /* PROPAGATE FORWARD STARTS                                  */
17     for $(w, k') \in$ forward[$v$] do
18         for $j \in$ anchors[$w$] do
19             $([x..x+\kappa'-1], (i, v, i+\kappa'-1)) = A[j]$;
20             $C^{\mathsf{a}}[j] \leftarrow \mathcal{T}^a_{k'}$.RMaxQ$(0, x-1)$;
21             $C^{\mathsf{b}}[j] \leftarrow x + \mathcal{T}^b_{k'}$.RMaxQ$(x, x+\kappa'-1)$;
22             $C^-[j] \leftarrow \max(C^-[j], C^{\mathsf{a}}[j], C^{\mathsf{b}}[j])$;
23             $C[j] = C^-[j] + \kappa'$;
       /* PROPAGATE FORWARD ENDS                                    */
24 return $\mathrm{argmax}_j\, C[j]$;
```

- $\mathsf{RMaxQ}(l, r)$: *Return* $\max_{l \leq k \leq r, (k,j) \text{ is a key in } \mathcal{T}} \texttt{value}(k, j)$ *(*Range Maximum Query*), or* $-\infty$ *if the range is empty.*

Moreover, the balanced binary search tree can be constructed in $O(n)$ *time, given the* n *pairs* $((k, j), \texttt{val})$ *sorted by component* (k, j)*.*

The algorithm processes the nodes in topological order, keeping the invariant that once node v is visited, the final values $C[j]$ and $C^{-}[j]$ are known for all anchors j included in node v. These values are then stored in the search trees. As a final step in the processing of v, the information stored in the search trees is propagated forward to nodes w, where v is the last node reaching w on some path-cover path. This propagated information is used for updating the intermediate values for MEMs at node w. These forward links are preprocessed with the following lemma:

Lemma 3 (Adaptation of [16, Lemma 3.1])**.** *Let* $G = (V, E)$ *be a DAG, and let* $P_1, \ldots, P_k$ *be a path cover of* G*. We can compute in* $O(k^2|V|)$ *time the set of* forward propagation links $\texttt{forward}[u]$ *defined as follows: for any node* v *and path* k'*,* $(v, k') \in \texttt{forward}[u]$ *if and only if* u *is the last node on path* k' *that reaches* v *such that* $u \neq v$*.*

Proof. The original DP algorithm [16] runs in $O(k|E|)$ time, but recently it has been shown [12, Algorithms 6 and 7] how to do this in time $O(k|E_{red}|)$, where E_{red} are the edges in the transitive reduction of G. Finally, Cáceres et al. [3, 4] showed a transitive sparsification scheme proving that $|E_{red}| \leq k|V|$. $\square$

Data structures $\mathcal{T}^{a}_{k'}$ store as primary keys all ending positions of MEMs in Q and as values the corresponding $C[j]$s for node MEMs $A[j]$ processed so far and reaching path $P_{k'}$ (line 15). When a new node MEM is added to a chain at line 20, the range query on $\mathcal{T}^{a}_{k'}$ guarantees that only chains ending before v in G and before the start of the new node MEM in Q are taken into account. Data structures $\mathcal{T}^{b}_{k'}$ also store as primary keys all ending positions of node MEMs in Q, but as values they store the values $C^{-}[j]$ with an invariant subtracted (line 16). This invariant is explained by the range query at line 21, that considers chains overlapping (only) in Q with the new node MEM to be added: consider the chain ending at node MEM $A[j'] = ([x'..x' + \kappa'' - 1], (i', v', i' + \kappa'' - 1))$ and the new node MEM $A[j] = ([x..x + \kappa' - 1], (i, v, i + \kappa' - 1)$ is to be added to this chain, where $x \leq x' + \kappa'' - 1 \leq x + \kappa' - 1$. This addition increases the part of Q covered by the chain (excluding the new node MEM) by $x - x'$. This is exactly the value computed at line 21, maximizing over such overlapping node MEMs.

4.2 Revisiting symmetric string-to-string chaining with MEMs

Before modifying the algorithm to properly consider overlaps of node MEMs in G, let us first modify the symmetric string-to-string chaining algorithm of Mäkinen and Sahlin [15, Algorithm 2] to harmonize the notation and to consider the simplification of [15, Theorem 6] that applies in the case of (string) MEMs. This modification computes the optimal chain given MEMs $A[1..N]$ between strings T and Q and is given as Algorithm 2.

