Longest Common Substring with Gaps and Related Problems

Aranya Banerjee
Georgia Institute of Technology, Atlanta, GA, USA

Daniel Gibney[1]
University of Texas at Dallas, Richardson, TX, USA

Sharma V. Thankachan
North Carolina State University, Raleigh, NC, USA

Abstract

The longest common substring (also known as longest common factor) and longest common subsequence problems are two well-studied classical string problems. The former is solvable in optimal $\mathcal{O}(n)$ time for two strings of length m and n with $m \leq n$, and the latter is solvable in $\mathcal{O}(nm)$ time, which is conditionally optimal under the Strong Exponential Time Hypothesis. In this work, we study the problem of longest common factor with gaps, that is, finding a set of at most k matching substrings obeying precedence conditions with maximum total length. For $k = 1$, this is equivalent to the longest common factor problem, and for $k = m$, this is equivalent to the longest common subsequence problem. Our work demonstrates that, for constant k, this problem can be solved in strongly subquadratic time, i.e., $nm^{1-\Theta(1)}$. Motivated by co-linear chaining applications in Computational Biology, we further demonstrate that the longest common factor with gaps results can be extended to the case where the matches are restricted to maximal exact matches (MEMs). To further demonstrate the applicability of our techniques, we show that a similar approach can be used for a restricted version of the episode matching problem where one seeks an ordered set of at most k matches whose concatenation equals a query pattern P and the length of the substring of T containing the matches is minimized. These solutions all run in strongly subquadratic time for constant k.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern Matching, Longest Common Subsequence, Episode Matching

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.16

Funding *Sharma V. Thankachan*: Supported by the U.S. National Science Foundation (NSF) award CCF-2316691.

1 Introduction

The longest common substring, or longest common factor, problem (LCF) asks: given strings $T_1[1..n]$ and $T_2[1..m]$ with $n \geq m$, find a common substring of T_1 and T_2 of maximum length. This classical problem was heavily influential in the field of string algorithms. Disproving a conjectured $\Omega(n \log n)$ lower bound on the time complexity on LCF by Knuth, Weiner developed a linear time solution for constant-sized alphabets that was the precursor to the modern suffix tree data structure [61]. The suffix tree can now be constructed in linear time for polynomially sized integer alphabets [31] and is one of the most widely used data structures in string algorithms.

The longest common subsequence problem (LCS) instead seeks the longest common subsequence between T_1 and T_2. This problem has been highly influential as well. The textbook dynamic programming algorithm running in $\mathcal{O}(nm)$ time underwent a series of

[1] corresponding author

improvements in the 1970s, leading to an $\mathcal{O}(kn)$ time algorithm by Hirschberg [35] where k is the length of a longest common subsequence. Other parameterizations based on the number of matching pairs have also been developed [36]. Lower bounds for LCS are conditioned on the Strong Exponential Time Hypothesis (SETH) or related conjectures in circuit complexity [1, 18]. They include that there is no $\mathcal{O}(nm^{1-\varepsilon} + mn^{1-\varepsilon})$ time algorithm for any constant $\varepsilon > 0$ for the general problem conditioned on SETH [18]. In the case of LCS of length k, a work by Bringmann and Künneman ruled out an algorithm running in the time $\mathcal{O}(n + k^2)$ [19]. This result contrasts with edit distance, where when the edit distance is at most k, an $\mathcal{O}(n + k^2)$ time algorithm is possible [50].

In this work, we consider a problem that transitions between LCF and LCS with the use of a parameter k that denotes the number of matched substrings appearing in the solution. Let $T_1[1\mathinner{..}n]$ and $T_2[1\mathinner{..}m]$ be input strings, where $n \geq m$. We say a set of k matching substrings $(T_1[i_1\mathinner{..}i'_1], T_2[j_1\mathinner{..}j'_1])$, $(T_1[i_2\mathinner{..}i'_2], T_2[j_2\mathinner{..}j'_2])$, $\ldots$, $(T_1[i_k\mathinner{..}i'_k], T_2[j_k\mathinner{..}j'_k])$ is strictly ordered if $i'_h < i_{h+1}$ and $j'_h < j_{h+1}$ for $h \in [1\mathinner{..}k-1]$. The first problem we study here is formally defined as follows.

▶ **Problem 1** (Longest Common Factor with Gaps (k-LCFg)). *Given two strings $T_1[1\mathinner{..}n]$, $T_2[1\mathinner{..}m]$ $(n \geq m)$ and a number k, find a strictly ordered set of at most k matching substrings such that the total combined length of the substrings is maximized.*

Observe that k-LCFg is equivalent to LCF when $k = 1$, and is equivalent to LCS when $k = m$. This problem was first defined by Li et al., under the name Longest k Tuple Common Substring. They focused on the case of more than two strings and applications to Computational Biology [44]. Their proposed algorithm runs in time $O(kn^k)$ for two strings. This was recently improved to $\mathcal{O}(knm)$ time by Li et al. [43].

Applications in Computational Biology. Besides providing a connection between LCF and LCS, the problems considered in this work are motivated by Computational Biology, particularly the co-linear chaining problem. Co-linear chaining is a powerful heuristic for finding good sequence alignments. As a preprocessing step for alignment, co-linear chaining takes a set of matching substrings, called anchors, and determines which matches should be preserved within the alignment based on some scoring criteria. This technique, or closely related techniques, are used in long-read mappers [24, 38, 42, 54, 55, 56] and generic sequence aligners [3, 17, 40, 48, 52]. The time complexity of the co-linear chaining algorithm is typically measured in terms of the size of the input anchor set, rather than the lengths of the underlying strings.

One commonly used set of anchors for co-linear chaining is maximal exact matches (MEMs). A match $(T_1[i_h\mathinner{..}i'_h], T_2[j_h\mathinner{..}j'_h])$ is left-maximal if $T_1[i_h - 1] \neq T_2[j_h - 1]$ or $i_h = 1$ or $j_h = 1$ and right-maximal if $T_1[i'_h + 1] \neq T_2[j'_h + 1]$ or $i_h = n$ or $j_h = m$. If a match is both left and right-maximal, we say that it is a maximal exact match. The advantages of MEMs over alternative sets of anchors include that MEMs are a superset of most commonly used anchors, including maximally extended k-mers and maximal unique matches (MUMs). However, using MEMs as anchors raises the issue of either handling the entire set of MEMs or selecting some subset of MEMs. In the former case, the entire set of MEMs can be quadratic in size relative to the sequence lengths. In the latter case, there is no guarantee that this subset gives the best co-linear chaining score relative to the solution obtained using the entire set of MEMs. By using parameter k, we bound the number of MEMs appearing in the chaining solution. This enables us to avoid the computation of the entire set of MEMs, preventing quadratic time complexity in the worst case for constant k. We consider the following problem.

▶ **Problem 2** (k-MEM Chain)**.** *Given two strings $T_1[1\,..\,n]$, $T_2[1\,..\,m]$ ($n \geq m$) and a number k, find a strictly ordered set of at most k maximal exact matches such that the total combined length is maximized.*

The condition that the subset of MEMs have maximum total length is equivalent to maximizing the "coverage" of the selected subset of anchors used in earlier co-linear chaining work. It should be noted, however, that Problem 2 is distinct from previous co-linear chaining formulations in that the input is the two strings rather than a set of anchors.

We also consider the case where a subset of MEMs is provided as input, and one wishes to find an optimal MEM chain with at most k of the provided MEMs. We assume each MEM in the input is provided as two starting positions and a length. The problem is then defined as follows.

▶ **Problem 3** (k-MEM Chain Given MEM Set)**.** *Given a subset of MEMs, $\mathcal{M}$, of the two (not necessarily provided) strings T_1, T_2 and a number k, find a strictly ordered subset of at most k MEMs from $\mathcal{M}$ such that the total combined length of the subset is maximized.*

Finally, to further demonstrate the usefulness of the underlying techniques, we also consider the problem of finding occurrences of a pattern with gaps so that the overall length of the occurrence is minimized. The problem is defined formally as follows.

▶ **Problem 4** (Pattern Matching with Gaps)**.** *Given a text $T[1\,..\,n]$ and pattern $P[1\,..\,m]$ and parameter k, we seek substrings (if they exist) $T[i_1\,..\,i'_1]$, $T[i_2\,..\,i'_2]$, …, $T[i_h\,..\,i'_h]$ such that $i'_1 < i_2$, …, $i'_{h-1} < i_h$, $T[i_1\,..\,i'_1] \cdot T[i_2\,..\,i'_2] \cdot \ldots \cdot T[i_h\,..\,i'_h] = P$, $h \leq k$, and $i'_h - i_1$ is minimized.*

Problem 4 is known as episode matching [29] when $k = m$, discussed more in Section 1.2.

1.1 Our Results

We observe in Sections 3, 4, 5 that Problems 1, 2, and 4, respectively, can be solved in $\mathcal{O}(knm)$ time using simple dynamic programming. We then give results for constant k with strongly subquadratic time complexity, stated below.

