String Indexing and Compression

Pedersen, Max Harry Rishøj

Publication date:
2023

Document Version
Publisher's PDF, also known as Version of record

String Indexing and Compression

Max Harry Rishøj Pedersen

July 2023

Preface

The research presented in this dissertation was conducted while I was enrolled as a PhD student at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark. My work was funded by the Danish Research Council project "Adaptive Compressed Computation" (DFF-8021-002498). I began my PhD studies February 1st, 2020 and ended on July 1st, 2023. My supervisors were Professors Inge Li Gørtz and Philip Bille. I spent two months at Technische Universität Dortmund with Professor Johannes Fischer as my host supervisor.

Acknowledgements. I want to thank my supervisors Inge Li Gørtz and Philip Bille for inspiring me to pursue my degree, as well as for being immensely supportive, helpful and sharp. I also thank my host supervisor Johannes Fischer and his PhD student Jonas Ellert for making my visit to Germany pleasant, fun and productive, and for being inspiring academic collaborators. Further thank you to all of my co-authors which, in addition to the previously mentioned people, include Eva Rotenberg, Teresa Anna Steiner and Tord Joakim Stordalen. I would also like to extend gratitude to all my colleagues and friends at DTU.

Last but not least, I would like to thank the love of my life, Lula Martina Brambati Lund. Without her unwavering support, I would not have made it this far and there would not be a dissertation today.

Abstract

We study the design of efficient algorithms and data structures for variants of combinatorial pattern matching, as well as compression of strings and string indices.

String Indexing for Top-k Close Consecutive Occurrences We study the *string indexing for top-k close consecutive occurrences problem*, where the goal is to preprocess a given string S into a compact data structure that efficiently supports the query: given a string P and positive integer k, report the k closest *consecutive occurrences* of P in S. Here, a consecutive occurrence is a pair (i, j), $i < j$, such that P occurs at positions i and j in S and there is no occurrence of P between i and j, and their distance is defined as $j - i$. We present three time-space trade-offs. Let n be the length of S, m the length of P, and $\epsilon \in (0, 1]$. Our first result achieves $O(n \log n)$ space and optimal query time of $O(m + k)$. Our second and third results achieve linear space and query times either $O(m + k^{1+\epsilon})$ or $O(m + k \log^{1+\epsilon} n)$. We also extend our solutions to several related problems.

Gapped Indexing for Consecutive Occurrences We study a variant of string indexing where the goal is to compactly represent a string such that given two patterns P_1 and P_2 and a gap range $[\alpha, \beta]$, we can quickly find the consecutive occurrences of P_1 and P_2 with distances in $[\alpha, \beta]$, i.e., pairs of subsequent occurrences of the two patterns that are between α and β characters apart. We present data structures that use $\widetilde{O}(n)$ space and $\widetilde{O}(|P_1|+|P_2|+n^{2/3})$ time for existence and counting queries and $\widetilde{O}(|P_1|+|P_2|+n^{2/3}\mathrm{occ}^{1/3})$ time for reporting queries. We complement this with a conditional lower bound based on the set intersection problem showing that any solution using $\widetilde{O}(n)$ space must use $\widetilde{\Omega}(|P_1| + |P_2| + \sqrt{n})$ query time.

Sliding Window String Indexing We study the *streaming sliding window string indexing problem*, where a string S arrives as a stream and the goal is to maintain an index of the last w characters, called the *window*. At any point in time a pattern matching query for a pattern P may arrive, also streamed, and all occurrences of P within the window must be returned. We present a simple $O(w)$ space data structure that uses $O(\log w)$ time with high probability to process each character from either stream, plus additional worst-case constant time per reported occurrence. Compared to previous work in similar scenarios this result is the first to achieve an efficient worst-case time per character from the input stream with high probability. We also consider a delayed variant of the problem, where a query may be answered at any point within the next δ characters that arrive from either stream. We present an $O(w + \delta)$ space data structure for this problem that improves the above time bounds to $O(\log(w/\delta))$. In particular, for a delay of $\delta = \epsilon w$ we obtain an $O(w)$ space data structure with constant time processing per character.

New Advances in Rightmost Lempel-Ziv The Lempel-Ziv (LZ) 77 factorization for a string S of length n, over a linearly-sortable alphabet, can be computed in $O(n)$ time. It is unknown whether this time can be achieved for the *rightmost* LZ parsing, where each referencing phrase points to its rightmost previous occurrence. The currently best solution takes $O(n(1 + \log \sigma/\sqrt{\log n}))$ time [Belazzougui & Puglisi SODA2016]. We show that this problem is much easier to solve for the LZ-End factorization [Kreft & Navarro DCC2010], where the rightmost factorization can be obtained in $O(n)$ time for the greedy parsing (with phrases of maximal length), and in $O(n + z\sqrt{\log z})$ time for any LZ-End parsing of z phrases. We also

make advances towards a linear time solution for the general case. We show how to solve several non-trivial subsets of the phrases of any LZ-like parsing in $O(n)$ time. As a prime example, we can find the rightmost occurrence of all phrases of length $\Omega(\log^{6.66} n/\log^2 \sigma)$ in $O(n/\log_\sigma n)$ time and space.

Fast Compression of Deterministic Finite Automata The *delayed deterministic finite automaton* [Kumar et al. SIGCOMM2006] is an effective compressed representation of *deterministic finite automata* that forms the basis of numerous subsequent compression results. The compression algorithm, as well as later algorithms based on the idea, have an inherent quadratic-time bottleneck as they consider every pair of states to compute the optimal compression. We present a simple, general framework based on locality-sensitive hashing for circumventing this bottleneck. We apply it to speed up several algorithms and experimentally evaluate them on real-world regular expression sets. We obtain an order of magnitude improvement in compression time, with little to no loss of compression, or even significantly better compression in some cases.

Contents

Chapter 1

Introduction

This dissertation is based on the full versions of the following papers:

Chapter 2: String Indexing for Top-k Close Consecutive Occurrences. Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, Eva Rotenberg, and Teresa Anna Steiner. In *Proceedings of the 40th Conference on Foundations of Software Technology*, 2020 and *Theoretical Computer Science*, 2022.

Chapter 3: Gapped Indexing for Consecutive Occurrences. Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Teresa Anna Steiner. In *Proceedings of the 32nd Annual Symposium on Combinatorial Pattern Matching*, 2021 and *Algorithmica*, 2023.

Chapter 4: Sliding Window String Indexing in Streams Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Tord Joaking Stordalen. In *Proceedings of the 34th Annual Symposium on Combinatorial Pattern Matching*, 2023. Awarded "best student paper".

Chapter 5: New Advances in Rightmost Lempel-Ziv. Jonas Ellert, Johannes Fischer, and Max Rishøj Pedersen. Submitted to a conference.

Chapter 6: Fast Compression of Determinstic Finite Automata. Philip Bille, Inge Li Gørtz, and Max Rishøj Pedersen. Unpublished.

1.1 Chapter Outline

All work in this dissertation relates to strings and string indexing. Chapters 2-4 study variants of string indexing for pattern matching, and Chapters 5 and 6 study string compression and compressed string indexing, respectively. In this chapter we provide an overview of the work presented in the later chapters. In Section 1.2 and 1.3 we outline the problems studied and our results. In Section 1.4 we highlight some of the main techniques used to achieve our results. In Section 1.5 we discuss some of the most interesting open questions that spring from our work.

1.2 String Indexing

Chapters 2-4 study variants of string indexing problems. Given two strings S and P, often called the *text* and the *pattern* respectively, the classic pattern matching problem is to find all the *occurrences* of P in S, i.e., the positions in S where a substring equal to P starts. Optimal solutions to this problem have been known for a long time, such as the KMP algorithm which solves the problem in worst-case $O(n+m)$ time, where $n = |S|$ and $m = |P|$ is the length of the text and pattern respectively. In the classic string indexing problem the goal is to preprocess S such that subsequent pattern matching queries can be quickly answered, i.e., without the

additional $O(n)$ term. In addition to computing all the occurrences of P (*reporting* queries), typical queries include determining if P occurs at all (*existence* queries) or counting how many times P occurs (*counting* queries). In this dissertation we consider the following variants of the string indexing problem:

- **Indexing for Top-k Close Consecutive Occurrences**: Preprocess a string S into a compact data structure such that given a pattern P and a positive integer k, we can quickly find the k closest *consecutive* occurrences of P in S, that is, the k closest (minimal distance) pairs of occurrences of P that have no other occurrences of P in between.
- **Indexing for Gapped Consecutive Occurrences**: Preprocess a string S into a compact data structure such that given two patterns P_1 and P_2 and an interval $[\alpha, \beta]$, we can quickly find the consecutive occurrences of P_1 and P_2 in S that are between α and β characters apart, that is, pairs (i, j) for $i < j$ and $\alpha \leq j - i \leq \beta$, such that P_1 occurs at i, P_2 occurs at j and neither P_1 nor P_2 occurs between i and j.
- **Sliding Window Indexing in Streams**: Given a positive integer w and a string S that is streamed one character at a time, maintain an index over the w most recently arrived characters of S, called the *window*, such that given a streamed pattern P we can efficiently find the occurrences of P in the window.

1.2.1 Indexing for Top-k Close Consecutive Occurrences

In this problem the goal is to preprocess a string S such that we can quickly answer queries of a pattern P and integer $k > 0$, finding the k closest consecutive occurrences of P. A consecutive occurrence of P is a pair (i, j) of occurrences of P in S such that P does not occur between i and j, and we define the distance to be $j - i$. Note that, if desired, the non-consecutive occurrences can be constructed from the consecutive occurrences.

History and Related Work

Though the problem is closely related to several well-studied problems, this specific formulation had not been studied before our work. There are several related string indexing problems for pattern matching under various distance constraints [BG14, IR09, BGP16, CPZ20, BGVV14, Lew11, KKL07], as well as indexing of collections of strings (often called *documents*) for reporting the top-k documents minimizing some function, e.g., the shortest distance of a consecutive occurrence within a document [HSTV14, NN17, SSTV13, HSTV14, NN17, MNN$^+$17]. For an overview see the survey by Navarro [Nav14].

Navarro and Thankachan [NT16] studied a closely related indexing problem, where the goal is to report consecutive occurrences with distances within a specified interval. Their solution uses $O(n \log n)$ space and reports the consecutive occurrences in $O(m + \text{occ})$ time, where $m = |P|$ is the length of the pattern and occ is the size of the output. We note that their solution can be extended to solve our problem with optimal $O(m + k)$ query time but uses $O(n \log n)$ space. The primary aim of our work was simultaneously achieving linear space and fast queries.

Results

In Chapter 2 we present several trade-offs for the problem. First, we present an $O(n \log n)$ space solution with optimal $O(m + k)$ query time, matching Navarro and Thankachan while using a simpler reduction. Second, we present two $O(\frac{n}{\epsilon})$ space solutions answering queries in respectively $O(m + k^{1+\epsilon})$ and $O(m + k \log^{1+\epsilon} n)$ time, for any ϵ with $0 < \epsilon \leq 1$. We extend these solutions to solve the closely related problem of finding the k *furthest* consecutive occurrences, and we show both an $O(n \log n)$ space and optimal time solution, and an $O(\frac{n}{\epsilon})$ space and $O(m + k^{1+\epsilon})$ time solution. We also extend our solutions to solve special cases of the problem studied by Navarro and Thankachan [NT16], i.e., finding all consecutive occurrences with distances within a query interval $[\alpha, \beta]$. If either α or β is fixed at indexing time, we obtain an $O(\frac{n}{\epsilon})$ space solution with

$O(m+\text{occ}^{1+\epsilon})$ query time. Additionally, if $\alpha = 1$ is fixed, we also obtain a solution with $O(m+\text{occ}\log^{1+\epsilon} n)$ query time. Finally we extend our solutions to the problem of finding all the non-overlapping consecutive occurrences, and obtain an $O(\frac{n}{\epsilon})$ space and $O(m+\text{occ}^{1+\epsilon})$ query time solution.

1.2.2 Indexing for Gapped Consecutive Occurrences

In this problem the goal is to preprocess a string S such that we can quickly answer queries of two patterns P_1 and P_2, and an interval $[\alpha, \beta]$, finding all the consecutive occurrences of P_1 and P_2 with distances between α and β. That is, pairs (i, j) for $\alpha \leq j - i \leq \beta$, such that i is an occurrence of P_1, j is an occurrence of P_2, and neither P_1 nor P_2 occurs between i and j.

History and Related Work

The problem is a generalization of related indexing problems. Navarro and Thankachan [NT16] studied the case where $P_1 = P_2$, and presented an $O(n \log n)$ space solution that reports the consecutive occurrences in optimal $O(m + \text{occ})$ time, where $m = |P|$ is the length of the (singular) pattern and occ is the size of the output. Kopelowitz and Krauthgamer [KK16] gave a linear-space data structure that can compute the distance of the closest occurrence of two query patterns P_1 and P_2 in $O(|P_1| + |P_2| + \sqrt{n}\log^{\epsilon} n)$ time. Note that this solves the existence version of our problem for $\alpha = 0$. They also proved a conditional lower bound for this problem, matching the upper bound up to polylogarithmic factors, which implies that the problem with two patterns is significantly harder than with one.

Results

In Chapter 3 we present two solutions for the general problem, i.e., with arbitrary query intervals $[\alpha, \beta]$. One is an $O(n \log n)$ space data structure that supports reporting queries in $O(|P_1|+|P_2|+n^{2/3}\text{occ}^{1/3} \log n \log\log n)$ time, for constant $\epsilon > 0$. The other is a linear-space data structure that supports counting queries (and by extension existence queries) in $O(|P_1| + |P_2| + n^{2/3} \log^{\epsilon} n)$ time. We also show a conditional lower bound for the existence problem, by a reduction similar to Kopelowitz and Krauthgamer's, that holds even for fixed $\alpha = 0$ and β. By extending our solution we obtain a matching upper bound, up to polylogarithmic factors, for this variant of the problem. For the general problem our lower bound leaves a polynomial gap to our upper bound.

1.2.3 Sliding Window Indexing in Streams

In this problem a string S is being streamed one character at a time, and the goal is to maintain a compact index over the w most recent characters to arrive, called the *window*. At any point the streaming of S can be interrupted by a pattern P, also streamed, and we must report all the occurrences of P in the window. For maintaining the index and answering queries we want to minimize the time spent per character that arrives from either stream. We additionally study a variant of the problem where a delay of δ characters is allowed from either stream before queries must be answered, which can be leveraged to improve indexing and query time.

History and Related Work

The problem is closely related to several well-studied indexing problems, though we introduced this specific formulation. There are several results on maintaining the suffix tree over a sliding window, which can solve this problem. However, the best of these solutions, by Brodnik and Jekovec [BJ18], spends constant *amortized* time per character that arrives, and only for constant-sized alphabets. There are also several results for online string indexing, where the goal is to incrementally construct a string index. The best of these solutions achieve either constant time per character for constant-sized alphabets [KN17], or $O(\log\log n + \log\log|\Sigma|)$ time for general alphabets [Kop12]. Both rely on processing the string from right to left, and are therefore not applicable in a streaming context. It is also unclear if they can be adapted to a sliding window, i.e.,

use space proportional to the window and not the entire substring so far processed. Recent work on fully dynamic suffix arrays [SLLM10, AB20, AB21, KK22] could also solve the problem, but due to their generality have polylogarithmically slower bounds and are significantly more complicated.