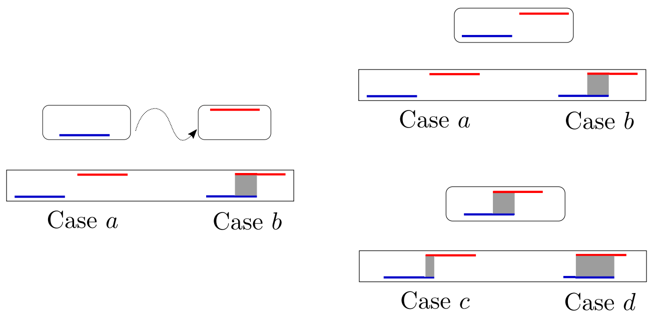


Figure 2: Precedence of MEMs partitioned to three classes (left, top right, and bottom right subfigures) by occurrence in graph/text (top part of each subfigure) and thereafter to two out of total four cases that require different data structure on the query (bottom part of each subfigure).

ALGORITHM 2: Symmetric chaining with two-sided overlaps using MEMs.

```
Input: An array $A[1..N]$ of (string) MEMs $(x, i, \kappa')$ between $Q$ and $T$.
Output: Index of a MEM ending a chain with maximum coverage $\max_j C[j]$.
Initialize data structures $\mathcal{T}^{\mathsf{a}}$ and $\mathcal{T}^{\mathsf{b}}$ with keys $(x + \kappa' - 1, j)$ and data structures $\mathcal{T}^{\mathsf{c}}$ and $\mathcal{T}^{\mathsf{d}}$ with keys $(x - i, j)$, where $(x, i, \kappa') = A[j]$, $1 \leq j \leq N$, and all trees with key $(0, 0)$. Associate values $-\infty$ to all keys.
$\mathcal{T}^{\mathsf{a}}$.upgrade$((0, 0), 0)$;
$M = \{(x, j) \mid (x, i, \kappa') = A[j], 1 \leq j \leq N\} \cup \{(x + \kappa' - 1, j) \mid (x, i, \kappa') = A[j], 1 \leq j \leq N\}$;
$M.sort()$;
for $(x', j) \in M$ do
    $(x, i, \kappa') = A[j]$;
    if $x == x'$ then
        /* Start of MEM. */
        $C^{\mathsf{a}}[j] = \mathcal{T}^{\mathsf{a}}$.RMaxQ$(0, x - 1)$;
        $C^{\mathsf{b}}[j] = x + \mathcal{T}^{\mathsf{b}}$.RMaxQ$(x, x + \kappa' - 1)$;
        $C^{\mathsf{c}}[j] = i + \mathcal{T}^{\mathsf{c}}$.RMaxQ$(-\infty, x - i)$;
        $C^{\mathsf{d}}[j] = x + \mathcal{T}^{\mathsf{d}}$.RMaxQ$(x - i + 1, \infty)$;
        $C^{-}[j] = \max(C^{\mathsf{a}}[j], C^{\mathsf{b}}[j], C^{\mathsf{c}}[j], C^{\mathsf{d}}[j])$;
        $C[j] = C^{-}[j] + \kappa'$;
        $\mathcal{T}^{\mathsf{c}}$.upgrade$((x - i, j), C^{-}[j] - i)$;
        $\mathcal{T}^{\mathsf{d}}$.upgrade$((x - i, j), C^{-}[j] - x)$;
    else
        /* End of MEM. */
        $\mathcal{T}^{\mathsf{a}}$.upgrade$((x + \kappa' - 1, j), C[j])$;
        $\mathcal{T}^{\mathsf{b}}$.upgrade$((x + \kappa' - 1, j), C^{-}[j] - x)$;
        $\mathcal{T}^{\mathsf{c}}$.update$((x - i, j), -\infty)$;
        $\mathcal{T}^{\mathsf{d}}$.update$((x - i, j), -\infty)$;
return $\text{argmax}_j\, C[j]$;
```

The algorithm uses the same two data structures as before to handle the cases illustrated at the top right of Figure 2. Moreover, the two additional data structures (balanced binary search trees) in Algorithm 2 handle the overlaps in T by dividing the computation further into cases c) and d) illustrated at the bottom right of Figure 2): c) if two MEMs overlap more in T than in Q, tree $\mathcal{T}^{\mathsf{c}}$ is used for storing the solution; d) otherwise, tree $\mathcal{T}^{\mathsf{d}}$ is used for storing the solution. We refer to the original work [15] for the derivation of the invariants and the range queries to handle these cases. The handling of these cases is highlighted with gray background in Algorithm 2.