▶ **Theorem 1.** *For Problems 1, 2, and 4 with $2 \leq k = \Theta(1)$, there exists an $\widetilde{\mathcal{O}}(nm^{1-(1/3)^{k-2}})$ time[2] algorithm.*

We achieve our results through a combination of dynamic programming, string algorithm techniques, multiple levels of blocking schemes, and observations limiting which matches need to be considered as the leftmost (or rightmost) match for given suffixes of T_1 and T_2. We believe the techniques presented here can provide a starting point for the development of faster algorithms and inspire interest in how the computational complexity of this problem transitions with the parameter k.

We also present a conditional lower bound on Problem 3 where a subset of MEMs, $\mathcal{M}$, is explicitly given (moved to full version for space considerations). Letting $M = |\mathcal{M}|$, we observe that an $\mathcal{O}(kM \log M)$ time algorithm is straightforward using dynamic programming. Our lower bound is within logarithmic factors of this time complexity for large k. In the $(\max, +)$-Convolution Problem, one is given an integer N and three sequences $x_0, \ldots, x_N$, $y_0, \ldots, y_N$, and $z_0, \ldots, z_N$, and wishes to compute $\max_{i+j+h=N}\{x_i + y_j + z_h\}$. The problem has been used previously to prove conditional hardness results for Capacitated Dynamic Programming [9] and the Tree Sparisty Problem [11].

[2] $\widetilde{\mathcal{O}}(\cdot)$ suppresses polylogarithmic factors.

▶ **Theorem 2.** *For $k = \Omega(M)$, conditioned on there being no strongly subquadratic time algorithm for the* $(\max, +)$*-Convolution Problem, Problem 3 cannot be solved in $\mathcal{O}(k^{1-\varepsilon}|\mathcal{M}|)$ time for any constant $\varepsilon > 0$.*

1.2 Related Work

Variants of Longest Common Factor. A previously studied LCF variant is finding the longest common factor with at most k mismatches. Babenko and Starikovskaya considered the case with 1 mismatch and provided an $\mathcal{O}(nm)$ time algorithm [10]. Flouri et al. later provided a quadratic time algorithm for arbitrary k [32]. Kociumaka et al. proved that, for $k = \omega(\log n)$, a strongly subquadratic time algorithm is not possible under SETH and considered the approximate version [39]. Thankachan et al. provided an $\mathcal{O}(n \log^k n)$ time solution [6, 60]. This was later improved to $\mathcal{O}(n \log^{k-1/2} n)$ time for $k > 0$ by Charalampopoulos et al. [26]. The same work presents a sublinear time algorithm for the standard/exact LCF problem when the alphabet size is bounded. When the LCF has length ℓ, Charalampopoulos et al. showed the problem can be solved in time $\mathcal{O}(n + n \log^{k+1} n/\sqrt{\ell})$ [25]. Also see [2, 59] for some related results on the k-edits version of LCF. Our k-LCFg problem allows for gaps of arbitrary length. In this way, it seems distinct from k-mismatch and k-edits LCF.

Episode Matching. A closely related problem is known as *episode matching.* In episode matching, one tries to find the shortest substring of T containing P as a subsequence. As such, episode matching differs from Problem 4 only in our parameter k that limits the maximum number of substrings used. Episode matching was introduced by Das et al. [29] and studied extensively since [8, 16, 23, 28, 46]. The fastest solution currently known was given by Das et al. and runs in time $\mathcal{O}(nm/\log m)$ [29]. A lower bound proving that under SETH no $\mathcal{O}((nm)^{1-\varepsilon})$ time algorithm exists for constant $\varepsilon > 0$ was provided by Bille et al. [13]. They further provided upper and lower bounds parameterized by the length of the pattern. This parameterization differs from the number of matches in the solution used here.

Gapped pattern matching is another highly related and well-studied problem. Here, the pattern is provided along with a set of length constrained gap positions within the pattern. Practical algorithms for this problem were considered by Bader et al. [12] and indexing versions of the problem were consider by Cáceres et al. [22]. See also [14, 15, 41, 49, 53].

Co-Linear Chaining. Many variants of the co-linear chaining problem exist. These depend on the cost function used for chaining and the types of chaining allowed. Often, the cost function considers both the coverage of the chain as well as some gap cost, which is not considered here. For co-linear chaining where the anchors in the output must be non-overlapping, several solutions exist running in $\mathcal{O}(r \log r)$ time where r is the number of input anchors (not necessarily MEMs) [4, 5, 51]. When overlap is allowed, but no gap cost is incorporated, Mäkinen and Sahlin provide a solution running in $\mathcal{O}(r \log^2 r)$ time [47] that builds on work by Shibuya and Kurochkin [57]. When anchors are allowed to overlap, and gap cost is incorporated, a work by Jain et al. provides a solution running in $\widetilde{\mathcal{O}}(r)$ time, with polylogarithmic factors depending on problem specifics [37].

2 Technical Preliminaries

Suffix Arrays, Suffix Trees, and the Burrows Wheeler Transform. We assume the input strings are over an integer alphabet $\{0, 1, \ldots, \sigma - 1\}$. The suffix array $\mathrm{SA}[1\,..\,n]$ of a string $T[1\,..\,n]$ is an array such that $T[\mathrm{SA}[i]\,..]$ is the ith smallest suffix lexicographically. The inverse suffix array $\mathrm{ISA}[1..n]$ is an array such that $\mathrm{ISA}[\mathrm{SA}[i]] = i$, or equivalently, $\mathrm{ISA}[i]$ gives the lexicographic rank of $T[i..]$. The Burrows-Wheeler Transform (BWT) [21] of T is a permutation of the symbols of T such that $\mathrm{BWT}[i] = T[\mathrm{SA}[i]-1]$ if $\mathrm{SA}[i] \neq 1$ and $T[n]$ otherwise. The suffix tree [61] of T is a compact trie constructed from all suffixes of T. We assume that the last symbol in T is unique so that there are n leaves of the suffix tree $\ell_1, \ell_2, \ldots, \ell_n$, ordered such that the root-to-leaf path for a leaf ℓ_i corresponds to suffix $T[\mathrm{SA}[i]..]$. For a node v in the suffix tree, its string depth equals the length of concatenated edge labels on the root-to-v path. The longest common extension of two suffixes of a string is defined as the length of their longest common prefix, which can be computed in constant time using suffix trees [34]. The suffix tree and the BWT can be built in linear time, even for polynomially sized alphabets [31]. For our input strings T_1 and T_2, we use $\mathrm{LCE}(i, j)$ to denote the length of the longest common prefix of $T_1[i..]$ and $T_2[j..]$. We use $\overline{\mathrm{LCE}}(i, j)$ to denote the length of the longest common suffix of $T_1[1..i]$ and $T_2[1..j]$.

d-Covers. A *d-cover* of an integer interval $[1..n]$ is a subset $\mathcal{D} \subseteq [1..n]$ such that for any $i, j \in [1..n-d]$ there exists $d' \in [0..d-1]$ such that $i + d', j + d' \in \mathcal{D}$. Observe that, given a d-cover for $[1..n]$, for any $i_1, i_2 \in [1..n-\ell]$ and substrings $T[i_1..i_1+\ell)$, $T[i_2..i_2+\ell)$ of $T[1..n]$ with $\ell \geq d$, there exist $x, y \in \mathcal{D}$ such that $0 \leq x - i_1 = y - i_2 < d$. If $T[i_1..i_1+\ell) = T[i_2..i_2+\ell)$, we say that x and y *anchor* the match $T[i_1..i_1+\ell) = T[i_2..i_2+\ell)$. There exists a d-cover of size $\mathcal{O}(n/\sqrt{d})$ that can be computed in $\mathcal{O}(n/\sqrt{d})$ time [33]. The construction of d-covers is based on difference covers [27, 45], and the use of d-covers for string problems was first proposed by Burkhardt and Kärkkäinen [20] and further used for LCE queries by Gawrychowski et al. [33].

Range Minimum/Maximum Queries. Our algorithms will use orthogonal range minimum queries (RmQs) and orthogonal range maximum queries (RMQs). For a set of n d-dimensional points, each with an associated weight, a "query" consists of an orthogonal d-dimensional range. The query response to an RmQ query is a point in that range with the smallest weight. The query response to an RMQ query is a point in that range with the largest weight. Using known techniques in computational geometry [30], d-dimensional RmQ and RMQ data structures answering queries in $\mathcal{O}(\log^{d-1} n)$ time can be built in $\mathcal{O}(n \log^{d-1} n)$ time and space.