Results

In Chapter 4 we present an $O(w)$ space data structure that spends $O(\log w)$ time per character that arrives from either stream, with high probability. Each reported occurrence of a pattern takes additional worst-case constant time. For the delayed problem variant we present an $O(w)$ space solution that uses $O(\log(w/\delta))$ time per character that arrives, with high probability. In particular, for $\delta = \epsilon w$ and constant $0 < \epsilon < 1$, we obtain a linear-space data structure that spends constant time per character that arrives. To obtain our results we introduce a novel, hierarchical structure for suffix trees that might be of independent interest, inspired by the classic log-structured merge trees [OCGO96b].

1.3 Compression

Chapters 5 and 6 study compression problems related to strings and string indexing. Specifically, we study the following problems:

- **Rightmost Lempel-Ziv**: An *LZ-like factorization* of a string S is a segmentation of S into phrases $f_1, \ldots, f_z$, where each phrase f_k is either the first occurrence of a character in S or a prefix of $f_k \ldots f_z$ that occurs at least twice in $f_1 \ldots f_k$. Given an LZ-like factorization $S = f_1 \ldots f_z$, compute for each phrase f_k the rightmost previous occurrence of f_k.
- **Compression of Deterministic Finite Automata**: Given a deterministic finite automaton, construct a *delayed* deterministic finite automaton with the same language and fewer total transitions.

1.3.1 Rightmost Lempel-Ziv

For a string S, the *LZ77 factorization* [LZ76] is a segmentation of S into z phrases $S = f_1 \ldots f_z$, such that each phrase f_k is either the first occurrence of a character or the longest prefix of $f_k \ldots f_z$ that occurs at least twice in $f_1 \ldots f_k$. The factorization can be compressed by encoding each phrase f_k as the pair $(d_k, |f_k|)$, where d_k is the distance to a previous occurrence of f_k. To maximize compression it is beneficial if each d_k is minimal, i.e., points to the rightmost previous occurrence of f_k, in which case it is a *rightmost parsing*. *LZ-End* [KN10, KN13] is variant of the classic LZ77 factorization, where each phrase f_k must reference a previous occurrence that is aligned to a phrase boundary, i.e., there must exist a $k' < k$ such that f_k is a suffix of $f_1 \ldots f_{k'}$. It has applications to string indexing, as accessing the compressed text is made significantly easier by this additional restriction. Notably, and unlike LZ77, the LZ-End factorization obtained by a greedy algorithm is not necessarily optimal.

History and Related Work

LZ77 parsings are well-studied and there are several linear-time algorithms using only $O(n \log \sigma)$ bits of working space (e.g. [FIKS18]), where σ is the size of the alphabet and $n = |S|$ is the length of the uncompressed text. There are fewer results on computing the rightmost LZ77 parsing, and no linear-time solutions. There are two algorithms using $O(n \log n)$ time and bits of space [ALU02, Lar14]. Ferragina et al. [FNV13] presented a faster algorithm, using $O(n(1+\log \sigma / \log\log n)))$ time and $O(n \log n)$ bits of space. The current best is by Belazzougui and Puglisi [BP16], using only $O(n \log \sigma)$ bits of space and $O(n(\log\log \sigma + \log \sigma/\sqrt{\log n}))$ deterministic time or $O(n(1 + \log \sigma/\sqrt{\log n}))$ time with randomization. The greedy LZ-End parsing can be computed in linear time, shown by Kempa and Kosolobov [KK17b], and also in $O(z + \ell)$ space [KK17a]. Bannai et al. [BFK+23] recently proved that computing the optimal LZ-End parsing (with minimal number of phrases) is NP-hard. There are no previous results on computing rightmost LZ-End parsings, to the best of our knowledge.

Results

In Chapter 5 we show that given any LZ-End factorization we can compute its rightmost parsing in $O(n + z\sqrt{\log n})$ time and $O(n \log n)$ bits of space, or $O(n)$ time if the factorization is greedy LZ-End. We also show how to solve several special cases of general rightmost LZ77. Given any LZ-like factorization, we can in linear time resolve (find the rightmost previous occurrence of) any subset of $O(n^\epsilon)$ distinct phrases, all phrases that are within $O(\log n)$ characters of a previous occurrence, and all phrases that occur at most $O(\log n)$ times in the factorization. Additionally, we can resolve all phrases of length $\Omega(\log^{6.66} n / \log^2 \sigma)$ in sublinear $O(n/\log_\sigma n)$ time and space, using similar techniques as Belazzougui and Puglisi [BP16] used to resolve all phrases of length $\Omega(\log^5 n)$ in $O(n)$ time and $O(n/\log_\sigma n)$ space.

1.3.2 Compression of Deterministic Finite Automata

Regular expressions are often converted into deterministic finite automata (DFA) to allow fast language membership queries, i.e., determine if the pattern P is in the language of the regular expression. The primary downside of DFAs is their space usage, and it is therefore interesting to compress them while maintaining their fast query speeds. A classic result in this lineage is the *delayed deterministic finite automaton* (D^{2}FA). By augmenting the DFA with unlabeled *default transitions*, it allows the removal of many labeled transitions without affecting the language of the automaton. Given a DFA, the goal in this problem is to efficiently construct a D^{2}FA that has the same language but is significantly smaller.

History and Related Work

Compression of DFAs is well-studied (see e.g. surveys [XCS+16, PS17]). Kumar et al. [KDY+06] introduced the *deterministic finite automaton* (D^{2}FA), and this technique forms the basis of several subsequent solutions [MPN+10, PLT14, LT14, BC13, KTW06, BC07b, LSL+17, MMK18, MKMK18, GWX+23]. They presented an $O(n^2(\log n + |\Sigma|))$ time construction algorithm, where n is the number of states in the DFA and Σ is the alphabet, along with a cubic-time algorithm for constructing D^{2}FAs with specifiable worst-case query time bounds. Becchi and Crowley [BC07b, BC13] presented an alternative construction algorithm for D^{2}FAs with worst-case query time bounds, running in quadratic time. Locality-sensitive hashing has been used extensively for compression for collections [DI03, OMST02, DAS10, KDLT04, PWZ11, SHWH12, KH15, XJFH11, BGPT23], however, to the best of our knowledge we are the first to use it for DFA compression.

Results

In Chapter 6 we present a simple, general framework for efficiently compressing DFAs using the D^{2}FA approach, enabling fast compression times for the class of algorithms utilizing default transitions. We apply it to obtain several fast compression algorithms, and experimentally evaluate them on real-world data sets from modern intrusion detection systems. We speed up the quadratic-time algorithm of Kumar et al. [KDY+06] to near-linear time, and in our experiments achieve an order of magnitude improvement in running time without loss of compression. We present an alternative time algorithm to the cubic-time algorithm by Kumar et al. [KDY+06] for bounding query time. Experimentally it is up to 50 times faster while achieving significantly better compression for the same query bound. Finally we speed up the similar algorithm by Becchi and Crowley [BC07b, BC13] from quadratic to linear time, which in our experiments is up to 37 times faster with only minor loss of compression.

1.4 Techniques

In this section we highlight some of the common techniques we apply to obtain our results in Chapters 2-6.

1.4.1 Segmentation

A basic technique we use repeatedly is segmenting the string, or equivalently, sampling text positions. Combined with precomputation, this can allow for efficient solutions by separate handling of short and long patterns. If occurrences of a pattern can be restricted to necessarily span certain text positions, finding them becomes significantly easier. This is widely used in string problems, and a version of it (albeit an advanced one) has recently been employed to obtain an optimal *longest common extension* (LCE) data structure in the word RAM model. For a string S, a longest common extension query asks: "given positions i, j, what is the maximum ℓ such that $S[i..i+\ell] = S[j..j+\ell]$?" Kempa and Kociumaka [KK19] presented a data structure with constant-time queries and $O(n/\log_{|\Sigma|} n)$ space and construction time. A key idea for their construction is *string synchronizing sets* [KK19,Koc19] (similar, but independently developed techniques include *partitioning sets* [BGP20b], *minimizers* [RHH+04], and *winnowing* [SWA03]), which is a text sampling technique they combine with precomputation to answer queries that have sufficiently long anwers.

Belazzougui and Puglisi [BP16] used regular text sampling to find previous occurrences of LZ77 phrases longer than a parameter δ. In Chapter 5 we build on their technique to solve the same problem in sublinear time, leveraging the LCE data structure of Kempa and Kociumaka. We sample every δth text position, for $\delta = \Omega(\text{polylog}\, n)$, akin to sampling the boundary positions of segments of size δ. Any phrase longer than δ must then span a sampled position. We reduce the problem of finding rightmost previous occurrences spanning one of the $O(n/\delta)$ sampled positions to a geometric problem (see Section 1.4.3) that we can solve in sublinear $O(n/\log_{|\Sigma|} n)$ time.

In Chapter 4 we segment the string with exponentially decreasing segment sizes. This allows us to quickly index new characters as they arrive in the stream, as each new segment to be indexed is small. This technique can be seen as a novel adaptation of the classic *log-structured merge-trees* by O'Neil et al. [OCGO96a] to text indexing. When answering pattern matching queries, we use our constructed indices to find occurrences that span at most a single segment. Any remaining occurrences are all close to the right window boundary, due to the geometrically decreasing segment sizes, and we find them in optimal $O(m)$ time by searching only the $O(m)$ rightmost characters of the window, where m is the length of the pattern.

In Chapter 3 we use segmentation to find consecutive occurrences that are far apart. We segment the string regularly and observe that consecutive occurrences (pairs of occurrences) that are further apart than our segment length must span at least one segment boundary. We find such pairs using orthogonal range successor queries (see Section 1.4.3), by searching for occurrences in both directions from each boundary. We find the remaining (close) consecutive occurrences using precomputed information.

1.4.2 Decomposition of Suffix Trees

A fundamental construct in stringology is the *suffix tree*, and we heavily utilize it to achieve our results in Chapters 2-5. For a set of prefix-free strings (no string in the set is a prefix of another string in the set), a *trie* is a rooted tree with single-character edge labels, such that (1) concatenating the edge labels of each root-to-leaf paths yields the set of strings, (2) common prefixes of strings maximally share root-to-leaf paths, and (3) the outgoing edges from each node are arranged in the lexicographic order of their labels. To obtain a *compact trie*, all non-branching paths are contracted into a single edge. For a string S over alphabet Σ, the *suffix tree* of S is the compact trie over the suffixes of $S\$$, where $\$ \notin \Sigma$ is special terminator symbol lexicographically smaller than all characters in the alphabet. The suffix tree can be stored in linear space by replacing the edge labels with pointers into S. It can be constructed in linear time for constant-sized alphabets [Wei73, Ukk95], and in the word RAM model even for polynomially-sized alphabets [FFM00, KSB06], i.e., $|\Sigma| = O(n^c)$ for constant c.

The suffix tree supports fast pattern matching, i.e., finding the positions in S where a pattern P occurs. This is done by finding the minimum-depth node v where P is a prefix of the concatenation of the edge labels of the root-to-v path. We call v the *locus* of P in the suffix tree, and the leaves below the locus correspond to the occurrences of P in S. Naively this takes $O(m \log |\Sigma|)$ time (by binary searching the outgoing edges in each node), where $m = |P|$. The suffix tree can be augmented with e.g. FKS perfect hashing [FKS84] to improve the matching time to optimal worst-case $O(m)$ time, though this requires

expected linear preprocessing time. To obtain our results in Chapters 2-3 we *decompose* the suffix tree in a variety of ways to obtain properties we can exploit for indexing.

To obtain our $O(n \log n)$ space solution for the top-k consecutive occurrences problem in Chapter 2, we use the *heavy path decomposition* [ST83]. It decomposes a rooted tree of size n into a set of disjoint *heavy paths*, with the property that any root-to-leaf path intersects at most $O(\log n)$ heavy paths. We apply this to the suffix tree and construct geometric data structures (see Section 1.4.3) for each heavy path that encode the consecutive occurrences of the substrings corresponding to nodes on the path. We show that each leaf contributes a constant amount of space to the data structure of each heavy path above it, so by the heavy path property the total space contributed by all leaves is $O(n \log n)$.

In Chapters 2 and 3 we apply a *clustering decomposition* [AHdLT97, AHT00, AR02, Fre97] to the suffix tree. For a tree of size n and a positive integer parameter τ, it decomposes the tree into $O(n/\tau)$ connected subgraphs, called *clusters*, each containing at most τ nodes. Any pair of clusters overlap in at most a single node (called a *boundary node*), and each cluster either has one boundary node (*leaf clusters*), or two boundary nodes (*path clusters*). We call the unique path between the two boundary nodes of a path cluster a *spine*. For any tree the decomposition can be constructed in linear time (see Lemma 3). We obtain several solutions by applying this decomposition and precomputing information for each cluster. When answering a query for a pattern P, we find the locus of P in the suffix tree and the cluster that contains it. We then answer using the precomputed information along with the set of leaves contained in the cluster, exploiting that there are at most τ such leaves.

In Chapter 2 we introduce a *recursive cluster decomposition* to obtain a linear space solution for the top-k consecutive occurrences problem. We decompose the suffix tree into clusters of size $\tau_1 = \sqrt{n}$, and recursively cluster the off-spine subtrees with $\tau_{i+1} = \sqrt{\tau_i}$. When answering a query, either the locus of the pattern is on a spine with enough precomputed information, or all the leaves below the locus are contained in a cluster of size k^2. This yields a $O(n \log \log n)$ space and $O(m + k^2)$ time solution. By generalizing the cluster sizing and appropriately encoding the data structures we obtain a linear-space and $O(m + k^{1+\epsilon})$ time solution. To obtain our $O(n + k \log^{1+\epsilon} n)$ time solution we use similar techniques, though we now compute $\log n$ non-recursive cluster decompositions and use different geometric queries (orthogonal range successor).

In Chapter 3 we introduce the *induced suffix tree decomposition*. For a substring of S we define its *induced suffix tree* to be the tree obtained by removing all leaves in the suffix tree of S that correspond to suffixes starting outside of the substring, and then contracting all non-branching paths. The induced suffix tree decomposition is then a balanced binary tree, called the *decomposition tree*, where the nodes at height i are the induced suffix trees over the size 2^i segments of S. We use the induced suffix tree decomposition to leverage our solution for existence queries to answer reporting queries. From the root in the decomposition tree we report close consecutive occurrences using precomputed information, and far consecutive occurrences using segmentation. We then use our existence solution to determine if there are further occurrences in the left and right half of the subproblem, i.e., the left and right children in the decomposition tree, and recurse to report those if there are.

1.4.3 Geometric Data Structures

Common to the problems we study in Chapters 2-5 is searching for occurrences of a pattern with restrictions on their position in the text. In Chapter 2 we search for pairs that are close together, in both Chapters 2 and 3 we find pairs of occurrences that are consecutive, in Chapter 4 we search for pairs in a specific part of the string (the window), and in Chapter 5 we are looking for previous occurrences, i.e., occurrences to the left of a given text position. These kinds of searches often reduce elegantly to geometric problems. A tool we typically leverage for this reduction is the *suffix array* [MM93a]. The suffix array of S is the suffixes of S in lexicographic order and represented by their starting positions, i.e., the ith element is the starting position of the ith lexicographically least suffix. Equivalently, it is also the leaves of the suffix tree of S in left-to-right order, and we can obtain one from the other in linear time.[1] The lexicographic ranks of the

[1] Obtaining the suffix tree from the suffix array in linear time requires the *longest common prefix* array, also computable in linear time. [KLA+01].

suffixes of S act as the "dimension" in our geometric queries that represent the pattern. Any occurrence of a pattern P in S will correspond to a suffix of S that P is a prefix of. The lexicographic ranks of all such suffixes form a contiguous range, and their starting positions can be read from the suffix array.