4.3 Integration of symmetry to DAG chaining

We will now merge the two algorithms from previous subsections to solve Problem 1. This algorithm is shown as Algorithm 3; lines highlighted with a dark gray background are from Algorithm 2, whereas lines highlighted with a light gray background are a hybrid of both, and the rest are from Algorithm 1. When visiting node v the algorithm executes the steps of Algorithm 2 on anchors included in v, with $C^{\mathsf{a}}[j]$ and $C^{\mathsf{b}}[j]$ having already been updated with anchors not included in v through forward propagation identical to Algorithm 1. The hybrid parts reflect the required changes to Algorithm 1 in order to visit the MEM anchors twice as in Algorithm 2. This merge covers all three cases of Figure 2.

Theorem 2. *Given labeled DAG $G = (V, E, \ell)$ with path cover $P_1, \ldots, P_k$, query string Q, and set $A[1..N]$ of node MEMs between Q and G, Algorithm 3 solves the symmetric DAG chaining with overlaps problem (Problem 1) in time $O(k^2|V| + kN \log N)$.*

Corollary 2. *The length of a longest common subsequence (LCS) between a path in a labeled DAG $G = (V, E, \ell)$ and string Q can be computed in time $O(n+m+k^2|V|+|E|+kN \log N)$, where $m = |Q|$, n is the total length of node labels, k is the width (minimum number of paths covering the nodes) of G, and N is the number of node MEMs.*

Proof. The node MEMs can be computed in time $O(n+m+N)$ with Lemma 1. A minimum path cover with k paths can be computed in $O(k^2|V|+|E|)$ time [4, 3]. Forward propagation links can be computed in $O(k^2|V|)$ time with Lemma 3. Finally, the term $kN \log N$ comes from Theorem 2. The connection between LCS and solution to symmetric chaining follows with identical arguments as in the proof of Corollary 3 in Appendix B. If P is a path containing an LCS of length c, then Algorithm 3 finds a chain of coverage exactly c as its execution considers the corresponding chain between $\ell(P)$ and Q as done in Algorithm 2. In this case node MEMs are not necessarily MEMs between $\ell(P)$ and Q, but exact matches supporting the necessary character matches, see Appendix B. $\square$

Note that the LCS connection can be easily adapted for long MEMs spanning two or more nodes of G: Definition 1 considers all (string) MEMs between Q and $\ell(P)$, for any arbitrary path P; we did not consider symmetric chains of long MEMs due to the difficulty of handling path overlaps efficiently (see also [16]).

ALGORITHM 3: Symmetric co-linear chaining between a sequence and a DAG using a path cover and node MEMs.

```
   Input: Same as in Algorithm 1.
   Output: Index of a MEM ending at a chain with maximum coverage
           $\max_j C[j]$ allowing overlaps in $G$.
1  Use Lemma 3 to find all forward propagation links.
2  for $k' \leftarrow 1$ to $k$ do
3      Initialize data structures $\mathcal{T}^{\mathsf{a}}_{k'}$ and $\mathcal{T}^{\mathsf{b}}_{k'}$ with keys $(x+\kappa'-1, j)$ and
         key $(0,0)$, and data structures $\mathcal{T}^{\mathsf{c}}_{k'}$ and $\mathcal{T}^{\mathsf{d}}_{k'}$ with keys $(x-i, j)$,
         where $([x..x+\kappa'-1],(i,v,i+\kappa'-1)) = A[j]$, $1 \leq j \leq N$.
         Associate values $-\infty$ to all keys.