3 Our Algorithm for k-LCFg

3.1 Algorithm Preliminaries

▶ **Lemma 3.** *There exists an optimal solution to any k-LCFg instance where all matches in the solution are right-maximal.*

Proof. We begin with an optimal k-LCFg solution and suppose some match in this solution is not right-maximal. Extending this match to the right by one symbol increases the length of this match by one and must decrease the length of the next consecutive match in the solution by one (otherwise, the starting solution was sub-optimal). Applying the same step repeatedly, we extend this match to the right until it is right-maximal while preserving the overall solution length. Repeating this for each non-right-maximal match, we eventually obtain a solution containing only right-maximal matches. ◀

Simple Dynamic Programming Solution. Before presenting a strongly subquadratic time algorithm for constant k, we observe that the problem can be solved in $\mathcal{O}(knm)$ time through straightforward dynamic programming. Let $\mathsf{CHAIN}(i, j, k)$ denote the length of an optimal k-LCFg solution for the suffixes $T_1[i\,..]$ and $T_2[j\,..]$. Our final answer, $\mathsf{CHAIN}(1, 1, k)$ can be computed using the following recurrence: $\mathsf{CHAIN}(i, j, k) = 0$ if $i > n$ or $j > m$ or $k = 0$, otherwise

$$\mathsf{CHAIN}(i, j, k) = \max\{\mathsf{CHAIN}(i, j + 1, k), \mathsf{CHAIN}(i + 1, j, k), Z\}$$

where $Z = \mathrm{LCE}(i, j) + \mathsf{CHAIN}(i + \mathrm{LCE}(i, j), j + \mathrm{LCE}(i, j), k - 1)$. The definition of Z works since by Lemma 3, we can always assume a match starting at i and j is right-maximal.

The dynamic programming solution provides a starting point for our improved solution. Our approach will utilize a k-stage approach that combines the observation that within a given suffix we only need the leftmost occurrence of a match, d-covers to handle large matches, and RMQ data structures to quickly recover pre-computed solutions.

Pairs of Interest. As mentioned above, we would like a way to consider the leftmost occurrences of matches within a specified subset of suffixes $\mathcal{S}_1$ of T_1 and $\mathcal{S}_2$ of T_2. This is captured formally by the following definition.

▶ **Definition 4** (Leftmost Pairs of Interest). *Let $\mathcal{S}_1 \subseteq [1\,..\,n]$ be a set of starting positions in T_1 and $\mathcal{S}_2 \subseteq [1\,..\,m]$ be a set of starting positions in T_2. A pair of indices (i, j) is called a leftmost pair of interest of $\mathcal{S}_1$ and $\mathcal{S}_2$ if $i \in \mathcal{S}_1, j \in \mathcal{S}_2$, and there does not exist an:*

- *$h \in \mathcal{S}_1$ such that $h < i$ and $\mathrm{LCE}(h, j) \geq \mathrm{LCE}(i, j)$, or an*
- *$h \in \mathcal{S}_2$ such that $h < j$ and $\mathrm{LCE}(i, h) \geq \mathrm{LCE}(i, j)$.*

Given the suffix tree of $T_1\$_1T_2\$_2$ and a set of suffix starting positions $\mathcal{S}_1$ of T_1 and $\mathcal{S}_2$ of T_2, one can compute a set of size $\mathcal{O}(|\mathcal{S}_1| + |\mathcal{S}_2|)$ containing all leftmost pairs of interest in $\mathcal{O}(|\mathcal{S}_1| + |\mathcal{S}_2|)$ time. Pairs of interest will be useful to us due to the following observation, which will be used implicitly throughout our algorithm. We call the first match occurring in a k-LCFg solution its leftmost match.

▶ **Observation 5.** *Consider an optimal k-LCFg solution for a suffix of T_1 containing $\mathcal{S}_1$ and a suffix of T_2 containing $\mathcal{S}_2$. If there exists an optimal solution having a leftmost match starting in $\mathcal{S}_1$ and $\mathcal{S}_2$, then there also exist $i \in \mathcal{S}_1$ and $j \in \mathcal{S}_2$ that are a leftmost pair of interest and the starting positions of a leftmost match in some optimal solution.*

Suffix-LCF Queries. Amir et al. presented a technique to preprocess two strings so that when two indices indicating the suffixes of the input strings are provided as a query, one efficiently obtains the longest common factor of the specified suffixes [7]. We call these suffix-LCF queries. Their data structure requires $\mathcal{O}(n \log^2 n)$ preprocessing time, $\mathcal{O}(n \log n)$ preprocessing space, and answers queries $\mathcal{O}(\log n)$ time.

Observe that for 2-LCFg, we can iterate over all leftmost pairs of interests (i, j) of $\mathcal{S}_1 = [1\,..\,n]$ and $\mathcal{S}_2 = [1\,..\,m]$, compute $\mathrm{LCE}(i, j)$ and perform a suffix-LCF query on the suffixes $T_1[i + \mathrm{LCE}(i, j)\,..]$ and $T_2[j + \mathrm{LCE}(i, j)\,..]$. The pair of matches with maximum overall length provides the optimal solution. The algorithm requires $\widetilde{\mathcal{O}}(n)$ time. Our main algorithm will be for the case $k \geq 3$.

3.2 Main Algorithm

We construct two suffix trees at the beginning of our algorithm: the suffix tree of $T_1\$_1T_2\$_2$ and the suffix tree of its reverse, where $\$_1$ and $\$_2$ are two special symbols that do not appear in T_1 or T_2. These suffix trees allow us to answer $\mathrm{LCE}(\cdot,\cdot)$ and $\overline{\mathrm{LCE}}(\cdot,\cdot)$ queries in $\mathcal{O}(1)$ time.

We initially consider a stage for each $f \in [1\mathinner{.\,.}k-1]$. The final stage, where $f = k$, is handled separately since it can be accomplished by considering the leftmost pairs of interest across all of T_1 and T_2. For each $f \in [1\mathinner{.\,.}k-1]$, we construct a data structure $\mathsf{DS}(f)$ that can efficiently compute $\mathsf{CHAIN}(i,j,f)$. We denote the size of $\mathsf{DS}(f)$ by $S(f)$, construction time by $C(f)$, and time for answering a query by $Q(f)$. Note that these are functions over f, n, and m, but we drop n and m for simplicity. For each level f, we define two parameters: b_f (called the block size) and $d_f = \sqrt{b_f}$ (called the length threshold). Although we wait to define b_f until the analysis of the algorithm, we note here that the block size b_f will increase as f increases.

For the base case $f = 1$, we make $\mathsf{DS}(1)$ equal to the suffix-LCF data structure of Amir et al. [7]. This has the properties that $S(1) = \widetilde{\mathcal{O}}(n), C(1) = \widetilde{\mathcal{O}}(n), Q(1) = \widetilde{\mathcal{O}}(1)$. We now construct a $\mathsf{DS}(f)$ for each $f \in [2\mathinner{.\,.}k-1]$ such that

$$\begin{aligned}
S(f) &= S(f-1) + \mathcal{O}(nm/b_f)\\
C(f) &= C(f-1) + Q(f-1)\cdot\mathcal{O}(nm/\sqrt{b_f})\\
Q(f) &= Q(f-1)\cdot\mathcal{O}(b_f\log b_f)
\end{aligned}$$

3.2.1 The Structure

For $f > 1$, $\mathsf{DS}(f)$ consists of $\mathsf{DS}(f-1)$ and $\mathsf{CHAIN}(i,j,f)$ for all $i \in [1\mathinner{.\,.}n]$ and $j \in [1\mathinner{.\,.}m]$, such that either i or j is a multiple of b_f. The number of such entries is bounded by $2nm/b_f$. Summing the size of $\mathsf{DS}(f-1)$ and the number of stored entries, the space $S(f)$ is $S(f-1) + \mathcal{O}(nm/b_f)$.

3.2.2 Construction

We will compute two quantities $\mathsf{CHAIN}_s(i,j,f)$ and $\mathsf{CHAIN}_l(i,j,f)$. $\mathsf{CHAIN}_s(i,j,f)$ is the length of an optimal f-LCFg solution of $T_1[i..]$ and $T_2[j..]$, such that the length of the first match in the solution is $\leq d_f$. $\mathsf{CHAIN}_l(i,j,f)$ is the maximum over the lengths of a set of potential f-LCFg solutions of $T_1[i..]$ and $T_2[j..]$, where this set includes all solutions such that the length of the first match is $> d_f$. As a result, $\mathsf{CHAIN}(i,j,f)$ is the maximum of $\mathsf{CHAIN}_s(i,j,f)$ and $\mathsf{CHAIN}_l(i,j,f)$.