In Chapters 2 and 3 we use this to locate consecutive occurrences via *orthogonal range successor* queries. For an integer array A, an orthogonal range successor query asks: "given (a, b, c), what is the minimum element $c' > c$ in $A[a, b]$?" Geometrically, this can be viewed as a three-sided query in a two-dimensional grid: "among the points $\{(i, A[i]) \mid 1 \leq i \leq |A|\}$, what is the lowest point above the horizontal line at c and between the vertical lines at a and b?" Given a pattern P and a text position i in S, orthogonal range successor queries on the suffix array allows us to locate the first occurrence of P after i. In the lexicographic range of the suffixes prefixed by P we search for the first text position after i. We use this extensively to locate consecutive occurrences, given either component. Several efficient data structures exist for the problem, and we use both the linear-space and $O(\log^{\epsilon} n)$ time data structure by Nekrich and Navarro [NN12], and the $O(n \log \log n)$ space and $O(\log \log n)$ time data structure by Zhou [Zho16].

In Chapter 2 we reduce the problem of finding close consecutive occurrences to orthogonal line segment intersection, which is defined as: preprocess a set of vertical line segments into a data structure such that given a horizontal line at height y, return the k leftmost line segments intersecting the line. For paths in the suffix tree (either heavy paths or cluster spines), we encode the consecutive occurrences of substrings that correspond to nodes on the path. Each consecutive occurrence becomes a vertical line segment, with the distance of the consecutive occurrence as the first coordinate. After preprocessing, if the locus of the pattern is on a path, we can in optimal $O(k)$ time find the k closest consecutive occurrences of the pattern.

In Chapter 4 we use *range maximum queries* to locate the rightmost occurrences of a pattern. For an integer array A, range maximum queries ask: "what is largest element in $A[a, b]$?" Using the classic solution by Garbon et al. [GBT84] we preprocess the suffix array to answer range maximum queries in constant time. Again using the lexicographic range of suffixes prefixed by P, we can in constant time obtain the rightmost occurrence of P. We recursively query to obtain all the occurrences in approximately right-to-left order, allowing us to report only the occurrences inside the window in optimal time (after obtaining the lexicographic range).

In Chapter 5 we find previous occurrences of long phrases in sublinear time, by reducing to three-dimensional range queries. We note this technique is similar to the one employed by Belazzougui and Puglisi [BP16] to find previous occurrences of long phrases in $O(n)$ time. We regularly sample text positions and for each sampled position i we construct a three-dimensional point of i, the lexicographic rank of the suffix starting at i, and the co-lexicographic rank of the prefix ending at i. We use the aforementioned LCE data structure of Kempa and Kociumaka [KK19] to compute the (co-)lexicographic ranks for all the points in sublinear time by comparison sorting. As each occurrence of a long phrase must cross a sampled text position, we can find the rightmost previous ones using geometric queries on the point set. We store the points in a recent three-dimensional orthogonal range searching data structure by Chan and Tsakalidis [CT18] and resolve all the long phrases in $O(n/\log_{|\Sigma|} n)$ time.

1.5 Open Questions

The most interesting open problem to me is computing the rightmost Lempel-Ziv parsing in linear time, or proving a lower bound. In Chapter 5 we show how to solve several special cases in linear, or even sublinear, time. The remaining case is factorizations with more than $z = O(n^{\epsilon})$ phrases, for any constant $0 < \epsilon < 1$, and then only the phrases that are simultaneously shorter than ℓ, are $\omega(\log n)$ characters from their rightmost source and appear $\omega(\log n)$ times in the factorization. Notably, our results hold for any LZ-like parsing and thus do not exploit the greedy construction of LZ77 phrases. It seems plausible that this could be a direction for closing the gap to linear time, if it is possible. If that is not necessary, i.e., linear time is achievable for any LZ-like factorization, that is arguably even more interesting.

Another open problem is extending the functionality of our sliding window index, presented in Chapter 4. We support reporting queries (and existence via reporting) but not counting. When reporting, we use the time we spend to report each occurrence to also filter any occurrences that are outside the window. Unfortunately,

this is not applicable when answering counting queries. Similarly, it would be interesting to support fast longest common extension queries. This is a classic augmentation to the suffix tree, via lowest common ancestor data structures, but this breaks down when the string is segmented. Lastly, improving the update time while retaining fast queries would be exciting.

Finally, in Chapter 3 we provide an upper bound and a lower bound for the gapped indexing for consecutive occurrences problem, with a polynomial time gap between them. We almost match the lower bound for one-sided intervals, but the general problem seems much harder. It is unclear if our techniques can extend to close this gap, or a different approach is required.

Chapter 2

String Indexing for Top-k Close Consecutive Occurrences

String Indexing for Top-k Close Consecutive Occurrences

Philip Bille phbi@dtu.dk
Inge Li Gørtz inge@dtu.dk
Max Pedersen mhrpe@dtu.dk
Eva Rotenberg erot@dtu.dk
Teresa Anna Steiner terst@dtu.dk

June 30, 2023

Abstract

The classic string indexing problem is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P, report all occurrences of P within S. In this paper, we study a basic and natural extension of string indexing called the string indexing for top-k close consecutive occurrences problem (Sitcco). Here, a consecutive occurrence is a pair (i, j), $i < j$, such that P occurs at positions i and j in S and there is no occurrence of P between i and j, and their distance is defined as $j - i$. Given a pattern P and a parameter k, the goal is to report the top-k consecutive occurrences of P in S of minimal distance. The challenge is to compactly represent S while supporting queries in time close to the length of P and k. We give three time-space trade-offs for the problem. Let n be the length of S, m the length of P, and $\epsilon \in (0, 1]$. Our first result achieves $O(n \log n)$ space and optimal query time of $O(m + k)$. Our second and third results achieve linear space and query times either $O(m+k^{1+\epsilon})$ or $O(m+k\log^{1+\epsilon} n)$. Along the way, we develop several techniques of independent interest, including a new translation of the problem into a line segment intersection problem and a new recursive clustering technique for trees.

2.1 Introduction

The classic string indexing problem is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P, report all occurrences of P within S. An *occurrence* of P within S is an index i, $0 \leq i < |S|$, such that $P = S[i \dots i + |P| - 1]$. In this paper, we introduce a basic extension of string indexing, where the goal is to report *consecutive occurrences* of the pattern P that occur *close* to each other in S. Here, a consecutive occurrence is a pair (i, j), $i < j$, such that P occurs at positions i and j in S and there is no occurrence of P between i and j, and close to each other means that the distance $j - i$ between the occurrences should be small. More precisely, given a pattern P and an integer parameter $k > 0$, define the *top-k close consecutive occurrences* of P to be the k consecutive occurrences of P in S with the smallest distances. Given a string S the *string indexing for top-k close consecutive occurrences* (Sitcco) problem is to preprocess S into a data structure that supports top-k close consecutive occurrences queries. The goal is to obtain a compact data structure while supporting fast queries in terms of the length of the pattern P and the number of reported occurrences k. For an example, see Figure 2.1.

Surprisingly, the Sitcco problem has not been studied before even though it is a natural variant of string indexing and several closely related problems have been extensively studied (see related work below).

$P =$ AN

$S =$ BATMAN AND ANNA SING NANANANA AND EAT BANANAS
(positions 0, 5, 10, 15, 20, 25, 30, 35, 40)

Figure 2.1: P occurs at positions 4, 7, 11, 22, 24, 26, 30, 39 and 41 in S. The top 5 close consecutive occurrences are $(22, 24)$, $(24, 26)$, $(39, 41)$, $(4, 7)$, and $(7, 11)$, with the tie between $(7, 11)$ and $(26, 30)$ broken arbitrarily.

2.1.1 Results and Techniques

To state the complexity bounds, let n and m denote the lengths of S and P, respectively. An immediate approach to solve the Sitcco problem is to store the suffix tree of S using $O(n)$ space. To answer a query on P with parameter k, we traverse the suffix tree to find *all* occurrences of P, construct the consecutive occurrences, and then sort these to output the top-k close consecutive occurrences. Naively, this requires two sorts of size occ, where occ is the total number of occurrences of P, giving a query time of $O(m+\text{occ}\log\text{occ})$. Using more advanced data structures [BFGLO09, BFP+73], the query time can be reduced to $O(m + \text{occ})$ while still using linear space. Note that occ can be much larger than k. Alternatively, we can store at every node in the suffix tree the set of all consecutive occurrences sorted by distance using $O(n^2)$ space. To answer a query we find the node corresponding to P and simply report the first k of the stored consecutive occurrences in optimal $O(m + k)$ time.

To achieve better trade-offs, one might try to use a strategy similar to range minimum query (RMQ), where the ranges are subsequent ranges in the suffix array and the values are distances between pairs of suffix indexes in S. However, there are several problems with that idea: first, there are $\Theta(|S|^2)$ possible pairs of suffix indexes within S, and it is not immediately clear how many of them can correspond to *consecutive* occurrences of a pattern (our arguments from Section 2.3 imply that this number is bounded by $O(n \log n)$). Secondly, when taking the union of two ranges, the set of closest (consecutive) pairs can change completely: consider for example the string $S =$ A B A C A B A C D A B D A C D A B D A C. While the string A has occurrences $\{0, 2, 4, 6, 9, 12, 15, 18\}$, the string AB has occurrences $\{0, 4, 9, 15\}$ and AC has $\{2, 6, 12, 18\}$. Note that for $P =$ A, the top-3 consecutive occurrences are $(0, 2)$, $(2, 4)$ and $(4, 6)$, while for AB they are $(0, 4)$, $(4, 9)$ and $(9, 15)$ and for AC they are $(2, 6)$, $(6, 12)$ and $(12, 18)$. Both the pairs and the distances are completely different between A and its extensions. Thus, there is an issue of non-decomposability, which is a main challenge in this particular problem. However, in the rest of our paper we will show that we can use suffix tree decompositions and amortized arguments to bound the number of changes that can happen in the set of consecutive occurrences of substrings corresponding to positions on some paths in the suffix tree.

We obtain the following significantly improved time-space trade-offs:

Theorem 1. *Given a string S of length n and ϵ, $0 < \epsilon \leq 1$, we can build a data structure that can answer* top-k close consecutive occurrences queries *using either*

(i) $O(n \log n)$ *space and* $O(m + k)$ *query time or*

(ii) $O(\frac{n}{\epsilon})$ *space and* $O(m + k^{1+\epsilon})$ *query time.*

(iii) $O(\frac{n}{\epsilon})$ *space and* $O(m + k \log^{1+\epsilon} n)$ *query time.*

Here, m is the length of the query pattern.

Hence, Theorem 1(i) achieves optimal query time using near-linear space. Alternatively, Theorem 1(ii) and (iii) achieve linear space, for constant ϵ, while supporting queries in near-optimal $O(m + k^{1+\epsilon})$ and $O(m + k \log^{1+\epsilon} n)$ time, respectively.

To achieve Theorem 1 we develop several data structural techniques that may be of independent interest. First, we translate the problem into a line segment intersection problem on the heavy path decomposition

of the suffix tree. This leads to the $O(n \log n)$ space and optimal query time bound of Theorem 1(i). We note that Navarro and Thankachan [NT16] used similar techniques for a closely related problem (see related work below). To reduce space, we introduce a novel recursive clustering method on trees. The decomposition partitions the tree into a hierarchy of depth $O(\log \log n)$ consisting of subtrees of doubly exponentially decreasing sizes. We show how to combine the decomposition with the techniques of the simple algorithm from Theorem 1(i) to obtain an $O(n \log \log n)$ space and $O(m + k^{1+\epsilon})$ query time solution. Then, we show how to efficiently compress the hierarchy of data structures into rank space leading to the linear space and $O(m + k^{1+\epsilon})$ query time bound of Theorem 1(ii). Finally, we show how to use $O(\log n)$ cluster decompositions of varying parameters together with an orthogonal range successor data structure to obtain the $O(m + k \log^{1+\epsilon})$ time bound of Theorem 1(iii).

We apply these techniques to three related problems: Firstly, we address the natural "opposite" problem of reporting the k consecutive occurrences of *largest* distance, which can be solved using similar but not identical techniques. Secondly, we apply our framework to the related problem of reporting consecutive occurrences with distances within a specified interval, considered by Navarro and Thankachan [NT16], and give an improvement for a special case. Finally, we show how this allows us to efficiently report all non-overlapping consecutive occurrences of a pattern.

2.1.2 Related Work

To the best of our knowledge, the Sitcco problem has not been studied before, even though distances between occurrences is a natural extension for string indexing and several related problems have been studied extensively.

A closely related problem was considered by Navarro and Thankachan [NT16], who showed how to efficiently report consecutive occurrences with distances within a specified interval. They gave an $O(n \log n)$ space and $O(m + \text{occ})$ time solution, where occ denotes the number of reported consecutive occurrences. We note that their result can be adapted to the Sitcco problem to achieve the same bounds as in Theorem 1(i). However, our solution is simpler and does not rely on heavy word RAM techniques such as persistent van Emde Boas trees [Cha13]. Our techniques can also be used to solve the problem considered by Navarro and Thankachan getting the same space and time bounds as they obtain, and we can achieve improved bounds in a special case (see Section 2.8).

A lot of work has been done on the related problem of string indexing for patterns under various *distance constraints*, where the goal is to report occurrences of (one or more) patterns that are within a given distance or interval of distances of each other [BG14, IR09, BGP16, CPZ20, BGVV14, Lew11, KKL07]. An important difference between those works and our work is that all those solutions use time proportional to *all* pairs of occurrences with distances in the given range, in contrast to only finding *consecutive* occurrences. Note that if the goal is to find occurrences of a given maximal distance, one can find the close *consecutive* occurrences first and then construct all pairs satisfying the constraint.

Another line of related work is indexing collections of strings, called *documents*. Here the goal is to find documents containing patterns subject to various constraints. For a comprehensive overview see the survey by Navarro [Nav14]. Several results on supporting efficient top-k queries are known [MNST20, SSTV13, HSTV14, BGST18, HPSW10, HSTV14, NN17, Tsu13, HPS$^+$13, HTSV13, NT14, MNN$^+$17]. In this context the goal is to efficiently report the k documents of smallest weight. The weights can depend on the query and can be the distance between the closest pair of occurrences of a given pattern [HSTV14, NN17, SSTV13, HSTV14, NN17, MNN$^+$17]. The problem can be solved in linear space and optimal $O(k)$ time, in addition to finding the locus of the pattern in the suffix tree [SSTV13]. While this problem statement resembles ours, there is no direct translation from those results to our problem, since the documents are considered individually, and for a single document only the pair of occurrences with minimum distance within the document is considered.

Finally, we note that since the initial publication of the results in this paper, a subset of the authors have recently considered indexing for consecutive occurrences of two different patterns P_1 and P_2 [BGPS21].