4      $\mathcal{T}^{a}_{k'}$.update$((0,0),0)$;
5      $\mathcal{T}^{b}_{k'}$.update$((0,0),0)$;
6  Initialize arrays: anchors, $C^-$ and $C$ as in Algorithm 1;
7  for $v \in V$ in topological order do
8      $M = \{(x,j) \mid ([x..x+\kappa'-1],(i,v,i+\kappa'-1)) = A[j], j \in$
         anchors$[v]\} \cup \{(x+\kappa'-1, j) \mid ([x..x+\kappa'-1],(i,v,i+\kappa'-1)) =$
         $A[j], j \in$ anchors$[v]\}$;
9      $M.sort()$;
       /* Update the data structures for every path that covers
          $v$, stored in paths$[v]$.                              */
10     for $k' \in$ paths$[v]$ do
11         for $(x',j) \in M$ do
12             $(x,i,\kappa') = A[j]$;
13             if $x == x'$ then
                   /* Start of MEM.                                  */
14                 $C^{\mathsf{a}}[j] = \mathcal{T}^{\mathsf{a}}_{k'}$.RMaxQ$(0, x-1)$;
15                 $C^{\mathsf{b}}[j] = x + \mathcal{T}^{\mathsf{b}}_{k'}$.RMaxQ$(x, x+\kappa'-1)$;
16                 $C^{\mathsf{c}}[j] = i + \mathcal{T}^{\mathsf{c}}_{k'}$.RMaxQ$(-\infty, x-i)$;
17                 $C^{\mathsf{d}}[j] = x + \mathcal{T}^{\mathsf{d}}_{k'}$.RMaxQ$(x-i+1, \infty)$;
18                 $C^-[j] = \max(C^-[j], C^{\mathsf{a}}[j], C^{\mathsf{b}}[j], C^{\mathsf{c}}[j], C^{\mathsf{d}}[j])$;
19                 $C[j] = C^-[j] + \kappa'$;
20                 $\mathcal{T}^{\mathsf{c}}_{k'}$.upgrade$((x-i, j), C^-[j]-i)$;
21                 $\mathcal{T}^{\mathsf{d}}_{k'}$.upgrade$((x-i, j), C^-[j]-x)$;
22             else
                   /* End of MEM.                                    */
23                 $\mathcal{T}^{\mathsf{a}}_{k'}$.upgrade$((x+\kappa'-1, j), C[j])$;
24                 $\mathcal{T}^{\mathsf{b}}_{k'}$.upgrade$((x+\kappa'-1, j), C^-[j]-x)$;
25                 $\mathcal{T}^{\mathsf{c}}_{k'}$.update$((x-i, j), -\infty)$;
26                 $\mathcal{T}^{\mathsf{d}}_{k'}$.update$((x-i, j), -\infty)$;
27     Execute PROPAGATE FORWARD subroutine of Algorithm 1;
28 return $\mathrm{argmax}_j\, C[j]$;
```

5 Discussion

In this paper, we focused on MEMs with no lower threshold on their length to achieve the connection with LCS. In practical applications, chaining is sped up by using as anchors only MEMs that are of length at least κ, a given threshold. Just finding all such κ-MEMs is a non-trivial problem and solvable in subquadratic time only on some specific graph classes [17]. However, once such κ-MEMs are found, one can split them to node-MEMs and then apply Algorithm 3 to chain them. The resulting chain optimizes the length $|C|$ of a longest common subsequence C between the query Q and a path P such that each match $C[k] = Q[i_k] = \ell(P)[j_k]$ is supported by an exact match of length at least κ, where $1 \leq k \leq |C|$, $i_1 < i_2 < \cdots < i_{|C|}$, and $j_1 < j_2 < \cdots < j_{|C|}$. That is, there is a κ-MEM $([x_k, y_k], [c_k, d_k])$ with respect to Q and $\ell(P)$ s.t. $x_k \leq i_k \leq y_k$ and $c_k \leq j_k \leq d_k$ for each k. Additionally, Ma et al. [14, Appendix C] showed that asymmetric co-linear chaining can be extended to graphs with cycles by considering the graph of the strongly connected components. In the extended version of this paper we will show how to combine our results to obtain symmetric chaining in general graphs.

References

[1] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Trans. Algorithms*, 16(2):17:1–17:54, 2020. `doi:10.1145/3381417`.

[2] Manuel Cáceres. Parameterized algorithms for string matching to dags: Funnels and beyond. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023, June 26-28, 2023, Marne-la-Vallée, France*, volume 259 of *LIPIcs*, pages 7:1–7:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPIcs.CPM.2023.7`.