Computing CHAIN_s. For each $j \in [1\mathinner{.\,.}m]$ such that j is a multiple of b_f, we compute $\mathsf{CHAIN}_s(i,j,f)$ as follows. We first initialize $\mathsf{CHAIN}_s(n+1,j,f) = 0$. Then, for i from n down to 1 and for all $h \in [1\mathinner{.\,.}d_f]$, we locate the leftmost right-maximal occurrence of $T_1[i\mathinner{.\,.}i+h)$ in $T_2[j\mathinner{.\,.}m]$ (if it exists). For each h, this can be accomplished in $\widetilde{\mathcal{O}}(1)$ time after $\widetilde{\mathcal{O}}(n)$ time preprocessing at the start of the algorithm (see full version for details). Suppose this match is $(T_1[i\mathinner{.\,.}i+h), T_2[y_h\mathinner{.\,.}y_h+h))$ for some $y_h \in [j\mathinner{.\,.}m]$. Applying the data structure for $f-1$, we let $w_h = h + \mathsf{CHAIN}(i+h, y_h+h, f-1)$ if a match for a given $h \in [1\mathinner{.\,.}d_f]$ exists and 0 otherwise. We then assign

$$\mathsf{CHAIN}_s(i,j,f) = \max_{1\leq h\leq d_f}\{\mathsf{CHAIN}_s(i+1,j,f), w_h\}$$

We perform a symmetric procedure for $i \in [1 \mathinner{.\,.} n]$ such that i is a multiple of b_f, for j from m down to 1, and $h \in [1 \mathinner{.\,.} d_f]$. Finding all desired $\mathsf{CHAIN}_s(\cdot,\cdot,f)$ values takes $2nmd_f/b_f = \mathcal{O}(nm/\sqrt{b_f})$ number of $\mathsf{CHAIN}(\cdot,\cdot,f-1)$ queries to $\mathsf{DS}(f-1)$.

Computing CHAIN$_l$. Before computing CHAIN_l for any particular i and j, we use a d_f-cover $\mathcal{D}$ of $T_1\$_1T_2\$_2$ to preprocess all MEMs that are longer than d_f. We call these *long MEMs.* We do so by considering all anchor pairs $a_1 \in \mathcal{D} \cap [1 \mathinner{.\,.} n], a_2 \in \mathcal{D} \cap [n+2 \mathinner{.\,.} n+m+1]$. Note that the number of elements of $\mathcal{D}$ in the range $[1 \mathinner{.\,.} n]$ is $\mathcal{O}(n/\sqrt{d_f})$ and the number of elements in $\mathcal{D}$ contained in the range $[n+2 \mathinner{.\,.} n+m+1]$ is $\mathcal{O}(m/\sqrt{d_f})$. This follows from the fact that the d_f-cover utilized by Colbourn and Ling is periodic in d_f [27].

Letting $a_2' = a_2 - (n+1)$, we compute $\mathrm{LCE}(a_1, a_2')$ and $\overline{\mathrm{LCE}}(a_1, a_2')$ to obtain a MEM for anchor pair a_1 and a_2' (if it exists). The total number of long MEMs is at most $(n/\sqrt{d_f}) \cdot (m/\sqrt{d_f}) = \mathcal{O}(nm/d_f) = \mathcal{O}(nm/\sqrt{b_f})$. We construct three RMQ structures over this set of long MEMs. These account for how a given i and j can intersect (or not intersect) a long MEM. To motivate the definitions, consider for indices i and j, the substring $T_1[x \mathinner{.\,.} x+h)$ of suffix $T_1[i \mathinner{.\,.} n]$ and the substring $T_2[y \mathinner{.\,.} y+h)$ of suffix $T_2[j \mathinner{.\,.} m]$ to be the leftmost match in a solution to $\mathsf{CHAIN}(i,j,f)$.

1. If $i < x$ and $j < y$ the solution value will be
$$h + \mathsf{CHAIN}(x+h, y+h, f-1)$$
2. If $y \le j < y+h$, and $i - x \le j - y$ the solution value will be
$$y + h - j + \mathsf{CHAIN}(x+h, y+h, f-1)$$
3. If $x \le i < x+h$, and $i - x > j - y$ the solution value will be
$$x + h - i + \mathsf{CHAIN}(x+h, y+h, f-1)$$

Based on the above, for a given long MEM $(T_1[x \mathinner{.\,.} x+h), T_2[y \mathinner{.\,.} y+h))$, we create the three points and weights described next. Each type of point will be stored in an RMQ structure, say RMQ_1, RMQ_2, and RMQ_3, respectively.

1. (x, y) with weight $h + \mathsf{CHAIN}(x+h, y+h, f-1)$
2. $(y, y+h, x-y)$ with weight $y + h + \mathsf{CHAIN}(x+h, y+h, f-1)$
3. $(x, x+h, x-y)$ with weight $x + h + \mathsf{CHAIN}(x+h, y+h, f-1)$

To compute $\mathsf{CHAIN}_l(i,j,f)$, we make the following queries:

1. $\mathrm{RMQ}_1([i+1 \mathinner{.\,.} n] \times [j+1 \mathinner{.\,.} m])$.
2. $\mathrm{RMQ}_2([1 \mathinner{.\,.} j] \times [j+1 \mathinner{.\,.} m] \times [i-j \mathinner{.\,.} \infty])$.
3. $\mathrm{RMQ}_3([1 \mathinner{.\,.} i] \times [i+1 \mathinner{.\,.} n] \times [-\infty \mathinner{.\,.} i-j-1])$.

Let w_g for $g \in \{1,2,3\}$ be the weight of the point returned by RMQ_g or 0 if no point is returned. Then we let $\mathsf{CHAIN}_l(i,j,f) = \max\{w_1, w_2 - j, w_3 - i\}$. Note that it is possible that the first match in the solution is actually smaller than d_f, particularly in the cases where the query result from RMQ_2 or RMQ_3 is used. Importantly, however, all solutions where the first match has a length greater than d_f are captured.

Since the preprocessing of long MEMs as described above is done only once for level f, finding all desired $\mathsf{CHAIN}_l(\cdot,\cdot,f)$ values requires $\mathcal{O}(nm/\sqrt{b_f})$ number of $\mathsf{CHAIN}(\cdot,\cdot,f-1)$ queries to $\mathsf{DS}(f-1)$. Therefore, the total time for construction after $\mathsf{DS}(f-1)$ has been computed is $Q(f-1) \cdot \mathcal{O}(nm/\sqrt{b_f})$ as claimed.

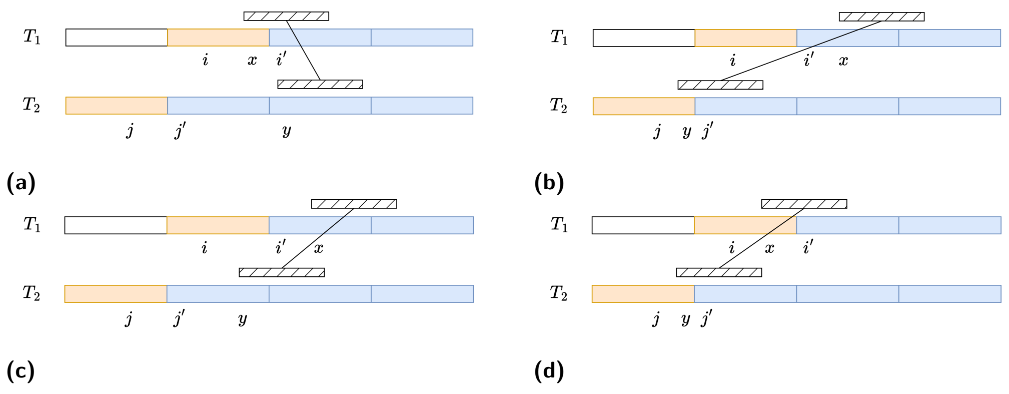

Figure 1 The cases considered when querying $\mathsf{DS}(f)$. The possible first match is an optimal solution for $T_1[i\,..]$ and $T_2[j\,..]$ shown with the dashed rectangles.

3.2.3 Querying

To find $\mathsf{CHAIN}(i, j, f)$ for an arbitrary $i \in [1\,..\,n]$ and $j \in [1\,..\,m]$ we let $i' = b_f\lceil i/b_f\rceil$ and $j' = b_f\lceil j/b_f\rceil$. Suppose $(T_1[x\,..\,x+h), T_2[y\,..\,y+h))$ is the first match in an optimal f-LCFg solution for $T_1[i\,..]$ and $T_2[j\,..]$. There are two possibilities:

1. **$x \geq i'$ or $y \geq j'$:** In this case, the answer is the max of $\mathsf{CHAIN}(i, j', f)$ and $\mathsf{CHAIN}(i', j, f)$, and can be reported in $\widetilde{\mathcal{O}}(1)$ time from the precomputed values (or a suffix-LCF query if $f = 1$). In particular, in the case shown in Figure 1a, the solution is obtained with the query $\mathsf{CHAIN}(i, j', f)$. In the case shown in Figure 1b, the solution is obtained with query $\mathsf{CHAIN}(i', j, f)$. The case in Figure 1c is solved by both queries.
2. **$x \in [i\,..\,i')$ and $y \in [j\,..\,j')$:** See Figure 1d. In this case, we compute the leftmost pairs of interests for the suffix sets $\mathcal{S}_1 = [i\,..\,i')$ and $\mathcal{S}_2 = [j\,..\,j')$. Note that $|\mathcal{S}_1| + |\mathcal{S}_2| \leq 2b_f$. We enumerate the right-maximal matches corresponding to the $\mathcal{O}(b_f)$ leftmost pairs of interest. For a match $(T_1[x'\,..\,x'+h), T_2[y'\,..\,y'+h))$ corresponding to a pair of interest, we associate the value $h + \mathsf{CHAIN}(x'+h, y'+h, f-1)$. We take the maximum value computed over all of these. If there are no pairs of interest, we consider this as returning the value 0.