2.1.3 Outline

The paper is organized as follows. In Section 2.2 we introduce some notation and recall results on string indexing. In Section 2.3 we build a simple data structure and prove Theorem 1(i). In Section 2.4 we recall a method for tree clustering and show how to use it to solve a simplified version of the problem. In Section 2.5 we introduce a recursive clustering method that allows us to use the ideas from Section 2.4 on the actual problem. This gives an $O(n \log \log n)$ space and $O(m + k^2)$ time data structure. In Section 2.6, we show how to reduce the space to linear while achieving the same query time, and then generalize the recursion to get Theorem 1(ii) for any $0 < \epsilon \leq 1$. In Section 2.7 we give the linear space solution with $O(m + k \log^{1+\epsilon} n)$ query time. Finally, in Section 2.8 we apply our techniques to related problems.

2.2 Preliminaries

We introduce some notation and recall basic results from string indexing.

A *string* S of length n is a sequence $S[0]S[1] \dots S[n-1]$ of characters from an alphabet Σ. A contiguous subsequence $S[i, j] = S[i]S[i+1] \dots S[j-1]$ is a *substring* of S. The substrings of the form $S[i, n]$ are the *suffixes* of S.

The *suffix tree* [Wei73] is a compact trie of all suffixes of $S\$$, where \$ is a symbol not in the alphabet, and is lexicographically smaller than any letter in the alphabet. Using perfect hashing [FKS84], it can be stored in $O(n)$ space and solve the string indexing problem (i.e., find and report all occurrences of a pattern P) in $O(m + \text{occ})$ time, where m is the length of P and occ is the number of times P occurs in S. The *suffix array* stores the suffix indices of $S\$$ in lexicographic order. The suffix tree has the property that the leaves below any node represent suffixes that appear in consecutive order in the suffix array. Brodal et al. [BFGLO09] show that there is a linear space data structure that allows outputting all entries within a given range of an array in sorted order using time linear in size of the output. This data structure on the suffix array together with the suffix tree can output all occurrences of a pattern *sorted by text order* in $O(n)$ space and $O(m+\text{occ})$ time.

For any node v in the suffix tree, we define $\text{str}(v)$ to be the string found by concatenating all labels on the path from the root to v. The *locus* of a string P, denoted $\text{locus}(P)$, is the minimum depth node v such that P is a prefix of $\text{str}(v)$.

2.3 A Simple $O(n \log n)$ Space Solution

In this section, we present a simple solution that solves the Sitcco problem in $O(n \log n)$ space and $O(m+k)$ query time. This solution will be a key component in our more advanced structures in the following sections. We note that the results by Navarro and Thankachan [NT16] for the related problem of reporting consecutive occurrences with distances within a specified interval can be modified to achieve the same complexities. However, our solution is simpler and does not rely on heavy word RAM techniques such as persistent van Emde Boas trees [Cha13].

Let $D(v)$ denote the set of consecutive occurrences of $\text{str}(v)$. Naively, if we store for each node v the set $D(v)$ in sorted order, we can directly answer a query for the top-k close consecutive occurrences of a pattern P by reporting the k smallest elements in $D(\text{locus}(P))$. This solves the problem in $O(n^2)$ space and $O(m + k)$ query time. The main idea in our simple solution is to build a heavy path decomposition of the suffix tree and compactly represent sets on the same path via a reduction to the orthogonal line segment intersection problem while maintaining optimal time queries. This is similar to the data structure by Navarro and Thankachan [NT16], but our reduction is different.

Heavy path decomposition A *heavy path decomposition* of a tree T is defined as follows: Starting from the root, at every node, we choose the edge to the child with the largest subtree as *heavy edge*, until we reach a leaf. Ties are broken arbitrarily. This defines a heavy path, and all edges hanging off the heavy path are

light edges. The root of a heavy path h is called the *apex* of the path, denoted apex(h). We then recursively decompose all subtrees hanging off the path. The heavy path decomposition has the following property:

Lemma 1 (Sleator and Tarjan [ST83])**.** *Given a tree T of size n and a heavy path decomposition of T, any root-to-leaf path in T contains at most $O(\log n)$ light edges.*

Orthogonal line segment intersection Similarly as Navarro and Thankachan [NT16], we are going to reduce the problem to a geometric problem on orthogonal line segment intersection. Specifically, we are going to reduce to the following problem: Let L be a set of n vertical line segments in a plane with non-negative x-coordinates. The *orthogonal line segment intersection problem* is to preprocess L to support the query:

- smallest-segments(y_0, k): return the first k segments intersecting the horizontal line with y-coordinate y_0 in left-to-right order.

We will assume that y_0 is an integer, which suffices for our purpose. Let N be the maximum y-coordinate of a segment in L. The following lemma follows easily from the results on partially persistent data structures by Driscoll et al. [DSST89].

Lemma 2. *We can solve the line segment intersection problem as described above in $O(n + N)$ space and $O(k)$ time.*

Proof. Consider the x-coordinates as the elements of a set X and the y-coordinate as time. The version of X at a time y_0 contains exactly the x-coordinates of the line segments which intersect the horizontal line at y_0. Now, the data structure is a partially persistent sorted doubly linked list L on the elements of X. The elements are sorted in increasing order. Since we have at most n line segments, the maximum size of X as well as the maximum number of updates is n. Each update changes only $O(1)$ pointers in the linked list. Using the node copying technique from Driscoll et al. [DSST89] we can build a partially persistent linked list using $O(n)$ space. To be able to find version y_0 in constant time, we keep an array of size N with a pointer to the root of the version at each possible time step. For a query (y_0, k), use the sorted linked list L to report the k smallest elements at time y_0.

If we use a linear scan to find the place to insert an element or find the element to be deleted we get a preprocessing time of $O(n^2)$. This can be improved to $O(n \log n)$ by using a (non-persistent) balanced binary search tree during the preprocessing holding all elements in the current version of L together with a pointer to their node in the current version. When performing an update the binary search tree is used to find the position where the element must be inserted/deleted in $O(\log n)$ time. After the preprocessing step the tree is discarded. □

2.3.1 Data Structure

We construct a heavy path decomposition of the suffix tree T of S. Our data structure consists of a line segment data structure from Lemma 2 for each heavy path of T that compactly encodes the sets $D(v)$ for each node v on the path.

We describe the contents of the data structure for a single heavy path $h = v_1, \ldots, v_\ell$, where v_1 is the apex of the path. Consider a consecutive occurrence (i, j) on some node on h and imagine moving down the heavy path from top to bottom. Either (i, j) is a consecutive occurrence at the apex of h or it will become a consecutive occurrence as soon as every suffix starting at an index between i and j has branched off the heavy path. Then it will stay a consecutive occurrence until either the suffix corresponding to i or the suffix corresponding to j (or both) branch off h. Thus, there exists an interval $[d_1, d_2]$ of depths on the heavy path such that $(i, j) \in D(v_d)$ if and only if $d \in [d_1, d_2]$. We say that (i, j) is *alive* in this interval.

We encode the consecutive occurrences by line segments in the plane which describe their distance and the interval in which they are alive along the heavy path. Conceptually, the x-coordinate in our coordinate system corresponds to the distance of a consecutive pair, and the y-coordinate corresponds to the depth on the heavy path. Now, for each consecutive occurrence (i, j), we define a vertical line segment with x-coordinate set to its distance, and y-coordinate spanning the interval $[d_1, d_2]$, where $[d_1, d_2]$ is the interval in

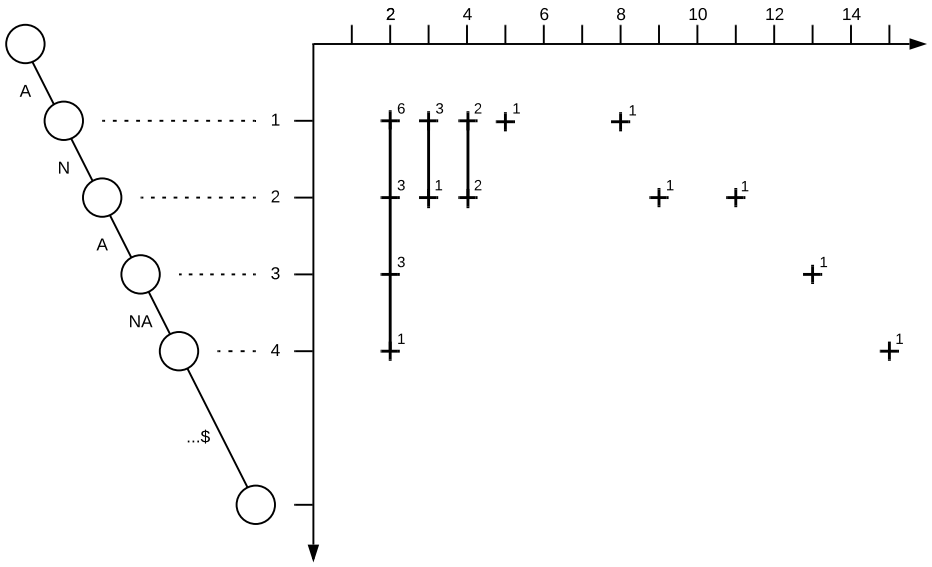

Figure 2.2: Line segments for a heavy path from the suffix tree for "`BATMAN-AND-ANNA-SING-NANANANA-AND-EAT-BANANAS`". Here, if we have overlapping line segments, we denote by a number how many consecutive occurrences the current segment corresponds to. At depth 1, we have a line segment corresponding to pairs of consecutive occurrences of string `A` - there are six pairs that have a distance of 2, three pairs that have a distance of 3, two pairs that have a distance of 4, and so on. At depth 2, we encode the consecutive occurrences of string `AN`. Some of them are the same as for string `A`.

which (i, j) is alive. For an example, see Figure 2.2. Our data structure for h stores the above line segments in the line segment data structure from Lemma 2. For each line segment in the data structure we store a pointer to the pair of occurrences it represents. The full data structure for T consists of the line segment data structures for all of the heavy paths in T.

Space analysis For a given heavy path h, a leaf in the subtree of apex(h) can be in at most two consecutive occurrences in $D(\text{apex}(h))$. Consider a light edge (v_d, u) leaving h at depth d. Any leaf in the subtree rooted at u can be part of at most two consecutive occurrences in $D(v_d)$. A single leaf can thus make at most two consecutive occurrences from $D(v_d)$ disappear in $D(v_{d+1})$ and at most one new consecutive occurrence appear. If we consider all leaves that leave h, we therefore get at most three changes per leaf. Thus, for a given heavy path h a leaf in the subtree of apex(h) can be in at most two consecutive occurrences in $D(\text{apex}(h))$ and can cause at most three changes of line segments in the line segment data structure for h. Since any root-to-leaf path can intersect at most $\log n$ heavy paths, any leaf can contribute $O(\log n)$ line segments. Overall, this means that there are at most $O(n \log n)$ line segments in total. For a single heavy path h the line segment data structure from Lemma 2 uses linear space in the number of segments and the length of h. The sum of the lengths of the heavy paths is $O(n)$, since the heavy paths are disjoint. Thus the total space usage is $O(n \log n)$.

2.3.2 Algorithm

Given a pattern P and an integer k we can now answer a query as follows. We begin by finding locus(P) in the suffix tree. Let h be the heavy path that the locus is on and let d_P be the depth of locus(P) on h. We do a smallest-segments(d_P, k) query on the line segment data structure stored for h and report the consecutive occurrences corresponding to the returned line segments.

Correctness By definition, $D(\text{locus}(P))$ contains the consecutive occurrences of P. Thus, every consecutive occurrence of P defines a line segment in the data structure for h and the horizontal line with y-coordinate set to d_P intersects exactly those line segments. Since we set the x-coordinate of every line segment to the distance of its consecutive occurrence, the line segments are sorted left-to-right by increasing distance. Thus, the first k line segments intersecting the horizontal line at $y = d_P$ correspond to the top-k close consecutive occurrences.

Time analysis The time for finding locus(P) in the suffix tree is $O(m)$. The time for querying the line segment data structure from Lemma 2 is $O(k)$, so the total time complexity is $O(m + k)$. This proves Theorem 1(i).

2.4 A Linear Space Solution for Fixed k

In this section, we present a linear space and $O(m + k)$ time solution for the simpler problem where k is known at construction time. That is, given a string S and a positive integer k, we preprocess S into a compact data structure such that given a pattern string P, we can efficiently find the top-k close consecutive occurrences of P in S. This data structure demonstrates one of the key ideas that our final result builds on.

The main idea behind the data structure is to store the line segment solution from Section 2.3 for some path segments of the suffix tree, such that all nodes that are not on these paths are within small subtrees. For nodes within such small subtrees we can find all consecutive occurrences without spending too much time. Specifically, we will partition the suffix tree into *clusters*, satisfying some properties. We are going to define this cluster partition next.

2.4.1 Cluster Partition

For a connected subgraph $C \subseteq T$, a *boundary node* v is a node $v \in C$ such that either v is the root of T, or v has an edge leaving C – that is, there exists an edge (v, u) in the tree T such that $u \in T \setminus C$. A *cluster* is a connected subgraph C of T with at most two boundary nodes. A cluster with one boundary node is called a *leaf cluster*. A cluster with two boundary nodes is called a *path cluster*. For a path cluster C, the two boundary nodes are connected by a unique path. We call this path the *spine* of C. A *cluster partition* is a partition of T into clusters, i.e. a set CP of clusters such that $\bigcup_{C \in CP} V(C) = V(T)$ and $\bigcup_{C \in CP} E(C) = E(T)$ and no two clusters in CP share any edges. Here, $E(G)$ and $V(G)$ denote the edge and vertex set of a (sub)graph G, respectively. We need the next lemma which follows from well-known tree decompositions [AHdLT97, AHT00, AR02, Fre97] (see Bille and Gørtz [BG11] Lemma 5.1 for a direct proof).

Lemma 3. *Given a tree T with n nodes and a parameter τ, there exists a cluster partition CP such that $|CP| = O(n/\tau)$ and every $C \in CP$ has at most τ nodes. Furthermore, such a partition can be computed in $O(n)$ time.*

2.4.2 Data Structure

For the suffix tree of S, we build a clustering as in Lemma 3 with parameter τ set to k to get $O(n/k)$ clusters of size at most k. For the spine of every path cluster, we build a line segment data structure similar to the one from Section 2.3. The difference is that for any depth, we only maintain the line segments that correspond to the top-k close consecutive occurrences for that depth. Let $v_1, \ldots, v_l$ denote the nodes on the

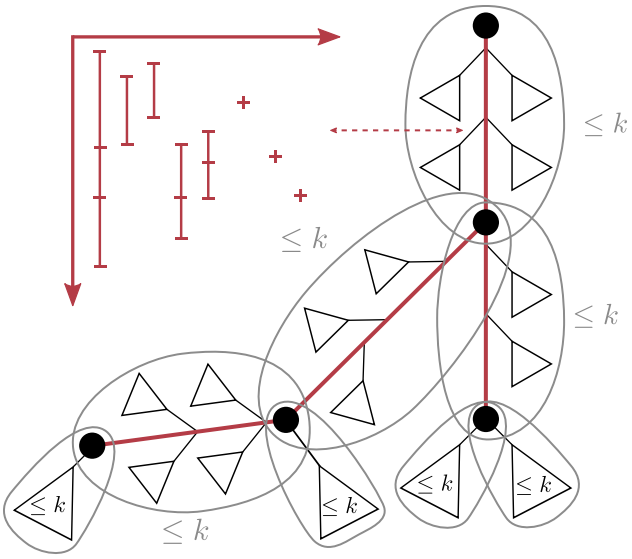

Figure 2.3: The suffix tree is divided into clusters (grey loops) of size $\leq k$ which are either leaf clusters, or path clusters with spines marked in red. For every spine we store a line segment data structure, also marked in red.

spine, starting at the top boundary node. Note that for any consecutive occurrence that appears for the first time in $D(v_{d+1})$ there is a consecutive occurrence in $D(v_d)$ of smaller distance which is no longer present in $D(v_{d+1})$. It follows that, when moving down the spine, once a consecutive occurrence (i, j) is amongst the k closest, it will stay amongst the k closest until suffix i or j branches off the spine. Thus, there exists an interval $[d_1, d_2]$ of consecutive depths such that (i, j) is amongst the k closest pairs in $D(v_d)$ if and only if $d \in [d_1, d_2]$. For a consecutive occurrence (i, j) that is amongst the k closest for any v on the spine, we define a line segment where the x-coordinate is its distance and the y-coordinate is spanning the interval $[d_1, d_2]$, where $[d_1, d_2]$ is the interval in which (i, j) is amongst the k closest pairs. For these line segments we store the data structure from Lemma 2. Again, for each line segment we store the pair of occurrences it represents. We store this data structure for the spine of each cluster and for every node that is on that spine we store a pointer to the data structure. For boundary nodes that are on multiple spines we store a pointer to any one of them. See Figure 2.3 for an illustration of this structure. Additionally we store the suffix array and the sorted range reporting data structure of Brodal et al. [BFGLO09] on the suffix array.