[3] Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I. Tomescu. Minimum path cover in parameterized linear time. *CoRR*, abs/2211.09659, 2022. `arXiv:2211.09659`, `doi:10.48550/arXiv.2211.09659`.

[4] Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I. Tomescu. Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 359–376. SIAM, 2022. `doi:10.1137/1.9781611977073.18`.

[5] Ghanshyam Chandra and Chirag Jain. Sequence to graph alignment using gap-sensitive co-linear chaining. In Haixu Tang, editor, *Research in Computational Molecular Biology - 27th Annual International Conference, RECOMB 2023, Istanbul, Turkey, April 16-19, 2023, Proceedings*, volume 13976 of *Lecture Notes in Computer Science*, pages 58–73. Springer, 2023. `doi:10.1007/978-3-031-29119-7\_4`.

[6] The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 10 2016. arXiv:https://academic.oup.com/bib/article-pdf/19/1/118/25406834/bbw089.pdf, doi:10.1093/bib/bbw089.

[7] Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2022, Snowbird, UT, USA, March 22-25, 2022*, pages 272–281. IEEE, 2022. doi:10.1109/DCC52660.2022.00035.

[8] Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2585–2599. SIAM, 2021. doi:10.1137/1.9781611976465.153.

[9] Mark de Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer Science & Business Media, 2000.

[10] 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.

[11] Massimo Equi, Veli Mäkinen, Alexandru I Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Transactions on Algorithms*, 19(3):1–25, 2023.

[12] Giorgos Kritikakis and Ioannis G Tollis. Fast and practical DAG decomposition with reachability applications. *arXiv preprint arXiv:2212.03945*, 2022. To appear in the proceedings of SEA 2023.

[13] Heng Li, Xiaowen Feng, and Chong Chu. The design and construction of reference pangenome graphs with minigraph. *Genome Biology*, 21:1–19, 2020.

[14] Jun Ma, Manuel Cáceres, Leena Salmela, Veli Mäkinen, and Alexandru I. Tomescu. Chaining for accurate alignment of erroneous long reads to acyclic variation graphs. *bioRxiv*, 2022. URL: https://www.biorxiv.org/content/early/2022/05/19/2022.01.07.475257, arXiv:https://www.biorxiv.org/content/early/2022/05/19/2022.01.07.475257.full.pdf, doi:10.1101/2022.01.07.475257.

[15] 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, June 17-19, 2020, Copenhagen, Denmark, volume 161 of *LIPIcs*, pages 25:1–25:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.CPM.2020.25`.

[16] Veli Mäkinen, Alexandru I. Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse dynamic programming on DAGs with small width. *ACM Trans. Algorithms*, 15(2):29:1–29:21, 2019. `doi:10.1145/3301312`.

[17] Nicola Rizzo, Manuel Cáceres, and Veli Mäkinen. Finding maximal exact matches in graphs, 2023. To appear in the proceedings of WABI 2023. URL: `https://arxiv.org/abs/2305.09752`, `arXiv:2305.09752`.

[18] Nicola Rizzo, Alexandru I. Tomescu, and Alberto Policriti. Solving string problems on graphs using the labeled direct product. *Algorithmica*, 84(10):3008–3033, 2022. `doi:10.1007/s00453-022-00989-x`.

[19] Tetsuo Shibuya and Igor Kurochkin. Match Chaining Algorithms for cDNA Mapping. In Gary Benson and Roderic D. M. Page, editors, *Algorithms in Bioinformatics*, pages 462–475, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

A Chaining for longer MEMs

We now show that graph MEMs of Definition 1 can be captured simply by concatenating node MEMs. Given two graph substrings $(i, P = u_1..u_k, j)$ and $(i', P' = v_1..v_{k'}, j')$, they can be concatenated into $(i, P, j) \cdot (i', P', j')$ only if one of the following two conditions hold: $(u_k, v_1) \in E$, $j = \|u_k\|$, and $i' = 1$; or $u_k = v_1$ and $i' = j + 1$. In the former case, $(i, P, j) \cdot (i', P', j') \coloneqq (i, P \cdot P', j')$, whereas in the latter case $(i, P, j) \cdot (i', P', j') \coloneqq (i, u_1..u_{k-1} \cdot v_1..v_{k'}, j')$. We then say that two MEMs $([x..y], (i, P, j))$ and $([x'..y'], (i', P', j'))$ can be concatenated if (i, P, j) can be concatenated to (i', P', j') and $x' = y + 1$, and in such case we analogously define $([x..y], (i, P, j)) \cdot ([x'..y'], (i', P', j') \coloneqq ([x..y'], (i, P, j) \cdot (i', P', j')$.