We then take the maximum over the values computed for the two possibilities above. The first computation takes $\widetilde{\mathcal{O}}(1)$ time, the second computation takes $\widetilde{\mathcal{O}}(b_f) + Q(f-1) \cdot \mathcal{O}(b_f \log b_f)$ time. Combined, the total query time is $Q(f-1) \cdot \widetilde{\mathcal{O}}(b_f)$.

3.2.4 Solving the Recurrence

Assuming constant $k \geq 3$, $Q(k-1) = \widetilde{\mathcal{O}}(b_2 \cdot \ldots \cdot b_{k-1})$ and $C(k-1) = \widetilde{\mathcal{O}}\left(nm \sum_{f=2}^{k-1} \frac{b_2 \cdot \ldots \cdot b_{f-1}}{\sqrt{b_f}}\right)$ where we take $b_2 \ldots b_{f-1} = 1$ if $f = 2$. We focus on minimizing $C(k-1)$.

For $f \in [2, k-1]$, let us suppose $b_f = \lceil m^{c3^f}\rceil$ for some constant $c > 0$ (we will fix c later). Then, the largest exponent in the product $b_2 \ldots b_{f-1}/\sqrt{b_f}$ is

$$c(3^2 + \ldots + 3^{f-1}) - \frac{c \cdot 3^f}{2} = c \cdot 9(1 + \ldots + 3^{f-3}) - \frac{c \cdot 3^f}{2} = \frac{c \cdot 9(3^{f-2} - 1)}{2} - \frac{c \cdot 3^f}{2} = \frac{-9c}{2}.$$

Hence, we have $C(k-1) = \widetilde{\mathcal{O}}(nm^{1-9c/2})$ and $Q(k-1) = \widetilde{\mathcal{O}}(m^{c(3^2+\ldots+3^{k-1})}) = \widetilde{\mathcal{O}}(m^{c(3^k-9)/2})$.

3.2.5 Obtaining the Final Answer

The optimal solution value, i.e., $\mathsf{CHAIN}(1, 1, k)$ can be answered by first constructing $\mathsf{DS}(k-1)$ and then performing $\mathcal{O}(n)$ number of $\mathsf{CHAIN}(\cdot, \cdot, k-1)$ queries. In particular, we iterate over all leftmost pairs of interest for the entire set of suffixes, $\mathcal{S}_1 = [1\,..\,n]$ and $\mathcal{S}_2 = [1\,..\,m]$ as follows: For a right-maximal match $(T[x\,..\,x+h), T_2[y\,..\,y+h))$ corresponding to a pair of interest, we make the query $\mathsf{CHAIN}(x+h, y+h, k-1)$. We take $\mathsf{CHAIN}(1, 1, k)$ to be $h + \mathsf{CHAIN}(x+h, y+h, k-1)$ from the pair of interest that maximizes this value.

This makes the total time $C(k-1)+Q(k-1)\cdot\widetilde{\mathcal{O}}(n)$, which is polylogarithmic factors from $nm^{1-9c/2} + nm^{c(3^k-9)/2}$. By equating the exponents, we see that we should set $c = 2/3^k$, yielding a final complexity of $\widetilde{\mathcal{O}}(nm^{1-1/3^{k-2}})$. We discuss in the full version of this work how back pointers can be used to recover the matches in the solution.

4 Our Algorithm for k-MEM Chain

We modify the definition of $\mathsf{CHAIN}(i, j, k)$ to now be the longest k-MEM Chain solution for the suffixes $T_1[i\,..]$ and $T_2[j\,..]$. An $\mathcal{O}(knm)$ time algorithm is straightforward with the following modification to the recurrence at the start of Section 3: $\mathsf{CHAIN}(i, j, k) = 0$ if $i > n$ or $j > m$ or $k = 0$, otherwise

$$\mathsf{CHAIN}(i, j, k) = \max\{\mathsf{CHAIN}(i, j+1, k), \mathsf{CHAIN}(i+1, j, k), Z\}$$

where $Z = 0$ if $T_1[i-1] = T_2[j-1]$, else $Z = \mathrm{LCE}(i, j)+\mathsf{CHAIN}(i+\mathrm{LCE}(i, j), j+\mathrm{LCE}(i, j), k-1)$. The case where $Z = 0$ is included to avoid including non-left-maximal matches. Solving the recurrence with standard dynamic programming has a time complexity $\mathcal{O}(knm)$. In what follows, we show the modifications needed to the main algorithm in Section 3 to solve k-MEM Chain in strongly subquadratic time for constant k.

4.1 Modified Pairs of Interest

We first modify the pairs of interest definition for MEMs.

▶ **Definition 6** (Modified Leftmost (Rightmost) Pairs of Interest). *Let $\mathcal{S}_1 \subseteq [1\,..\,n]$ be a set of starting positions in T_1 and $\mathcal{S}_2 \subseteq [1\,..\,m]$ be a set of starting positions in T_2. A pair of indices (i, j) is called a leftmost (rightmost resp.) pair of interest of $\mathcal{S}_1$ and $\mathcal{S}_2$ if $i \in \mathcal{S}_1, j \in \mathcal{S}_2$, and there does not exist an:*

- *$h \in \mathcal{S}_1$ such that $h < i$ ($h > i$ resp.), $T_1[h-1] \neq T_2[j-1]$ or $h = 1$ or $j = 1$, and $\mathrm{LCE}(h, j) \geq \mathrm{LCE}(i, j)$; or an*
- *$h \in \mathcal{S}_2$ such that $h < j$ ($h > j$ resp.), $T_1[i-1] \neq T_2[h-1]$ or $i = 1$ or $h = 1$, and $\mathrm{LCE}(i, h) \geq \mathrm{LCE}(i, j)$.*

We next show how heavy path decomposition can be used to compute a set of size $\mathcal{O}((|\mathcal{S}_1| + |\mathcal{S}_2|) \log(|\mathcal{S}_1| + |\mathcal{S}_2|))$ containing all modified leftmost (rightmost) pairs of interest in $\widetilde{\mathcal{O}}(|\mathcal{S}_1| + |\mathcal{S}_2|)$ time. These are returned in the form of 2D-points with weights such that point (x, y) has x value equal to the starting position of the MEM in T_1, y value equal to the starting position of the MEM in T_2, and weight equal to the MEM's length.

The heavy path decomposition of a tree categorizes nodes into heavy and light nodes, such that the root is always light and exactly one child of every node is heavy, specifically the child with the heaviest subtree size (number of leaves under it) with ties broken arbitrarily. The technique was introduced by Sleator and Tarjan [58]. A well-known key property of heavy path decomposition used by our algorithm for k-MEM Chain is the following:

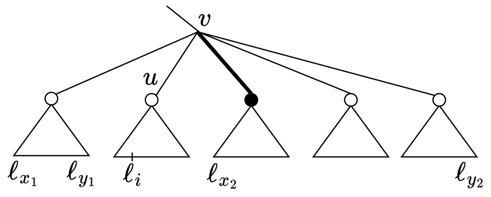

Figure 2 Node u is a light child of v in the heavy path decomposition of $\mathcal{T}_{\mathcal{L}}$.

▶ **Observation 7.** *For any root-to-leaf path in a tree with n vertices, at most $\log n$ light nodes appear on the path.*

Let $\mathcal{T}$ be the suffix tree of $T_1\$_1T_2\$_2$. We first find the subset of leaves $\mathcal{L}$ of $\mathcal{T}$ corresponding to the specified suffix sets $\mathcal{S}_1$ and $\mathcal{S}_2$ and obtain the sparse suffix tree, $\mathcal{T}_{\mathcal{L}}$. Sparse suffix tree construction can be done in $\widetilde{\mathcal{O}}(|\mathcal{L}|)$ time by adding the LCA of each adjacent pair of sampled suffixes as an internal node in the sparse suffix tree. We next preprocess $\mathcal{T}_{\mathcal{L}}$. Recall that ℓ_i represents the i^{th} leaf in $\mathcal{T}$ (rather than the i^{th} leaf in $\mathcal{T}_{\mathcal{L}}$). For the leaves of these subtrees, we will use their BWT values to determine left-maximality. We also use an additional value 1 or 2 to indicate which string the corresponding suffix belongs to. Hence, we assign to each leaf $\ell_i \in \mathcal{L}$ a 2D point $(i, \mathrm{BWT}[i], b)$ and weight $\mathrm{SA}[i]$ where $b = 1$ if leaf ℓ_i corresponds to a suffix of T_1 and $b = 2$ if leaf ℓ_i corresponds to a suffix of T_2. In the case of leftmost pairs of interest, we construct a 2D-RmQ over this set of points, and for rightmost pairs of interest, we construct a 2D-RMQ. Either case requires $\mathcal{O}(|\mathcal{L}| \log |\mathcal{L}|)$ time. In $\mathcal{O}(|\mathcal{L}|)$ time, we also make it so that every node in $\mathcal{T}_{\mathcal{L}}$ has a pointer to its leftmost and rightmost leaf in its subtree. Next, we take a heavy path decomposition of $\mathcal{T}_{\mathcal{L}}$.