Space analysis We show that for every path cluster there are $O(k)$ line segments: We still have the property that a line segment only ends if a corresponding leaf branches off the spine. In that case, it might be replaced either by a new consecutive occurrence or by a consecutive occurrence that was there before but was not amongst the k closest. Note that at any node on the spine except the boundary nodes, any subtrees branching off the spine are fully contained within the cluster, and as such have total size at most k. Between the top boundary node and the next node on the spine, we have no bound as to how many leaves can branch off — however, since we only store line segments corresponding to the top-k consecutive occurrences, at most k line segments can be replaced by k other line segments. For the rest of the spine, at most k leaves can branch off in total. Every leaf that branches off can cause at most two line segments to end and two new line segments to begin. As such there can be at most $O(k)$ line segments. As the size of the line segment data structure is linear in the number of line segments and in the length of the spine, any line segment data structure of a path cluster uses $O(k)$ space. As both the sorted range reporting data structure and the suffix array have linear space complexity, the complete data structure occupies $O((n/k)k + n) = O(n)$ space.

2.4.3 Algorithm

Given a pattern P we can now answer the top-k query. We begin by finding locus(P) in the suffix tree. If the locus is on a spine, we query the line segment data structure for that spine. Otherwise the locus is either in a subtree hanging off a spine or in a leaf cluster. In both cases, there are at most k occurrences of our pattern P. We find all occurrences of P in text order, using the sorted range reporting data structure. This allows us to report the consecutive occurrences: Let $i_1, ..., i_l$ denote the leaves in text order, then the consecutive occurrences are $(i_1, i_2), (i_2, i_3), ...(i_{l-1}, i_l)$. Note that $l \leq k$, since the size of the subtree is at most k.

Correctness By construction, for any depth on a spine, the top-k close consecutive occurrences of the corresponding substring will have corresponding line segments present at that depth in the line segment data structure. If the locus is on a spine, then by the arguments in Section 2.3, the line segment data structure will report the top-k close consecutive occurrences. If the locus is not on a spine, then there are at most k occurrences of P in total, since any subtree hanging off a spine and any leaf cluster has at most k leaves. Thus, by constructing and reporting all consecutive occurrences of P we report the top-k close consecutive occurrences.

Time analysis We find the locus in $O(m)$ time. If we land on a spine we report in $O(k)$ time. Otherwise, we are in a subtree of size at most $O(k)$ and thus P has at most k occurrences. Using sorted range reporting we can find the occurrences in text order using $O(k)$ time. The total time for a query is thus $O(m + k)$.

We are going to use this data structure with different parameters in Section 2.5. For a general parameter τ, we have the following lemma:

Lemma 4. *For any positive integer τ, there exists a cluster partition of the suffix tree and a linear space data structure with the following properties:*

1. *For any $k \leq \tau$ and P such that locus(P) is on the spine of a cluster, we can report the top-k close consecutive occurrences in $O(m + k)$ time.*
2. *For any P such that locus(P) is not on a spine, we can report the top-k close consecutive occurrences in $O(m + \tau)$ time.*

Proof. We build the data structure described in this section for parameter τ taking the role of k. In case 1, we query the line segment data structure for the depth of locus(P) on the path and k. Since $k \leq \tau$ this will correctly output the top-k close consecutive occurrences of P. In case 2, we have shown that we can construct the top-τ close consecutive occurrences. Using the linear time selection algorithm by Blum et al. [BFP+73] we can find the top-k of those: We use the algorithm to find the consecutive occurrence of k^{th} smallest distance d; then we traverse all the consecutive occurrences and output those of distance $\leq d$. If needed, we crop the output to report no more than k consecutive pairs. □

2.5 An $O(n \log \log n)$ Space Solution for General k

We now show how to leverage the solution from Lemma 4 to obtain a data structure that can answer queries for any k. The idea is to recursively cluster the suffix tree, such that we always either land on a spine with a sufficient number of consecutive occurrences stored, or in a sufficiently small subtree.

2.5.1 Data Structure

Our data structure consists of the suffix tree decomposed into clusters of decreasing size, with the line segment data structure stored for every spine as before. We build it in the following way. First we build the solution from Lemma 4 with parameter $\tau_1 = \sqrt{n}$, resulting in clusters of size at most $\sqrt{n}$. For every subtree hanging off a spine and every leaf cluster, we apply the solution with parameter $\tau_2 = \sqrt{\tau_1}$. We keep recursively

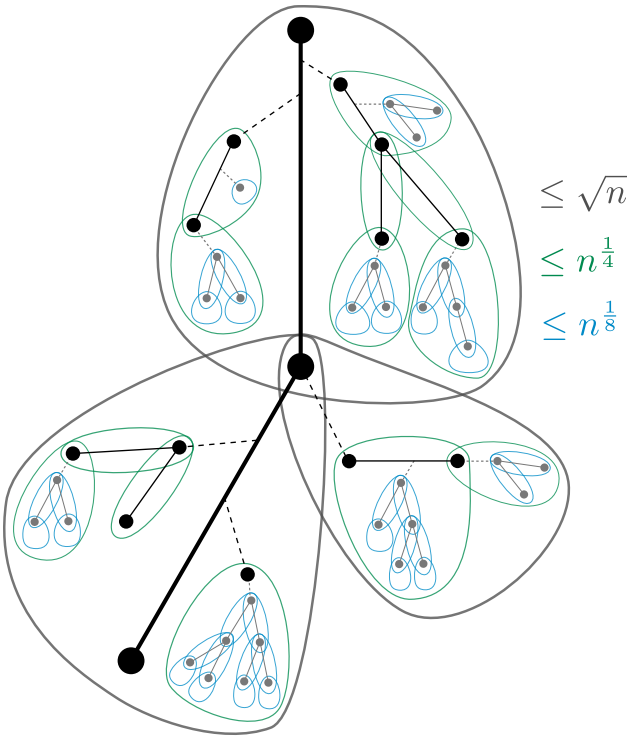

Figure 2.4: Here, we see the recursive clustering: The black clustering is the coarsest clustering and the green and blue are finer sub-clusterings.

applying the solution with parameter $\tau_i = \sqrt{\tau_{i-1}}$ until reaching a constant cluster size. For notational convenience, additionally define $\tau_0 = n$. See Figure 2.4 for an illustration of this data structure. Again we additionally store the suffix array and the sorted range reporting data structure of Brodal et al. [BFGLO09] on the suffix array.

Space analysis The suffix array and sorted range reporting structure occupy $O(n)$ space. For a tree of size $\tilde{n}$ and any τ, the data structure from Lemma 4 uses at most $O(\tilde{n})$ space. Since at every recursion level, we build the data structure from Lemma 4 on non-overlapping subtrees of the suffix tree, every recursion level uses at most $O(n)$ space. As the cluster size at every level of recursion is the square root of the previous cluster size, there are at most $O(\log \log n)$ levels. The complete data structure thus uses $O(n \log \log n)$ space.

2.5.2 Algorithm

Given a query with pattern P and parameter k, we can now answer in the following way. As before, we begin by finding the locus of the pattern in the suffix tree. This node is now either on the spine of some cluster or in a cluster of constant size. If it is on the spine of a cluster of size τ_i, and if $k \leq \tau_i$, then we query the line segment data structure for that spine, which allows us to report the top-k close consecutive occurrences. Otherwise, we find all occurrences of P and construct the top-k close consecutive occurrences by using linear time selection as in the proof of Lemma 4.

Correctness The correctness of the algorithm follows by the same arguments as previous sections.

Time analysis Finding the locus in the suffix tree takes $O(m)$ time. The locus is either on the spine of a cluster, or within a cluster of constant size. In a constant sized cluster, clearly we can do all operations

described above in constant time. If the locus is on the spine of a cluster with parameter τ_i, and $k \leq \tau_i$, then we are in case 1 of Lemma 4 with $\tau = \tau_i$ and can report the top-k close consecutive occurrences using a total of $O(m + k)$ time. If $k > \tau_i$, then we are in case 2 of Lemma 4 with $\tau = \tau_{i-1}$. Note that $\tau_{i-1} = \tau_i^2 < k^2$. Therefore, we can find the top-k close consecutive occurrences in $O(m + \tau_{i-1}) = O(m + k^2)$ time. In total, the worst case query time is then $O(m + k^2)$. In summary, this gives the following result:

Lemma 5. *Given a string S of length n, we can build a data structure that can answer* top-k close consecutive occurrences queries *using $O(n \log\log n)$ space and $O(m + k^2)$ query time. Here, m is the length of the query pattern.*

2.6 A Linear Space Solution

We now show how to reduce the space consumption of the solution presented in Section 2.5. Observe that in any cluster of level i, we only have $O(\tau_i)$ objects. If we can reduce all objects within a cluster to a "universe size" of $O(\tau_i)$ instead of $O(n)$, we can use $O(\tau_i \log \tau_i)$ bits instead of $O(\tau_i \log n)$ bits per cluster. In the following, consider a cluster C of level i.

Reducing the line segment data structure In the line segment data structure for cluster C, by the analysis of previous sections, there are at most $O(\tau_i)$ line segments and τ_i different depths on the path. Let c be a constant such that there are at most $c\tau_i$ line segments for each cluster. We map every unique x-coordinate of a line segment to a unique element in $\{1, \ldots, c\tau_i\}$ in a way that preserves order. That is, map the minimum x-coordinate to 1, the smallest x-coordinate that is bigger than the minimum to 2, and so on. This gives us a modified line segment data structure that preserves the properties we need but is restricted to a $c\tau_i \times \tau_i$ grid.

Reducing the leaf pointers For any line segment, we have to store pointers that allow us to report the corresponding pair of consecutive occurrences. Doing so naively uses $2 \log n$ bits per line segment. In the following, we show how to reduce that to $4 \log \tau_i$, for a cluster C of level i. The idea is to store the offset within the suffix array range defined by the top boundary node r of C. More precisely, let $[a_r, b_r]$ be the range in the suffix array spanning the leaves below r. Then for any leaf l in the subtree rooted at r define $\text{off}(l) = SA^{-1}(l) - a_r$. By the way our recursion is defined, C is fully contained in a subtree of size at most τ_i^2, and thus r has at most τ_i^2 leaves below it. It follows that for any leaf l in the subtree of r, $\text{off}(l)$ is a number between in $[0, \tau_i^2 - 1]$ and can be stored using $2\lceil \log \tau_i \rceil$ bits.

2.6.1 Data Structure

Our data structure is now defined as follows: We have a clustering of the suffix tree as in Section 2.5. For every spine on level i, we store the line segment data structure reduced to a $c\tau_i \times \tau_i$ grid. Every line segment corresponding to a pair (i, j) stores the pair $(\text{off}(i), \text{off}(j))$ as additional information. For every node on the spine, we store a pointer to the spine data structure and to the top boundary node of the spine. Additionally, we store the suffix array and the sorted range reporting structure, as well as two integers for every node in the suffix tree, that define the range of leaves below the node in the suffix array.

Space analysis The suffix array and the sorted range reporting data structure use space $O(n)$. Storing the range in the suffix array plus at most two pointers per node uses $O(n)$ space. For a cluster C of level i, we store the line segment data structure from Lemma 2 for a $c\tau_i \times \tau_i$ grid. Since the data structure from Lemma 2 works in the word RAM model (as do all data structures presented in this paper), we can store the data structure using $O(\tau_i \log \tau_i)$ bits. For each of the at most $c\tau_i$ line segments we store $4 \log \tau_i$ bits for the encoding of the consecutive pair. Thus, we can store the data structure for cluster C using $O(\tau_i \log \tau_i)$ bits. As in the previous section, at every recursion level, we cluster non-overlapping subtrees. The reduced

cluster solution of a subtree of size $\tilde{n}$ with parameter τ_i uses $O(\frac{\tilde{n}}{\tau_i}\tau_i \log \tau_i) = O(\tilde{n} \log \tau_i)$ bits. The total space for all clusters of level i thus becomes $O(n \log \tau_i)$. Summing over all recursion levels, we get

$$\sum_{i=0}^{\lfloor \log \log n \rfloor} O(n \log \tau_i) = \sum_{i=0}^{\lfloor \log \log n \rfloor} O\left(n \log n^{(1/2^i)}\right) = \sum_{i=0}^{\lfloor \log \log n \rfloor} \frac{1}{2^i} O(n \log n) = O(n \log n) \text{ bits},$$

that is, $O(n)$ words.

2.6.2 Algorithm

We query the data structure as follows: If we land on a spine and $k \leq \tau_i$, we query the line segment data structure and get k pairs of the form $(\text{off}(i), \text{off}(j))$. We then use the pointer to get to the root of the spine and use the range in the suffix array to translate each encoding back to the original suffix number, using constant time per leaf. Otherwise, we proceed as described in Section 2.5. Since the decoding can be done in constant time per leaf, the time complexities are the same as in Section 2.5. We have shown the following result:

Lemma 6. *Given a string S of length n, we can build a data structure that can answer* top-k close consecutive occurrences queries *using $O(n)$ space and $O(m + k^2)$ query time. Here, m is the length of the query pattern.*

In order to get Theorem 1(ii), we cluster according to a parameter ϵ, $0 < \epsilon \leq 1$, using the following recursion:

$$\tau_0 = n \text{and} \tau_i = \tau_{i-1}^{\frac{1}{1+\epsilon}} .$$

Hence, the total space in bits is now:

$$\sum_{i=0}^{\lfloor \log_{1+\epsilon} \log n \rfloor} O\left(n \log n^{1/(1+\epsilon)^i}\right) = \sum_{i=0}^{\infty} \left(\frac{1}{1+\epsilon}\right)^i O(n \log n) = \left(1 + \frac{1}{\epsilon}\right) O(n \log n),$$

that is, $O\left(\frac{n}{\epsilon} \log n\right)$ bits, so $O\left(\frac{n}{\epsilon}\right)$ words. For the query time, there are again two cases. In the case where locus(P) is on a spine with $k \leq \tau_i$, we get optimal $O(m + k)$ time, as before. For the other case, we have at most $\tau_{i-1} = \tau_i^{1+\epsilon} < k^{1+\epsilon}$ occurrences of P, which gives us a time complexity of $O(m + k^{1+\epsilon})$. This concludes the proof of Theorem 1(ii).