Let A be the set of node MEMs found using algorithm of Section 3. We call a sequence of node MEMs $A'[1..p] \subseteq A$ a *perfect chain* if $A'[j]$ can be concatenated to $A'[j + 1]$ for $1 \leq j < p$. Note that the concatenation of all such node MEMs in the perfect chain yields a longer exact match.

Theorem 1. *For every MEM $([x..y], (i, P, j))$ between G and Q, there is a perfect chain $A'[1..p] \subseteq A$ such that $A'[1] \cdots A'[p] = ([x..y], (i, P, j))$.*

Proof. Let path P be spanning nodes v_1, v_2, …, v_p and spelling $\ell(v_1)[i..\|v_1\|]$ $\ell(v_2) \cdots \ell(v_{p-1})$ $\ell(v_p)[1..j]$. That is, there exist exact matches $([i_1..i_2 - 1], (i, v_1, \|v_1\|))$, $([i_2..i_3 - 1], (1, v_2, \|v_2\|))$, …, $([i_{p-1}..i_p - 1], (1, v_{p-1}, \|v_{p-1}\|))$, $([i_p..i_{p+1} - 1], (1, v_p, j))$ between Q and G. It is clear that if those matches are node MEMs then they form a perfect chain as they can be concatenated. Indeed, matches $([i_l..i_{l+1} - 1], (1, v_l, \|v_l\|))$ are right-maximal for $1 < l < p$ since they end at the end of a node label. For the same reason: matches $([i_l..i_{l+1} - 1], (1, v_l, \|v_l\|))$ are left-maximal for $1 < l < p$; $([i_1..i_2 - 1], (i, v_1, \|v_1\|))$ is right-maximal; $([i_p..i_{p+1} - 1], (1, v_p, j))$ is left-maximal. Finally, if we suppose by contradiction that match $([i_1..i_2 - 1], (i, v_1, \|v_1\|))$ $(([i_p..i_{p+1} - 1], (1, v_p, j)))$ can be extended to the left (right) to $([i_1 - 1..i_2 - 1], (i - 1, v_1, \|v_1\|))$ $(([i_p..i_{p+1}], (1, v_p, j + 1)))$ we contradict the maximality of $([x..y], (i, P, j))$. □

Corollary 1. *The set A is a* compact representation *of the set M of MEMs between query Q and a labeled DAG $G = (V, E, \ell)$: it holds $|A| \leq \|M\|$, where $\|M\|$ is the length of the encoding of the paths in MEMs as the explicit sequence of its nodes.*

Proof. The corollary follows from Theorem 1 and the fact that for every node MEM using node v there is at least one MEM between Q and G whose path contains v. Indeed, v can be used in multiple MEM paths. □

B Co-linear chaining on strings using MEMs gives LCS

We first prove [15, Theorem 7][2]. A string $C[1..\ell]$ is an LCS of strings Q and T if it is a longest string that can be written as $C = Q[y_1]..Q[y_\ell] = T[j_1]..T[j_\ell]$ with $1 \leq y_1 < .. < y_\ell \leq |Q|$ and $1 \leq j_1 < .. < j_\ell \leq |T|$. Given a set A of anchors

[2]We provide this proof for completeness since the original proof is incomplete as checked with co-author Mäkinen.

being exact matches between Q and T, we define an *anchor-restricted LCS* if it is a longest string such that it can be written as before but additionally for every character match $Q[y_l] = T[j_l]$ there exists an anchor $(x_l, i_l, \kappa'_l) \in A$ such that $x_l \leq y_l \leq x_l + \kappa'_l - 1$, $i_l \leq j_l \leq i_l + \kappa'_l - 1$ (the anchor supports the character match) and $y_l - x_l = j_l - i_l$ (the match occurs within the same offset in the anchor).