Assume we are seeking leftmost pairs of interest. For rightmost pairs of interest, the same procedure is used but with RMQ queries instead. We iterate through the internal nodes of $\mathcal{T}_{\mathcal{L}}$. For each node v, we consider each of its light children. Let u be a light child of v. We iterate through the leaves in the subtree for u. Let ℓ_i be a leaf in the subtree for u.

First, assume ℓ_i corresponds to a suffix of T_1. For the union of children of v to the left of u and the union of children of v to the right of u, we get the leftmost and rightmost leaves. Say these are ℓ_{x_1},ℓ_{y_1} and ℓ_{x_2}, ℓ_{y_2}, respectively (see Figure 2). Having a BWT value in either range $[-\infty \mathinner{..} \mathrm{BWT}[i]-1]$ or $[\mathrm{BWT}[i]+1 \mathinner{..} \infty]$ implies a left-maximal match with the substring starting at $\mathrm{SA}[i]$. Since we are seeking leftmost pairs of interest, we make the RmQ queries, $\mathrm{RmQ}([x_1 \mathinner{..} y_1] \times [-\infty \mathinner{..} \mathrm{BWT}[i]-1] \times [2 \mathinner{..} 2])$, $\mathrm{RmQ}([x_1 \mathinner{..} y_1] \times [\mathrm{BWT}[i]+1 \mathinner{..} \infty] \times [2 \mathinner{..} 2])$, $\mathrm{RmQ}([x_2 \mathinner{..} y_2] \times [-\infty \mathinner{..} \mathrm{BWT}[i]-1] \times [2 \mathinner{..} 2])$, $\mathrm{RmQ}([x_2 \mathinner{..} y_2] \times [\mathrm{BWT}[i]+1 \mathinner{..} \infty] \times [2 \mathinner{..} 2])$ and take the point with the minimum weight of those returned (For rightmost pairs of interest, the point with the maximum weight is taken). Assuming some point $(j, \mathrm{BWT}[j], 2)$ is found, we create the point $(\mathrm{SA}[i], \mathrm{SA}[j])$ with weight equal to the string depth of v and add it to the set of points to output.

If instead ℓ_i corresponds to a suffix of T_2, then we find ℓ_{x_1},ℓ_{y_1} and ℓ_{x_2}, ℓ_{y_2} as above. We make the RmQ queries, $\mathrm{RmQ}([x_1 \mathinner{..} y_1] \times [-\infty \mathinner{..} \mathrm{BWT}[i]-1] \times [1 \mathinner{..} 1])$, $\mathrm{RmQ}([x_1 \mathinner{..} y_1] \times [\mathrm{BWT}[i]+1 \mathinner{..} \infty] \times [1 \mathinner{..} 1])$, $\mathrm{RmQ}([x_2 \mathinner{..} y_2] \times [-\infty \mathinner{..} \mathrm{BWT}[i]-1] \times [1 \mathinner{..} 1])$, $\mathrm{RmQ}([x_2 \mathinner{..} y_2] \times [\mathrm{BWT}[i]+1 \mathinner{..} \infty] \times [1 \mathinner{..} 1])$ and take the point with the minimum weight of those returned. Assuming some point $(j, \mathrm{BWT}[j], 1)$ is found, we create the point $(\mathrm{SA}[j], \mathrm{SA}[i])$ with weight equal to the string depth of v and add it to the set of points to output.

Applying the key property of heavy path decomposition that every leaf has at most $\log |\mathcal{T}_{\mathcal{L}}|$ light ancestors, each leaf gets processed by the above algorithm at most $\log |\mathcal{T}_{\mathcal{L}}|$

times. Since each time a leaf is considered requires $\mathcal{O}(\log|\mathcal{L}|)$ time, the total time taken per leaf is $\mathcal{O}(\log^2|\mathcal{L}|)$ and the total time over the entire sparse suffix tree $\mathcal{T}_{\mathcal{L}}$ is $\mathcal{O}(|\mathcal{L}|\log^2|\mathcal{L}|)$. However, since each time a leaf is considered at most one point is added to the output set, the size of the set returned is $\mathcal{O}(|\mathcal{L}|\log|\mathcal{L}|)$.

Further Modifications. For space considerations, details of the further modifications to the algorithm from Section 3 are moved to the full version of this work.

5 Our Algorithm for Pattern Matching with Gaps

We solve Problem 4 in this section. Define $\mathsf{CHAIN}(i,j,k)$ to be equal to the leftmost index i'_h such that there exists a gapped match of $P[j\,..]$ in $T[i\,..]$ ending at i'_h or ∞ if no such occurrence exists. Formally, i'_h is the smallest i'_h such that there exists $T[i_1\,..\,i'_1]$, $T[i_2\,..\,i'_2]$, …, $T[i_h\,..\,i'_h]$ where $h \leq k$, $T[i_1\,..\,i'_1]\cdot T[i_2\,..\,i'_2]\cdot\ldots\cdot T[i_h\,..\,i'_h] = P[j\,..]$, and $i_1 \geq i$. An $\mathcal{O}(knm)$ time solution is achieved by solving the following recurrence:

$$\mathsf{CHAIN}(i,j,k) = \min\{\mathsf{CHAIN}(i+1,j,k), \mathsf{CHAIN}(i+\mathrm{LCE}(i,j), j+\mathrm{LCE}(i,j), k-1)\}$$

The final solution is obtained by taking the $i \in [1\,..\,n]$ that minimizes $\mathsf{CHAIN}(i,1,k)-i$. The matches in the solution can be found with back pointers. We now show how our techniques provide a strongly subquadratic time algorithm for constant k.

5.1 Construction

For $f = 1$, using a suffix tree of $T\$_1P\$_2$, we can locate the leftmost occurrence of $P[j\,..]$ in the suffix $T[i\,..]$ in $\widetilde{\mathcal{O}}(1)$ time after $\widetilde{\mathcal{O}}(n)$ time preprocessing. Hence, we can answer $\mathsf{CHAIN}(i,j,1)$ queries in $\widetilde{\mathcal{O}}(1)$ time. For $f \in [2\,..\,k-1]$, we have the following construction of $\mathsf{DS}(f)$: we compute for every $j \in [2\,..\,m]$ and $i \in [1\,..\,n]$ where i is a multiple of b_f, $\mathsf{CHAIN}(i,j,f)$ by taking the minimum of $\mathsf{CHAIN}_s(i,j,f)$ and $\mathsf{CHAIN}_l(i,j,f)$, which are found as follows.

Computing CHAIN_s. For $h \in [1\,..\,d_f]$, we find the leftmost right-maximal match of $P[j\,..\,j+h)$ in the suffix $T[i\,..]$ (if it exists). If such a match exists, say starting at position x in T, do a query $\mathsf{DS}(f-1)$ for $\mathsf{CHAIN}(x+h, j+h, f-1)$. Let w_h be equal to the query result if the match exists and ∞ otherwise. We make $\mathsf{CHAIN}_s(i,j,f) = \min_{1\leq h\leq d_f} w_h$. The total number of queries is $\mathcal{O}(nmd_f/b_f)$, which is $\mathcal{O}(nm/\sqrt{b_f})$.

Computing CHAIN_l. We utilize a d_f-cover $\mathcal{D}$ of $T\$_1P\$_2$. There are $\mathcal{O}(nm/d_f)$ anchor pairs $a_1 \in \mathcal{D}\cap[1\,..\,n]$ and $a_2 \in \mathcal{D}\cap[n+2\,..\,n+m+1]$. We iterate over all of these pairs. For a given pair a_1 and $a'_2 = a_2-(n+1)$, we compute $\mathrm{LCE}(a_1,a'_2)$ and $\overline{\mathrm{LCE}}(a_1,a'_2)$.

For a given MEM found this way, say $T[x\,..\,x+h)$ and $P[y\,..\,y+h)$, we use $\mathsf{DS}(f-1)$ to compute $\mathsf{CHAIN}(x+h, y+h, f-1)$. Observe that for a given i and j,

1. If $j < y$, or $j \geq y+h$, or $y \leq j < y+h$ and $i-x > j-y$, then no portion of exact match $T[x\,..\,x+h)$ and $P[y\,..\,y+h)$ is included in the first match of a solution.
2. If $y \leq j < y+h$, and $i-x \leq j-y$, then the portion of the match including $P[j\,..\,y+h)$ can potentially be used as the first match in the solution with value $\mathsf{CHAIN}(x+h, y+h, f-1)$.