2.7 A Different Tradeoff

In this section we give a solution query time $O(m + k \log^{1+\epsilon} n)$. The idea is to store a finer set of cluster decompositions than in the previous section and store sublinear information for each cluster decomposition. Then we use a bounded number of orthogonal range successor queries in each cluster.

Orthogonal range successor The *orthogonal range successor problem* is to preprocess an array $A[0, \ldots, n-1]$ into a data structure that efficiently supports the following queries:

- RangeSuccessor(a, b, x): return the successor of x in $A[a, \ldots, b]$, that is, the minimum $y > x$ such that there is an $i \in [a, b]$ with $A[i] = y$.
- RangePredecessor(a, b, x): return the predecessor of x in $A[a, \ldots, b]$, that is, the maximum $y < x$ such that there is an $i \in [a, b]$ with $A[i] = y$.

Nekrich and Navarro [NN12] give a linear space data structure such that each range successor query takes $O(\log^{\epsilon} n)$ time. We will use a range successor data structure on the suffix array to answer the following type of queries: Given an index i and the suffix array range of a pattern P, find the next position in the text after i where P occurs.

2.7.1 Data Structure

We store the linear space range successor data structure from Nekrich and Navarro [NN12] and the sorted range reporting data structure by Brodal et al. [BFGLO09] on the suffix array of S. Further, we store the suffix tree of S together with the following cluster decompositions. For each $\kappa = 2, 4, 8, 16, \ldots, 2^{\lfloor \log n \rfloor}$ we build the clustering decomposition of Lemma 3 of the suffix tree for cluster size $\tau = \kappa \log n$. For each boundary node v, we store the top-κ close consecutive occurrences of $\text{str}(v)$, sorted by text position. For each node v in the suffix tree, we additionally store two bit vectors of length $\log n$. The first one is used to store for which κ node v is a boundary node, that is, the ith bit is set to 1 if v is a boundary node in the cluster decomposition with $\kappa = 2^i$ and 0 otherwise. Similarly, the other bit vector stores for which κ node v is on a spine in the cluster decomposition with $\kappa = 2^i$.

Space analysis The suffix array, the range successor data structure and the sorted range reporting data structure all uses $O(n)$ space. The suffix tree together with the bit vectors saved in the nodes also uses $O(n)$ space. By Lemma 3, there are $O(n/(\kappa \log n)$ boundary nodes in the cluster decomposition with cluster size $\kappa \log n$. For each boundary node we store κ values and thus the space used for a fixed κ is $O(\kappa n/(\kappa \log n)) = O(n/\log n)$. There are $O(\log n)$ different values of κ and therefore the total space is $O(n)$.

2.7.2 Algorithm

Given a pattern P and parameter k we can now answer the top-k query in the following way. As before, we begin by finding the locus of P in the suffix tree. Then we find the smallest power of two bigger than k, i.e. $\kappa = 2^{\lceil \log k \rceil}$, and consider the cluster decomposition defined for κ. There are two cases depending on whether $\text{locus}(P)$ is on a spine in the cluster decomposition for κ or not.

If $\text{locus}(P)$ is not on a spine, then as in case 2 of Lemma 4, there are at most $\tau = \kappa \log n$ leaves below $\text{locus}(P)$, and we can construct the top-k close occurrences in time $O(\kappa \log n)$. If $\text{locus}(P)$ is on a spine, then we find the lower boundary node b of the cluster (note we can do that by traversing the suffix tree and checking at most $O(\kappa \log n)$ nodes). We have the following property.

Claim 1. *Each of the top-k close consecutive occurrences of P is either*

1. *stored at b or*
2. *includes an occurrence of P corresponding to a leaf within the cluster.*

Proof. Any top-k consecutive occurrence of P, where both occurrences are below b, is also among the top-k consecutive occurrences of $\text{str}(b)$. This is true because P is a prefix of $\text{str}(b)$, so the occurrences of $\text{str}(b)$ is a subset of the occurrences of P. Thus any consecutive occurrence of P where both occurrences are below b is also a consecutive occurrence of $\text{str}(b)$. A consecutive occurrences of $\text{str}(b)$ that is not consecutive occurrences of P must be split by an occurrence of P that is not below b giving rise to a least two closer consecutive occurrences of P. Thus the i-closest occurrence of $\text{str}(b)$ must have distance at least the same as the i-closest occurrence of P. Every occurrence of P that is not below b is within the cluster. □

We find all occurrences of P that are within the cluster in text order using the sorted range reporting data structure. We can do this using two calls to the sorted range reporting data structure since the occurrences of P within the cluster correspond to two intervals in the suffix array, namely the range of $\text{locus}(P)$ minus the range of b. For each such occurrence i, we use an orthogonal range successor query $j = \mathsf{RangeSuccessor}(\text{range}(\text{locus}(P)), i)$ to find the next occurrence j and then an orthogonal predecessor query $i' = \mathsf{RangePredecessor}(\text{range}(\text{locus}(P)), j)$ to find the last occurrence i' before j. This gives us a consecutive occurrence (i', j). To avoid recomputing the same consecutive occurrence we skip through the list until we get to an occurrence that is after i'.

Now we have the sorted list of the top-κ consecutive occurrences of $\text{str}(b)$ stored at b, and a sorted list of all consecutive occurrences of P that include an occurrence corresponding to a leaf within the cluster. By Claim 1, any of the top-k consecutive occurrences of P is part of one of these lists. However, some of

the consecutive occurrences of str(b) might not be consecutive occurrences of P, since P might have extra occurrences inbetween. Let (i, j) be a consecutive occurrence of str(b), and let i' be the last occurrence of P before j. If $i' \neq i$ then i' is wihtin the cluster and thus (i', j) is part of the consecutive occurrences we already computed. We merge the two lists, deleting all consecutive occurrences of str(b) that are not consecutive occurrences of P and all duplicates. We then find the top-k of remaining consecutive occurrences, by using the linear time selection algorithm as in the proof of Lemma 4.

Time analysis Finding the locus in the suffix tree takes $O(m)$ time. Finding b and takes $O(\kappa \log n)$ time, and finding the occurrences of P within the cluster using sorted range reporting also takes $O(\kappa \log n)$ time. We make $O(\kappa \log n)$ calls to the orthogonal range reporting data structure each using $O(\log^{\epsilon} n)$ time. In total the time for this is $O(\kappa \log^{1+\epsilon} n)$. Merging the two lists and using the selection algorithm takes time linear in the total length of the two lists which is $O(\kappa \log n)$. Since $\kappa < 2k$ the total time complexity $O(m+k \log^{1+\epsilon} n)$. This concludes the proof of the main result.

2.8 Extensions

Our results can be extended to a couple of related problems. In Section 2.8.1, we show how we can modify our data structure to solve the "opposite" problem of reporting the k consecutive occurrences of *largest* distance. The extension is quite natural, though it does require some careful analysis. In Section 2.8.2, we then relate the solutions from Section 2.8.1 and the solutions to Sitcco to the problem of finding consecutive occurrences with distances in a specified interval, considered by Navarro and Thankachan [NT16]. We show improved complexities for the special case where one of the interval bounds is known at indexing time. Finally, we show how to use those results to efficiently find all pairs of non-overlapping consecutive occurrences.

2.8.1 Top-k Far Consecutive Occurrences

Given a pattern P and an integer parameter $k > 0$, define the *top-k far consecutive occurrences* of P to be the k consecutive occurrences of P in S with the largest distances. Given a string S the *string indexing for top-k far consecutive occurrences problem* (Sitfco) is to preprocess S into a data structure that supports top-k far consecutive occurrences queries. The goal is to obtain a compact data structure while supporting fast queries in terms of the length of the pattern P and the number of reported occurrences k.

Line segments and an $O(n \log n)$ space solution We can solve the Sitfco problem using the same strategy as for the Sitcco problem, with small modifications. We need a similar data structure from Lemma 2 to report the line segments with largest x-coordinates. As previously, assume we are given a set L of n vertical line segments. We need a data structure for the following problem:

- largest-segments(y_0, k): return the first k segments intersecting the horizontal line with y-coordinate y_0 in right-to-left order.

As before, we can assume integer coordinates and let N be the maximum y-coordinate of any line segment in L.

Lemma 7. *We can build a data structure that can answer largest-segment queries in $O(n + N)$ space and $O(k)$ time.*

Proof. We build the same data structure as in Lemma 2, but keep the partially persistent linked list sorted in decreasing order. The rest follows as before. □

Now, using this data structure in the solution described in Section 2.3, we immediately get an analogous result for Sitfco:

Lemma 8. *Given a string S of length n, we can build a data structure that can answer* top-k far consecutive occurrences *queries using $O(n \log n)$ space and $O(m + k)$ query time.*

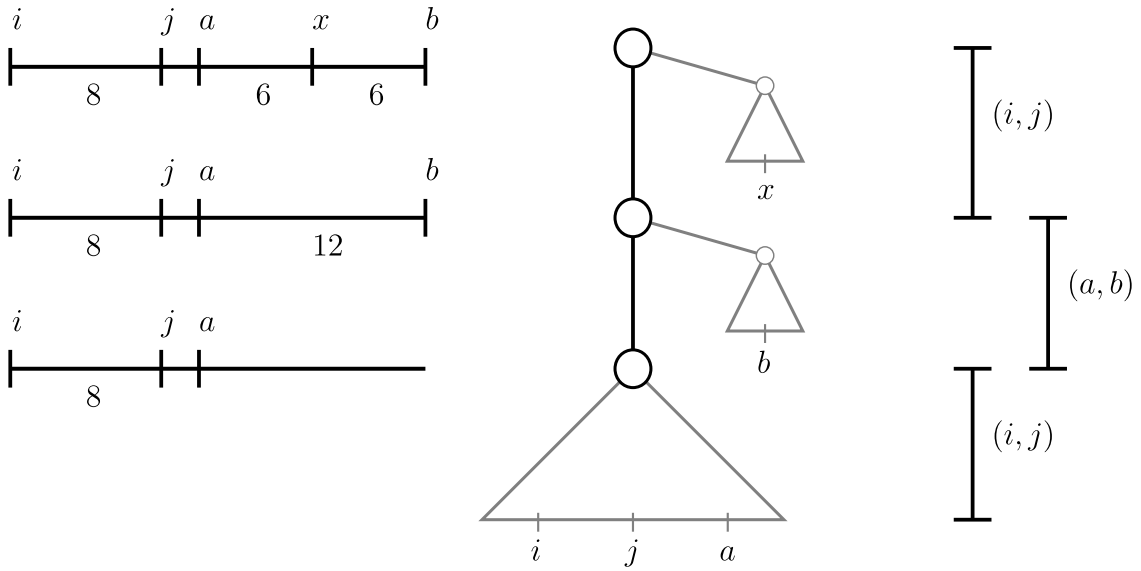

Figure 2.5: Illustration of a pair defining more than one line segment. To the left are the positions of the occurrences in S, in the middle is the spine of a cluster and to the right are the corresponding line segments. The pair (i, j) is amongst the to k farthest until the occurrence x disappears, after which it is pushed out by the pair (a, b). When b then disappears, (i, j) is again amongst the k farthest.

Modifications to the linear space data structure Now we extend the cluster solutions from Sections 2.5 and 2.6. We build the same recursive clusters as in Section 2.5. For each spine of a cluster of size τ_i, we keep the line segments corresponding to the τ_i consecutive occurrences of largest distance at every depth on the spine. That is, if a consecutive occurrence (i, j) is among the k farthest within $D(v_d)$ for some v_d on the spine, define line segments for all maximal consecutive intervals $[d_1, d_2]$ such that (i, j) is amongst the k farthest within $D(v_d)$ for any $d \in [d_1, d_2]$. Again, the x-coordinate of the line segment is the distance $j - i$, and the y-coordinate spans $[d_1, d_2]$. Note that in this case, a consecutive occurrence might define more than one line segment. See Figure 2.5 for an illustration of pair defining more than one line segment. We store these line segments in the data structure from Lemma 7.

Space analysis When moving down a spine from v_{d-1} to v_d, only three different types of changes can happen to the set of the k farthest consecutive occurrences. We again denote $D(v_d)$ to be the set of all consecutive occurrences of $\text{str}(v_d)$. The possibles types of changes are then as follows.

- A consecutive occurrence can be removed from the k farthest because a consecutive occurrence of larger distance is added to $D(v_d)$. The consecutive occurrence of larger distance can only appear if an occurrence in between branched off. This leaf accounts for this change. A leaf can account for at most one such change, which triggers a line segment ending and a new line segment appearing at depth d.

- A consecutive occurrence (i, j) can disappear because either i or j branched off. Then this leaf accounts for this change. A leaf can account for at most two such changes.

- A consecutive occurrence that was present in $D(v_{d-1})$ but not amongst the k farthest can be added to the k farthest in $D(v_d)$. This can only happen if a consecutive occurrence of greater distance disappeared because one or both of its occurrences branched off. Then this leaf accounts for this new line segment also, additional to the charge of the disappearing consecutive occurrence(s). A leaf can account for at most two such changes.

In total, any leaf can account for at most a constant number of changes. Thus, we get the same space complexities as in Section 2.5.

Algorithm To answer a query we proceed as in Section 2.5. We first find locus(P). If it is on a spine of a cluster of size $O(\tau_i)$ and $k < \tau_i$, we query the line segment data structure to report the top-k far consecutive occurrences. Otherwise, we find all occurrences of P in text order, construct the consecutive occurrences and use linear time selection to output the k consecutive occurrences of largest distance. This is correct by the same arguments as Section 2.5, and by similar arguments, achieves the same time complexities. The rank space reduction from Section 2.6 can be applied analogously. This gives us the following result:

Theorem 2. *Given a string S of length n and ϵ, $0 < \epsilon \leq 1$, we can build a data structure that can answer* top-k far consecutive occurrences *queries using either*

(i) $O(n \log n)$ *space and* $O(m + k)$ *query time or*

(ii) $O(\frac{n}{\epsilon})$ *space and* $O(m + k^{1+\epsilon})$ *query time.*

Here, m is the length of the query pattern.

Remark We note that the construction from Section 2.7 does not generalize to the top-k far consecutive occurrences problem, since the corresponding version of Claim 1 does not hold. A top-k far consecutive occurrence of P, where both occurrences are below b is not necessarily among the top-k far consecutive occurrences of str(b). A consecutive occurrence of str(b) can be split by an occurrence of P not below b. This gives two smaller occurrences, and might cause the $(k+1)$th furthest occurrence below b to be among the top-k far occurrences of P. Each occurrence of P from within the cluster can split a consecutive occurrence below b, and thus we would need to store $\Theta(\tau)$ occurrences below b, which no longer gives a linear space solution.