Theorem 3 ([15, Theorem 7])**.** *Given a set of anchors A of exact matches between two strings Q and T, the length of an anchored-restricted LCS equals the coverage of a maximum coverage chain under the co-linear chaining formulation of Mäkinen and Sahlin [15].*

Proof. The authors of [15] proved that every chain $A'[1..N']$ of anchors induces a common subsequence between Q and T whose length equals the coverage of A': each anchor $A'[l] = (x_l, i_l, \kappa'_l)$ contributes c_l characters to this subsequence such that c_l is the minimum between the characters of $[x_l...x_l + \kappa'_l - 1]$ not covered by the rest of the chain $A[l+1..N']$ and the characters of $[i_l...i_l + \kappa'_l - 1]$ not covered by the rest of the chain $A[l+1..N']$. We now prove that if c is the length of an anchored-restricted LCS, then there is a *weak chain* [15] of A with coverage c, where weak chain is such that consecutive anchors $(x_l, i_l, \kappa'_l), (x_{l+1}, i_{l+1}, \kappa'_{l+1})$ of a weak chain satisfy $x_l < x_{l+1}$ and $i_l < i_{l+1}$. Our proof technique consists in filtering out anchors supporting the LCS (while preserving the coverage of the chain) so that the final set of anchors corresponds to a weak chain.

Let $C[1..c]$ be an anchored-restricted LCS such that $C = Q[y_1]..Q[y_c] = T[j_1]..T[j_c]$ with $1 \leq y_1 < .. < y_c \leq |Q|$ and $1 \leq j_1 < .. < j_c \leq |T|$, and let (x_l, i_l, κ'_l) the anchor supporting the match $Q[y_l] = T[j_l]$ for $1 \leq l \leq c$, that is $x_l \leq y_l \leq x_l + \kappa'_l - 1$ and $i_l \leq j_l \leq i_l + \kappa'_l - 1$. We will show that we can remove anchors from the beginning of the chain so that (after the removal) $x_1 < x_2$ and $i_1 < i_2$ (the first anchor *weakly precedes* the second anchor) while maintaining the coverage. The proof follows inductively by removing the first anchor and applying the same procedure in the rest of the chain and the rest of the anchored-restricted LCS until no anchors remain. We first show that we can obtain a non-strict inequality $x_1 \leq x_2$ and then how to filter anchors when $x_1 = x_2$. The argument for i_1 and i_2 follows symmetrically.

Consider the case where $x_1 \geq x_2$, then $x_2 \leq x_1 \leq y_1 < y_2 \leq x_2 + \kappa'_2 - 1$ that is, the second anchor is also covering $Q[x_1]$. If $i_1 \geq i_2$, then $i_2 \leq i_1 \leq j_1 < j_2 \leq i_2 + \kappa'_2 - 1$, thus the second anchor is also supporting the first match $Q[i_1]$ and thus we can safely remove the first anchor without changing the coverage of the chain. Otherwise $i_1 < i_2$, and suppose that the second anchor does not cover $T[i_1]$ (if it does we can remove the first anchor as before), that is $j_1 < i_2$ (the case $j_1 > i_2 + \kappa'_2 - 1$ does not exist since $j_1 < j_2 \leq i_2 + \kappa'_2 - 1$). In this case we can replace the match $Q[y_1] = T[j_1]$ by the match $Q[x_2] = T[j_2]$, which is covered by the second anchor and thus we can safely remove the first anchor (as we have discovered another anchored-restricted LCS). Indeed, the character match $Q[x_2] = T[i_2]$ exists since the second anchor is an exact match and it can replace the match $Q[y_1] = T[j_1]$ since it does not interfere with $Q[y_2] = T[j_2]$ ($x_2 \leq x_1 \leq y_1 < y_2$, and $j_2 > i_2$ since $j_2 - i_2 = y_2 - x_2 > 0$). ☐

Corollary 3. *The chaining algorithm by Mäkinen and Sahlin [15] computes the length of an LCS between strings Q and T if it is given all (string) MEMs between Q and T as input anchors.*

Proof. It suffices to note that every character match of an LCS of length c is supported by some MEM within the same offset (as one can start the match there and extend it to the left and right character by character), and thus, by Theorem 3, the chaining algorithm by Mäkinen and Sahlin [15] finds a chain of coverage at least c. $\square$