As such, we create the point $(y, y+h, x-y)$ with weight $\mathsf{CHAIN}(x+h, y+h, f-1)$.

Once all such points are created, they are stored in an RmQ structure. Now for a given i and j we make the query $[1\,..\,j]\times[j+1\,..\,m]\times[i-j\,..\,\infty]$ and let $\mathsf{CHAIN}_l(i,j,f)$ be equal to the weight of the resulting point or ∞ if no point exists. The total number of queries done to $\mathsf{DS}(f-1)$ is $\mathcal{O}(nm/d_f)$, which is $\mathcal{O}(nm/\sqrt{b_f})$.

5.2 Querying

To find $\mathsf{CHAIN}(i, j, f)$ for an arbitrary $i \in [1\,..\,n]$ and $j \in [2\,..\,m]$, we let $i' = b_f \lceil i/b_f \rceil$. For each $h \in [i\,..\,i')$ we compute $\mathrm{LCE}(h, i)$ and, using $\mathsf{DS}(f-1)$, the value $\mathsf{CHAIN}(h + \mathrm{LCE}(h,i), j + \mathrm{LCE}(h,j), f-1)$. We take the minimum of these values and the precomputed value $\mathsf{CHAIN}(i', j, f)$. The total number of queries to $\mathsf{DS}(f-1)$ is $\mathcal{O}(b_f)$.

5.3 Analysis and Obtaining the Final Answer

The analysis is the same as in Section 3.2.4. We have for $f \geq 2$, $S(f) = S(f-1) + \mathcal{O}(nm/b_f)$, $C(f) = C(f-1) + Q(f-1) \cdot \widetilde{\mathcal{O}}(nm/\sqrt{b_f})$, $Q(f) = Q(f-1) \cdot \mathcal{O}(b_f)$. We let $b_f = \lceil m^{c3^f} \rceil$ with $c = 2/3^k$. This makes $C(k-1) = \widetilde{\mathcal{O}}(nm^{1-9c/2})$ and $Q(k-1) = \mathcal{O}(m^{c(3^2+\ldots+3^{k-1})}) = \mathcal{O}(m^{c(3^k-9)/2})$.

To obtain the final answer, for each $i \in [1\,..\,n]$, we compute $\mathrm{LCE}(i,1)$. For every i where $\mathrm{LCE}(i,1) \geq 1$, we use $\mathsf{DS}(k-1)$ to obtain $\mathsf{CHAIN}(i + \mathrm{LCE}(i,1), 1 + \mathrm{LCE}(i,1), k-1)$. A minimum is take over all resulting values. The total time is $C(k-1) + n \cdot Q(k-1) = \widetilde{\mathcal{O}}(nm^{1-9c/2} + nm^{c(3^k-9)/2})$. With $c = 2/3^k$, this yields an overall time complexity of $\widetilde{\mathcal{O}}(nm^{1-1/3^{k-2}})$. Back pointers can be used to obtain the actual solution.

6 Open Problems and Discussion

Another interesting variant of the Pattern Matching with Gaps problem is the following:

▶ **Problem 5** (Pattern Matching with (Min-Max) Gap). *Given a text $T[1\,..\,n]$ and pattern $P[1\,..\,m]$ and parameter k, find substrings (if they exist) $T[i_1\,..\,i'_1]$, $T[i_2\,..\,i'_2]$, …, $T[i_h\,..\,i'_h]$ such that $i'_1 < i_2$, …, $i'_{h-1} < i_h$, $T[i_1\,..\,i'_1] \cdot T[i_2\,..\,i'_2] \cdot \ldots \cdot T[i_h\,..\,i'_h] = P$, $h \leq k$, and $\max_{2 \leq s \leq h}\{i_s - i'_{s-1}\}$ is minimized.*

This can be solved with dynamic programming in time $\widetilde{\mathcal{O}}(knm)$. We use a threshold value g. We now define $\mathsf{CHAIN}(i,j,k)$ equal to the smallest $i_1 \geq i$ such that there exists $T[i_1\,..\,i'_1] \cdot T[i_2\,..\,i'_2] \cdot \ldots \cdot T[i_h\,..\,i'_h] = P[j\,..]$ where $i'_1 < i_2$, and $\max_{2 \leq s \leq h}\{i_s - i'_{s-1}\} \leq g$. Then, $\mathsf{CHAIN}(i,j,k) = \min\{\mathsf{CHAIN}(i+1,j,k), Z\}$ where if $\mathrm{LCE}(i,j) = m - j + 1$ or $\mathsf{CHAIN}(i + \mathrm{LCE}(i,j), j + \mathrm{LCE}(i,j), k-1) - (i + \mathrm{LCE}(i,j) - 1) \leq g$ then $Z = i$, otherwise $Z = \infty$. With logarithmic factor overhead we perform binary search of the g value to find the smallest g such that $\mathsf{CHAIN}(i,1,k) \neq \infty$ for some i.

We leave open the problem of developing a strongly subquadratic time solution for this problem. One apparent difficulty in utilizing the same techniques that were used in Problem 4 is that, when computing the analogous CHAIN_s values, it is no longer sufficient to only consider the leftmost occurrence in $T[i\,..]$ of a fixed fragment $P[j\,..\,j+h)$. At the same time, picking the rightmost occurrence of $P[j\,..\,j+h)$ that is at most g from i may result in eliminating potential solutions for the remaining suffix of P.

Future Directions. We leave open whether this time complexity can be improved significantly and whether a $\Omega(nm)$ conditional lower bound can be established for small k, e.g., $k = \mathrm{polylog}(n)$. Such a lower bound would make an $\mathcal{O}(kn)$ time algorithm (like that known for LCS where k is the length of the subsequence [35]) impossible under the applied hardness assumption. The current SETH-based conditional lower bounds for LCS use $\tilde{\Theta}(n)$ gaps between elements of the subsequence. The SETH-based lower bound k-mismatch LCS for $k = \omega(\log n)$ of Kociumaka et al. [39] is also difficult to adapt without imposing some additional restrictions on the problem, such as a constraint on the allowed gap length.

References

1 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 375–388. ACM, 2016. doi:10.1145/2897518.2897653.

2 Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 218–230. SIAM, 2015. doi:10.1137/1.9781611973730.17.

3 Mohamed I Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Coconut: an efficient system for the comparison and analysis of genomes. *BMC bioinformatics*, 9(1):476, 2008. doi:10.1186/1471-2105-9-476.

4 Mohamed Ibrahim Abouelhoda and Enno Ohlebusch. Multiple genome alignment: Chaining algorithms revisited. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*, volume 2676 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2003. doi:10.1007/3-540-44888-8_1.

5 Mohamed Ibrahim Abouelhoda and Enno Ohlebusch. Chaining algorithms for multiple genome comparison. *J. Discrete Algorithms*, 3(2-4):321–341, 2005. doi:10.1016/j.jda.2004.08.011.

6 Srinivas Aluru, Alberto Apostolico, and Sharma V. Thankachan. Efficient alignment free sequence comparison with bounded mismatches. In Teresa M. Przytycka, editor, *Research in Computational Molecular Biology - 19th Annual International Conference, RECOMB 2015, Warsaw, Poland, April 12-15, 2015, Proceedings*, volume 9029 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2015. doi:10.1007/978-3-319-16706-0_1.

7 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/s00453-020-00744-0.

8 Alberto Apostolico and Mikhail J. Atallah. Compact recognizers of episode sequences. *Inf. Comput.*, 174(2):180–192, 2002. doi:10.1006/INCO.2002.3143.

9 Kyriakos Axiotis and Christos Tzamos. Capacitated dynamic programming: Faster knapsack and graph algorithms. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 19:1–19:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.19.

10 Maxim A. Babenko and Tatiana Starikovskaya. Computing the longest common substring with one mismatch. *Probl. Inf. Transm.*, 47(1):28–33, 2011. doi:10.1134/S0032946011010030.

11 Arturs Backurs, Piotr Indyk, and Ludwig Schmidt. Better approximations for tree sparsity in nearly-linear time. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2215–2229. SIAM, 2017. doi:10.1137/1.9781611974782.145.

12 Johannes Bader, Simon Gog, and Matthias Petri. Practical variable length gap pattern matching. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, volume 9685 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2016. doi:10.1007/978-3-319-38851-9_1.

13 Philip Bille, Inge Li Gørtz, Shay Mozes, Teresa Anna Steiner, and Oren Weimann. The fine-grained complexity of episode matching. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 4:1–4:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CPM.2022.4.

14 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind. String matching with variable length gaps. *Theor. Comput. Sci.*, 443:25–34, 2012. `doi:10.1016/J.TCS.2012.03.029`.