2.8.2 Consecutive Occurrences with Gaps

Given a string S the *string indexing for consecutive occurrences with gaps problem* (SICOG) is to preprocess S into a compact data structure, such that for any pattern P and a range $[\alpha, \beta]$ we can efficiently find all consecutive occurrences of P where the distance lies within $[\alpha, \beta]$. The SICOG problem was considered by Navarro and Thankachan [NT16] and they give an $O(n \log n)$ space and $O(m + \text{occ})$ time solution, where occ is the number of consecutive pairs with distance in $[\alpha, \beta]$. Using the data structure from Section 2.3, we get an $O(n \log n)$ space and $O(m + \log n + \text{occ})$ time solution for the SICOG problem, which can be optimized using the same strategy as in [NT16] to achieve the same complexities. However, for a special case of the problem where either α or β is known at indexing time we can get a similar trade-off as for the SITCCO problem. We first describe our solution for the fixed-α variant using the techniques from Sections 2.5 and 2.6, and then the fixed-β variant follows by applying the same ideas combined with the data structure from Section 2.8.1.

Data structure We build the same data structure as in Section 2.5, with a slight modification. In the line segment data structure stored at every spine, instead of storing the τ_i closest pairs, we store the τ_i closest pairs that have distance $\geq \alpha$. This clearly occupies no more space than the solution from Section 2.5 and we can still apply the space optimizations of Section 2.6.

Algorithm Given P and β, we can now answer a query as follows. We begin by finding locus(P). If it is in a subtree of constant size, we construct all the consecutive occurrences of P and report those that have distance within $[\alpha, \beta]$. If it is on a spine of a cluster of size τ_i, we query the line segment data structure. For every consecutive occurrence we find, we check if the distance is $\leq \beta$. If we encounter a pair with distance $> \beta$, we stop reporting. If all τ_i consecutive occurrences at locus(P) have distance $\leq \beta$, we then find all the

the consecutive occurrences of P, just as in Section 2.5, and scan them once to report all the consecutive occurrences with distance in $[\alpha, \beta]$.

For the analysis, define a *relevant pair* to be a consecutive occurrence with a distance in $[\alpha, \beta]$. If there are less than τ_i relevant pairs for any locus, then they will all be stored and reported by the line segment data structure. As they are stored in order of increasing distance, once we reach a pair with distance $> \beta$, no further relevant pairs exist. If there are more than τ_i relevant pairs, then we consider all occurrences of the pattern and report from those. Thus we always answer the query correctly.

If there is a consecutive occurrence among the τ_i line segments with distance $> \beta$, we spend time $O(m+\text{occ})$ finding the locus and querying the line segment data structure. Otherwise we have that $\text{occ} \geq \tau_i$, and thus $\text{occ}^{1+\epsilon} \geq \tau_{i-1}$. As before, P has at most τ_{i-1} occurrences. Therefore, by the same arguments as in the previous section, we get the following result:

Theorem 3. *Given a string S of length n and $\alpha > 0$, we can build for any ϵ satisfying $0 < \epsilon \leq 1$ an $O(\frac{n}{\epsilon})$ space data structure that can answer the following query in $O(m + \text{occ}^{1+\epsilon})$ time: For a query pattern P and $\beta \geq \alpha$, report all consecutive occurrences of P in S where the distance lies in $[\alpha, \beta]$. Here, m is the length of the pattern and* occ *is the number of reported occurrences.*

By combining the same arguments with the solution for top-k far consecutive occurrences, we get the following result for β fixed at indexing time:

Theorem 4. *Given a string S of length n and $\beta > 0$, we can build for any ϵ satisfying $0 < \epsilon \leq 1$ an $O(\frac{n}{\epsilon})$ space data structure that can answer the following query in $O(m + \text{occ}^{1+\epsilon})$ time: For a query pattern P and α where $0 < \alpha \leq \beta$, report all consecutive occurrences of P in S where the distance lies in $[\alpha, \beta]$. Here, m is the length of the pattern and* occ *is the number of reported occurrences.*

We note that the construction from Section 2.7 does not generalize to the problem where α is fixed, as the corresponding version of Claim 1 do not hold in this case. A consecutive occurrence below b of distance at least α can be split by an occurrence of P from within the cluster introducing two new consecutive occurrences that might both have distance less than α.

We can, however, get a solution to the problem when $\alpha = 1$. Using the data structure from Section 2.7, we can get all consecutive occurrences that are at most β apart as follows. We query the data structure for the top-k close consecutive occurrences with $k = 1, 2, 4, \ldots$, each time checking if all the top-k close consecutive occurrences have distance at most β. As soon as we find a consecutive occurrence that has a distance more than β among our top-k close consecutive occurrences, we stop and report all the occurrences found in this last call that have distance at most β. This way we ensure that $k \leq 2 \cdot \text{occ}$. We only find the locus once. The total query time is $O(m + \sum_{i=0}^{\lceil \log \text{occ} \rceil} 2^i \log^{1+\epsilon} n) = O(m + \text{occ} \log^{1+\epsilon} n)$.

Lemma 9. *Given a string S of length n we can build an $O(n)$ space data structure that can answer the following query in $O(m + \text{occ} \log^{1+\epsilon} n)$ time: For a query pattern P and $\beta > 0$, report all consecutive occurrences with distance at most β.*

Non-overlapping consecutive occurrences A natural and well-studied variant of string indexing is the problem of finding sets of *non-overlapping* occurrences of a pattern P. Here, a set of non-overlapping occurrences is a set of occurrences $\{i_1, \ldots, i_k\}$ of P such that the distance between any two of them is at least $|P|$. Several papers study the problem of finding the set of non-overlapping occurrences of maximum size [KKL07, CP09, GST20, HAKT18]. Note that Theorem 4 applied to $\alpha = |P|$ solves a different variant of finding sets of non-overlapping occurrences: Namely, finding all pairs of non-overlapping *consecutive* occurrences. We call this problem the *string indexing for non-overlapping consecutive occurrences problem* (Sinoco). The Sinoco problem is inherently different from finding the maximum set of non-overlapping occurrences: For example, the maximum set of non-overlapping occurrences of the pattern $P = \texttt{NANA}$ in the string $S = \texttt{NANANANA}$ has size 2. However, there are no non-overlapping *consecutive* occurrences. To the best of our knowledge, the Sinoco problem has not been studied before. An immediate corollary of the results in Navarro and Thankachan [NT16] and Theorem 4 gives the following trade-offs for solving Sinoco:

Corollary 1. *Given a string S of length n and ϵ, $0 < \epsilon \leq 1$, we can build a data structure that can find all non-overlapping consecutive occurrences of a query pattern P using either*

(i) $O(n \log n)$ *space and* $O(m + \mathrm{occ})$ *query time or*

(ii) $O(\frac{n}{\epsilon})$ *space and* $O(m + \mathrm{occ}^{1+\epsilon})$ *query time.*

Here, m is the length of the query pattern and occ *is the number of reported occurrences.*

Proof. Apply the results in [NT16] and Theorem 4 with $\beta = n$ and $\alpha = |P|$. □

2.9 Conclusion and Open Problems

We have introduced the natural problem of string indexing for top-k close consecutive occurrences, and have given both a near-linear space solution achieving optimal query time and a linear space solution achieving a query time that is close to optimal. Using these techniques, we have given new solutions for the problem of string indexing for consecutive occurrences with gaps (Sicog). Furthermore, we have introduced the problem of finding all non-overlapping consecutive occurrences of a pattern (Sinoco) and showed that it can be reduced to a special case of Sicog.

These results open interesting new directions for further research. The most obvious open problem is to see whether it is possible to further improve the results for the main problem considered in this paper, especially, achieve linear space and optimal query time simultaneously. Secondly, it is still open whether it is possible to get an $O(m + \mathrm{occ})$ time and linear space solution for the special case of the Sicog problem where one of the interval endpoints is fixed, or even $o(n \log n)$ space for the general problem. For the Sinoco problem, one might find better solutions that do not reduce it to Sicog but use additional insights about the specific structure of the problem.

Acknowledgments

We thank anonymous reviewers of an earlier draft of this paper for their insightful comments and suggestions for improvement. Philip Bille, Inge Li Gørtz, Max Pedersen and Eva Rotenberg were partially supported by the Danish Research Council grant *Adaptive Compressed Computation* (DFF-8021-002498).

Chapter 3

Gapped Indexing for Consecutive Occurrences

Abstract

The classic string indexing problem is to preprocess a string S into a compact data structure that supports efficient pattern matching queries. Typical queries include *existential queries* (decide if the pattern occurs in S), *reporting queries* (return all positions where the pattern occurs), and *counting queries* (return the number of occurrences of the pattern). In this paper we consider a variant of string indexing, where the goal is to compactly represent the string such that given two patterns P_1 and P_2 and a gap range $[\alpha, \beta]$ we can quickly find the consecutive occurrences of P_1 and P_2 with distance in $[\alpha, \beta]$, i.e., pairs of subsequent occurrences with distance within the range. We present data structures that use $\widetilde{O}(n)$ space and query time $\widetilde{O}(|P_1|+|P_2|+n^{2/3})$ for existence and counting and $\widetilde{O}(|P_1|+|P_2|+n^{2/3}\mathrm{occ}^{1/3})$ for reporting. We complement this with a conditional lower bound based on the set intersection problem showing that any solution using $\widetilde{O}(n)$ space must use $\widetilde{\Omega}(|P_1|+|P_2|+\sqrt{n})$ query time. To obtain our results we develop new techniques and ideas of independent interest including a new suffix tree decomposition and hardness of a variant of the set intersection problem.

3.1 Introduction

The classic string indexing problem is to preprocess a string S into a compact data structure that supports efficient pattern matching queries. Typical queries include *existential queries* (decide if the pattern occurs in S), *reporting queries* (return all positions where the pattern occurs), and *counting queries* (return the number of occurrences of the pattern). An important variant of this problem is the *gapped string indexing problem* [BG14, IR09, BGP16, CPZ20, BGVV14, Lew11, KKL07, APU11]. Here, the goal is to compactly represent the string such that given two patterns P_1 and P_2 and a *gap range* $[\alpha, \beta]$ we can quickly find occurrences of P_1 and P_2 with distance in $[\alpha, \beta]$. Searching and indexing with gaps is frequently used in computational biology applications [BB94, HBFB99, FG08, HSSSS11, Mye92, MM93b, NR03, BGP16, CPZ20, BGVW12].

Another variant is *string indexing for consecutive occurrences* [NT16, BGP+20a, AS09, APS04]. Navarro and Thankachan [NT16] study the problem of compactly representing the string such that given a pattern P and a gap range $[\alpha, \beta]$ we can quickly find *consecutive occurrences* of P with distance in $[\alpha, \beta]$, i.e., pairs of subsequent occurrences with distance within the range.

In this paper, we consider the natural combination of these variants that we call *gapped indexing for consecutive occurrences*. Here, the goal is to compactly represent the string such that given two patterns P_1 and P_2 and a gap range $[\alpha, \beta]$ we can quickly find the consecutive occurrences of P_1 and P_2 with distance in $[\alpha, \beta]$.

We can apply standard techniques to obtain several simple solutions to the problem. To state the bounds, let n be the size of S. If we store the suffix tree for S, we can answer queries by searching for both query

strings, merging the results, and removing all non-consecutive occurrences. This leads to a solution using $O(n)$ space and $\widetilde{O}(|P_1| + |P_2| + \text{occ}_{P_1} + \text{occ}_{P_2})$ query time, where occ_{P_1} and occ_{P_2} denote the number of occurrences of P_1 and P_2, respectively[1]. However, $\text{occ}_{P_1} + \text{occ}_{P_2}$ may be as large as $\Omega(n)$ and much larger than the size of the output.

Alternatively, we can obtain constant query time for counting queries by precomputing the answer for each pair of suffix tree nodes and each possible distance in $O(n^3)$ space.

In this paper, we introduce new solutions that significantly improve the above time-space trade-offs. Specifically, we present data structures that use $\widetilde{O}(n)$ space and query time $\widetilde{O}(|P_1|+|P_2|+n^{2/3})$ for existence and counting and $\widetilde{O}(|P_1| + |P_2| + n^{2/3}\text{occ}^{1/3})$ for reporting. We complement this with a conditional lower bound based on the set intersection problem showing that any solution using $\widetilde{O}(n)$ space must use $\widetilde{\Omega}(|P_1| + |P_2| + \sqrt{n})$ query time. To obtain our results we develop new techniques and ideas of independent interest including a new suffix tree decomposition and hardness of a variant of the set intersection problem.

3.1.1 Setup and Results

Throughout the paper, let S be a string of length n. Given two patterns P_1 and P_2 a *consecutive occurrence* in S is a pair of occurrences (i, j), $0 \leq i < j < |S|$ where i is an occurrence of P_1 and j an occurrence of P_2, such that no other occurrences of either P_1 or P_2 occurs in between. The *distance* of a consecutive occurrence (i, j) is $j - i$. Our goal is to preprocess S into a compact data structure that given pattern strings P_1 and P_2 and a gap range $[\alpha, \beta]$ supports the following queries:

- $\mathsf{Exists}(P_1, P_2, \alpha, \beta)$: determine if there is a consecutive occurrence of P_1 and P_2 with distance within the range $[\alpha, \beta]$.
- $\mathsf{Count}(P_1, P_2, \alpha, \beta)$: return the number of consecutive occurrences of P_1 and P_2 with distance within the range $[\alpha, \beta]$.
- $\mathsf{Report}(P_1, P_2, \alpha, \beta)$: report all consecutive occurrences of P_1 and P_2 with distance within the range $[\alpha, \beta]$.

We present new data structures with the following bounds:

Theorem 5. *Given a string of length n, we can*

(i) *construct an $O(n)$ space data structure that supports $\mathsf{Exists}(P_1, P_2, \alpha, \beta)$ and $\mathsf{Count}(P_1, P_2, \alpha, \beta)$ queries in $O(|P_1| + |P_2| + n^{2/3} \log^{\epsilon} n)$ time for constant $\epsilon > 0$, or*

(ii) *construct an $O(n \log n)$ space data structure that supports $\mathsf{Report}(P_1, P_2, \alpha, \beta)$ queries in $O(|P_1|+|P_2|+ n^{2/3}\text{occ}^{1/3} \log n \log\log n)$ time, where* occ *is the size of the output.*

Hence, ignoring polylogarithmic factors, Theorem 5 achieves $\widetilde{O}(n)$ space and query time $\widetilde{O}(|P_1| + |P_2| + n^{2/3})$ for existence and counting and $\widetilde{O}(|P_1| + |P_2| + n^{2/3}\text{occ}^{1/3})$ for reporting. Compared to the above mentioned simple suffix tree approach that finds all occurrences of the query strings and merges them, we match the $\widetilde{O}(n)$ space bound, while reducing the dependency on n in the query time from worst-case $\Omega(|P_1| + |P_2| + n)$ to $\widetilde{O}(|P_1| + |P_2| + n^{2/3})$ for Exists and Count queries and $\widetilde{O}(|P_1| + |P_2| + n^{2/3}\text{occ}^{1/3})$ for Report queries.