15 Philip Bille and Mikkel Thorup. Regular expression matching with multi-strings and intervals. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1297–1308. SIAM, 2010. `doi:10.1137/1.9781611973075.104`.

16 Luc Boasson, Patrick Cégielski, Irène Guessarian, and Yuri V. Matiyasevich. Window-accumulated subsequence matching problem is linear. *Ann. Pure Appl. Log.*, 113(1-3):59–80, 2001. `doi:10.1016/S0168-0072(01)00051-3`.

17 Nick Bray, Inna Dubchak, and Lior Pachter. Avid: A global alignment program. *Genome research*, 13(1):97–102, 2003. `doi:10.1101/gr.789803`.

18 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97. IEEE Computer Society, 2015. `doi:10.1109/FOCS.2015.15`.

19 Karl Bringmann and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1216–1235. SIAM, 2018. `doi:10.1137/1.9781611975031.79`.

20 Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2003. `doi:10.1007/3-540-44888-8_5`.

21 Michael Burrows, D J Wheeler D I G I T A L, Robert W. Taylor, David J. Wheeler, and David Wheeler. A block-sorting lossless data compression algorithm. In , 1994. URL: `https://api.semanticscholar.org/CorpusID:2167441`.

22 Manuel Cáceres, Simon J. Puglisi, and Bella Zhukova. Fast indexes for gapped pattern matching. In Alexander Chatzigeorgiou, Riccardo Dondi, Herodotos Herodotou, Christos A. Kapoutsis, Yannis Manolopoulos, George A. Papadopoulos, and Florian Sikora, editors, *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings*, volume 12011 of *Lecture Notes in Computer Science*, pages 493–504. Springer, 2020. `doi:10.1007/978-3-030-38919-2_40`.

23 Patrick Cégielski, Irène Guessarian, and Yuri V. Matiyasevich. Multiple serial episodes matching. *Inf. Process. Lett.*, 98(6):211–218, 2006. `doi:10.1016/J.IPL.2006.02.008`.

24 Mark J Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC bioinformatics*, 13(1):238, 2012. `doi:10.1186/1471-2105-13-238`.

25 Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Linear-time algorithm for long LCF with k mismatches. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.CPM.2018.23`.

26 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Faster algorithms for longest common substring. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 30:1–30:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.ESA.2021.30`.

27 Charles J. Colbourn and Alan C. H. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75(1-2):9–12, 2000. `doi:10.1016/S0020-0190(00)00080-6`.

28 Maxime Crochemore, Costas S. Iliopoulos, Christos Makris, Wojciech Rytter, Athanasios K. Tsakalidis, and T. Tsichlas. Approximate string matching with gaps. *Nord. J. Comput.*, 9(1):54–65, 2002.

29 Gautam Das, Rudolf Fleischer, Leszek Gasieniec, Dimitrios Gunopulos, and Juha Kärkkäinen. Episode matching. In Alberto Apostolico and Jotun Hein, editors, *Combinatorial Pattern Matching, 8th Annual Symposium, CPM 97, Aarhus, Denmark, June 30 - July 2, 1997, Proceedings*, volume 1264 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 1997. doi:10.1007/3-540-63220-4_46.

30 Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008. URL: https://www.worldcat.org/oclc/227584184.

31 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

32 Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Inf. Process. Lett.*, 115(6-8):643–647, 2015. doi:10.1016/J.IPL.2015.03.006.

33 Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Faster longest common extension queries in strings over general alphabets. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 5:1–5:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.5.

34 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

35 Daniel S Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.

36 James W Hunt and Thomas G Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

37 Chirag Jain, Daniel Gibney, and Sharma V. Thankachan. Co-linear chaining with overlaps and gap costs. In Itsik Pe'er, editor, *Research in Computational Molecular Biology - 26th Annual International Conference, RECOMB 2022, San Diego, CA, USA, May 22-25, 2022, Proceedings*, volume 13278 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2022. doi:10.1007/978-3-031-04749-7_15.

38 Chirag Jain, Arang Rhie, Nancy F Hansen, Sergey Koren, and Adam M Phillippy. Long-read mapping to repetitive reference sequences using winnowmap2. *Nature Methods*, pages 1–6, 2022. doi:10.1038/s41592-022-01457-8.

39 Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya. Publisher correction: Longest common substring with approximately k mismatches. *Algorithmica*, 85(10):3323, 2023. doi:10.1007/S00453-023-01119-X.

40 Stefan Kurtz et al. Versatile and open software for comparing large genomes. *Genome biology*, 5(2):R12, 2004. doi:10.1186/gb-2004-5-2-r12.

41 Moshe Lewenstein. Indexing with gaps. In Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri, editors, *String Processing and Information Retrieval, 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings*, volume 7024 of *Lecture Notes in Computer Science*, pages 135–143. Springer, 2011. doi:10.1007/978-3-642-24583-1_14.

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

43 Tiantian Li, Haitao Jiang, Xuefeng Cui, Haodi Feng, and Daming Zhu. Longest order-consistent and number-limited common substrings. *Available at SSRN 4724959*, 2024.

44 Tiantian Li, Daming Zhu, Haitao Jiang, Haodi Feng, and Xuefeng Cui. Longest k-tuple common sub-strings. In Donald A. Adjeroh, Qi Long, Xinghua Mindy Shi, Fei Guo, Xiaohua

Hu, Srinivas Aluru, Giri Narasimhan, Jianxin Wang, Mingon Kang, Ananda Mondal, and Jin Liu, editors, *IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2022, Las Vegas, NV, USA, December 6-8, 2022*, pages 63–66. IEEE, 2022. doi:10.1109/BIBM55620.2022.9995199.

45 Mamoru Maekawa. A square root N algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

46 Veli Mäkinen, Gonzalo Navarro, and Esko Ukkonen. Transposition invariant string matching. *J. Algorithms*, 56(2):124–153, 2005. doi:10.1016/J.JALGOR.2004.07.008.

47 Veli Mäkinen and Kristoffer Sahlin. Chaining with overlaps revisited. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161, pages 25:1–25:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.25.

48 Guillaume Marçais, Arthur L Delcher, et al. Mummer4: A fast and versatile genome alignment system. *PLoS computational biology*, 14(1):e1005944, 2018. doi:10.1371/journal.pcbi.1005944.

49 Michele Morgante, Alberto Policriti, Nicola Vitacolonna, and Andrea Zuccolo. Structured motifs search. *J. Comput. Biol.*, 12(8):1065–1082, 2005. doi:10.1089/CMB.2005.12.1065.

50 Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986. doi:10.1007/BF01840446.

51 Gene Myers and Webb Miller. Chaining multiple-alignment fragments in sub-quadratic time. In Kenneth L. Clarkson, editor, *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA*, pages 38–47. ACM/SIAM, 1995. URL: http://dl.acm.org/citation.cfm?id=313651.313661.

52 Christian Otto, Steve Hoffmann, Jan Gorodkin, and Peter F Stadler. Fast local fragment chaining using sum-of-pair gap costs. *Algorithms for Molecular Biology*, 6(1):4, 2011. doi:10.1186/1748-7188-6-4.

53 M. Sohel Rahman, Costas S. Iliopoulos, Inbok Lee, Manal Mohamed, and William F. Smyth. Finding patterns with variable length gaps or don't cares. In Danny Z. Chen and D. T. Lee, editors, *Computing and Combinatorics, 12th Annual International Conference, COCOON 2006, Taipei, Taiwan, August 15-18, 2006, Proceedings*, volume 4112 of *Lecture Notes in Computer Science*, pages 146–155. Springer, 2006. doi:10.1007/11809678_17.

54 Jingwen Ren and Mark JP Chaisson. lra: A long read aligner for sequences and contigs. *PLOS Computational Biology*, 17(6):e1009078, 2021. doi:10.1371/journal.pcbi.1009078.

55 Kristoffer Sahlin and Veli Mäkinen. Accurate spliced alignment of long RNA sequencing reads. *Bioinformatics*, 37(24):4643–4651, 2021. doi:10.1093/bioinformatics/btab540.

56 Fritz J Sedlazeck et al. Accurate detection of complex structural variations using single-molecule sequencing. *Nature methods*, 15(6):461–468, 2018. doi:10.1038/s41592-018-0001-7.

57 Tetsuo Shibuya and Igor Kurochkin. Match chaining algorithms for cDNA mapping. In *Algorithms in Bioinformatics, Third International Workshop, WABI 2003, Budapest, Hungary, September 15-20, 2003, Proceedings*, pages 462–475, 2003. doi:10.1007/978-3-540-39763-2_33.

58 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

59 Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In Benjamin J. Raphael, editor, *Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*, volume 10812 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2018. doi:10.1007/978-3-319-89929-9_14.

60 Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. *J. Comput. Biol.*, 23(6):472–482, 2016. doi:10.1089/cmb.2015.0235.

61 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.