We complement Theorem 5 with a conditional lower bound based on the set intersection problem. Specifically, we use the Strong SetDisjointness Conjecture from [GKLP17] to obtain the following result:

Theorem 6. *Assuming the Strong SetDisjointness Conjecture, any data structure on a string S of length n that supports Exists queries in $O(n^{\delta} + |P_1| + |P_2|)$ time, for $\delta \in [0, 1/2]$, requires $\widetilde{\Omega}\left(n^{2-2\delta-o(1)}\right)$ space. This bound also holds if we limit the queries to only support ranges of the form $[0, \beta]$, and even if the bound β is known at preprocessing time.*

[1] $\widetilde{O}$ and $\widetilde{\Omega}$ ignores polylogarithmic factors

With $\delta = 1/2$, Theorem 6 implies that any near linear space solution must have query time $\widetilde{\Omega}(|P_1| + |P_2| + \sqrt{n})$. Thus, Theorem 5 is optimal within a factor roughly $n^{1/6}$. On the other hand, with $\delta = 0$, Theorem 6 implies that any solution with optimal $\widetilde{O}(|P_1| + |P_2|)$ query time must use $\widetilde{\Omega}(n^{2-o(1)})$ space.

Finally, note that Theorem 6 holds even when the gap range is of the form $[0, \beta]$. As a simple extension of our techniques, we show how to improve our solution from Theorem 5 to match Theorem 6 in this special case.

3.1.2 Techniques

To obtain our results we develop new techniques and show new interesting properties of consecutive occurrences. We first consider Exists and Count queries. The key idea is to split gap ranges into large and small distances. For large distances there can only be a limited number of consecutive occurrences and we show how these can be efficiently handled using a segmentation of the string. For small distances, we cluster the suffix tree and store precomputed answers for selected pairs of nodes. Since the number of distinct distances is small we obtain an efficient bound on the space.

We extend our solution for Exists and Count queries to handle Report queries. To do so we develop a new decomposition of suffix trees, called the *induced suffix tree decomposition* that recursively divides the suffix tree in half by index in the string. Hence, the decomposition is a balanced binary tree, where every node stores the suffix tree of a substring of S. We show how to traverse this structure to efficiently recover the consecutive occurrences.

For our conditional lower bound we show a reduction based on the set intersection problem. Along the way we show that set intersection remains hard even if all elements in the instance have the same frequency.

3.1.3 Related Work

As mentioned, string indexing for gaps and consecutive occurrences are the most closely related lines of work to this paper. Another related area is *document indexing*, where the goal is to preprocess a collection of strings, called *documents*, to report those documents that contain patterns subject to various constraints. For a comprehensive overview of this area see the survey by Navarro [Nav14].

A well studied line of work within document indexing is *document indexing for top-k* queries [MNST20, SSTV13, HSTV14, BGST18, HPSW10, NN17, Tsu13, HPS$^+$13, HTSV13, NT14, MNN$^+$17]. The goal is to efficiently report the top-k documents of smallest weight, where the weight is a function of the query. Specifically, the weight can be the distance of a pair of occurrences of the same or two different query patterns [HSTV14, NN17, SSTV13, MNN$^+$17]. The techniques for top-k indexing (see e.g. Hon et al. [HSTV14]) can be adapted to efficiently solve gapped indexing for consecutive occurrences in the special case when the gap range is of the form $[0, \beta]$. However, since these techniques heavily exploit that the goal is to find the top-k *closest occurrences*, they do not generalize to general gap ranges.

There are several results on conditional lower bounds for pattern matching and string indexing [LMNT15, GKLP17, ACLL14, KPP16, AKL$^+$16, KK16]. Notably, Kopelowitz and Krauthgamer [KK16] consider the *snippets* problem, where the goal is to preprocess the text to enable fast reporting of the closest pair of occurrences of query patterns P_1 and P_2. They prove a lower bound for that problem based on SetDisjointness, which is closely related to our lower bound in Theorem 6. Our result uses a similar reduction but introduces an intermediate step of potential independent interest, where we prove hardness for instances of SetDisjointness where every element has the same frequency. Ultimately, this leads to a clean proof of the final lower bound.

Other results also establish a link between indexing for two patterns and set intersection. Ferragina et al. [FKMS03] and Cohen and Porat [CP10] reduce the *two dimensional substring indexing problem* to set intersection (though the goal was to prove an upper, not a lower bound). In the two dimensional substring indexing problem the goal is to preprocess pairs of strings such that given two patterns we can output the pairs that contain a pattern each. Larsen et al. [LMNT15] prove a conditional lower bound for the document version of indexing for two patterns, i.e., finding all documents containing both of the two query patterns. Goldstein et al. [GKLP17] show that similar lower bounds can be achieved via conjectured hardness of set

intersection. Our reduction is still quite different from these, since we need a translation from intersection to distance.

3.1.4 Outline

The paper is organized as follows. In Section 3.2 we define notation and recall some useful results. In Section 3.3 we show how to answer Exists and Count queries, proving Theorem 5(i). In Section 3.4 we show how to answer Report queries, proving Theorem 5(ii). In Section 3.5 we prove the lower bound, proving Theorem 6. Finally, in Section 3.6 we apply our techniques to solve the variant where $\alpha = 0$.

3.2 Preliminaries

Strings. A *string* S of length n is a sequence $S[0]S[1]\dots S[n-1]$ of characters from an alphabet Σ. A contiguous subsequence $S[i,j] = S[i]S[i+1]\dots S[j]$ is a *substring* of S. The substrings of the form $S[i, n-1]$ are the *suffixes* of S. The *suffix tree* [Wei73] is a compact trie of all suffixes of $S\$$, where \$ is a symbol not in the alphabet, and is lexicographically smaller than any letter in the alphabet. Each leaf is labelled with the index i of the suffix $S[i, n-1]$ it corresponds to. Using perfect hashing [FKS84], the suffix tree can be stored in $O(n)$ space and solve the string indexing problem (i.e., find and report all occurrences of a pattern P) in $O(m + \text{occ})$ time, where m is the length of P and occ is the number of times P occurs in S.

For any node v in the suffix tree, we define $\text{str}(v)$ to be the string found by concatenating all labels on the path from the root to v. The *locus* of a string P, denoted $\text{locus}(P)$, is the minimum depth node v such that P is a prefix of $\text{str}(v)$. The *suffix array* stores the suffix indices of $S\$$ in lexicographic order. We identify each leaf in the suffix tree with the suffix index it represents. The suffix tree has the property that the leaves below any node represent suffixes that appear in consecutive order in the suffix array. For any node v in the suffix tree, $\text{range}(v)$ denotes the range that v spans in the suffix array. The *inverse suffix array* is the inverse permutation of the suffix array, that is, an array where the ith element is the index of suffix i in the suffix array.

Orthogonal range successor. The *orthogonal range successor problem* is to preprocess an array $A[0,\dots,n-1]$ into a data structure that efficiently supports the following queries:

- RangeSuccessor(a, b, x): return the successor of x in $A[a,\dots,b]$, that is, the minimum $y > x$ such that there is an $i \in [a, b]$ with $A[i] = y$.
- RangePredecessor(a, b, x): return the predecessor of x in $A[a,\dots,b]$, that is, the maximum $y < x$ such that there is an $i \in [a, b]$ with $A[i] = y$.

3.3 Existence and Counting

In this section we give a data structure that can answer Exists and Count queries. The main idea is to split the query interval into "large" and "small" distances. For large distances we exploit that there can only be a small number of consecutive occurrences and we check them with a simple segmentation of S. For small distances we cluster the suffix tree and precompute answers for selected pairs of nodes.

We first show how to use orthogonal range successor queries to find consecutive occurrences. Then we define the clustering scheme used for the suffix tree and give the complete data structure.

3.3.1 Using Orthogonal Range Successor to Find Consecutive Occurrences

Assume we have found the loci of P_1 and P_2 in the suffix tree. Then we can answer the following queries in a constant number of orthogonal range successor queries on the suffix array:

- $\mathsf{FindConsecutive}_{P_2}(i)$: given an occurrence i of P_1, return the consecutive occurrence (i, j) of P_1 and P_2, if it exists, and NO otherwise.
- $\mathsf{FindConsecutive}_{P_1}(j)$: given an occurrence j of P_2, return the consecutive occurrence (i, j) of P_1 and P_2, if it exists, and NO otherwise.

Given a query $\mathsf{FindConsecutive}_{P_2}(i)$, we answer as follows. Compute $j = \mathsf{RangeSuccessor}(\text{range}(\text{locus}(P_2)), i)$ to get the closest occurrence of P_2 after i. Compute $i' = \mathsf{RangePredecessor}(\text{range}(\text{locus}(P_1)), j)$ to get the closest occurrence of P_1 before j. If $i = i'$ then no other occurrence of P_1 exists between i and j and they are consecutive. In that case we return (i, j). Otherwise, we return NO.

Similarly, we can answer $\mathsf{FindConsecutive}_{P_1}(j)$ by first doing a $\mathsf{RangePredecessor}$ and then a $\mathsf{RangeSuccessor}$ query. Thus, given the loci of both patterns and a specific occurrence of either P_1 or P_2, we can in a constant number of $\mathsf{RangeSuccessor}$ and $\mathsf{RangePredecessor}$ queries find the corresponding consecutive occurrence, if it exists.

3.3.2 Data Structure

To build the data structure we will use a cluster decomposition of the suffix tree.

Cluster Decomposition A cluster decomposition of a tree T is defined as follows: For a connected subgraph $C \subseteq T$, a *boundary node* v is a node $v \in C$ such that either v is the root of T, or v has an edge leaving C – that is, there exists an edge (v, u) in the tree T such that $u \in T \setminus C$. A *cluster* is a connected subgraph C of T with at most two boundary nodes. A cluster with one boundary node is called a *leaf cluster*. A cluster with two boundary nodes is called a *path cluster*. For a path cluster C, the two boundary nodes are connected by a unique path. We call this path the *spine* of C. A *cluster partition* is a partition of T into clusters, i.e. a set CP of clusters such that $\bigcup_{C \in CP} V(C) = V(T)$ and $\bigcup_{C \in CP} E(C) = E(T)$ and no two clusters in CP share any edges. Here, $E(G)$ and $V(G)$ denote the edge and vertex set of a (sub)graph G, respectively. We need the next lemma which follows from well-known tree decompositions [AHdLT97, AHT00, AR02, Fre97] (see Bille and Gørtz [BG11] for a direct proof).

Lemma 10. *Given a tree T with n nodes and a parameter τ, there exists a cluster partition CP such that $|CP| = O(n/\tau)$ and every $C \in CP$ has at most τ nodes. Furthermore, such a partition can be computed in $O(n)$ time.*

Data Structure We build a clustering of the suffix tree of S as in Lemma 10, with cluster size at most τ, where τ is some parameter satisfying $0 < \tau \leq n$. Then the counting data structure consists of:

- The suffix tree of S, with some additional information for each node. For each node v we store:
  - The range v spans in the suffix array, i.e., range(v).
  - A bit that indicates if v is on a spine.
  - If v is on a spine, a pointer to the lower boundary node of the spine.
  - If v is a leaf, the local rank of v. That is, the rank of v in the text order of the leaves in the cluster that contains v. Note that this is at most τ.
- The inverse suffix array of S.
- A range successor data structure on the suffix array of S.
- An array $M(u, v)$ of length $\lfloor \frac{n}{\tau} \rfloor + 1$ for every pair of boundary nodes (u, v). For $1 \leq x \leq \lfloor \frac{n}{\tau} \rfloor$, $M(u, v)[x]$ is the number of consecutive occurrences (i, j) of str(u) and str(v) with distance at most x. We set $M(u, v)[0] = 0$.

Denote $M(u, v)[\alpha, \beta] = M(u, v)[\beta] - M(u, v)[\alpha - 1]$, that is, $M(u, v)[\alpha, \beta]$ is the number of consecutive occurrences of str(u) and str(v) with a distance in $[\alpha, \beta]$.

Space Analysis. We store a constant amount of words per node in the suffix tree. The suffix tree and inverse suffix array occupy $O(n)$ space. For the orthogonal range successor data structure we use the data structure of Nekrich and Navarro [NN12] which uses $O(n)$ space and $O(\log^\epsilon n)$ time, for constant $\epsilon > 0$. There are $O\left(n^2/\tau^2\right)$ pairs of boundary nodes and for each pair we store an array of length $O\left(n/\tau\right)$. Therefore the total space consumption is $O\left(n + n^3/\tau^3\right)$.

3.3.3 Query Algorithm

We now show how to count the consecutive occurrences (i, j) with a distance in the interval, i.e. $\alpha \leq j - i \leq \beta$. We call each such pair a *valid occurrence*.

To answer a query we split the query interval $[\alpha, \beta]$ into two: $[\alpha, \lfloor \frac{n}{\tau} \rfloor]$ and $[\lfloor \frac{n}{\tau} \rfloor + 1, \beta]$, and handle these separately.

3.3.3.1 Handling Distances $> \frac{n}{\tau}$.

We start by finding the loci of P_1 and P_2 in the suffix tree. As shown in Section 3.3.1, this allows us to find the consecutive occurrence containing a given occurrence of either P_1 or P_2. We implicitly partition the string S into segments of (at most) $\lfloor n/\tau \rfloor$ characters by calculating τ segment boundaries. Segment i, for $0 \leq i < \tau$, contains characters $S[i \cdot \lfloor \frac{n}{\tau} \rfloor, (i+1) \cdot \lfloor \frac{n}{\tau} \rfloor - 1]$ and segment τ (if it exists) contains the characters $S[\tau \cdot \lfloor \frac{n}{\tau} \rfloor, n-1]$. We find the last occurrence of P_1 in each segment by performing a series of $\mathsf{RangePredecessor}$ queries, starting from the beginning of the last segment. Each time an occurrence i is found we perform the next query from the segment boundary to the left of i, continuing until the start of the string is reached. For each occurrence i of P_1 found in this way, we use $\mathsf{FindConsecutive}_{P_2}(i)$ to find the consecutive occurrence (i, j) if it exists. We check each of them, discard any with distance $\leq \frac{n}{\tau}$ and count how many are valid.

3.3.3.2 Handling Distances $\leq \frac{n}{\tau}$.

In this part, we only count valid occurrences with distance $\leq \frac{n}{\tau}$. Consider the loci of P_1 and P_2 in the suffix tree. Let C_i denote the cluster that contains $\text{locus}(P_i)$ for $i = 1, 2$. There are two main cases.

At least one locus is not on a spine If either locus is in a small subtree hanging off a spine in a cluster or in a leaf cluster, we directly find all consecutive occurrences as follows: If $\text{locus}(P_1)$ is in a small subtree then we use $\mathsf{FindConsecutive}_{P_2}(i)$ on each leaf i below $\text{locus}(P_1)$ to find all consecutive occurrences, count the valid occurrences and terminate. If only $\text{locus}(P_2)$ is in a small subtree then we use $\mathsf{FindConsecutive}_{P_1}(j)$ for each leaf j below $\text{locus}(P_2)$, count the valid occurrences and terminate.

Both loci are on the spine If neither locus is in a small subtree then both are on spines. Let (b_1, b_2) denote the lower boundary nodes of the clusters C_1 and C_2, respectively. There are two types of consecutive occurrences (i, j):

(i) Occurrences where either i or j are inside C_1 resp. C_2.

(ii) Occurrences below the boundary nodes, that is, i is below b_1 and j is below b_2.

See Figure 3.1(a). We describe how to count the different types of occurrences next.

Type (i) occurrences To find the valid occurrences (i, j) where either $i \in C_1$ or $j \in C_2$ we do as follows. First we find all the consecutive occurrences (i, j) where i is a leaf in C_1 by computing $\mathsf{FindConsecutive}_{P_2}(i)$ for all leaves i below $\text{locus}(P_1)$ in C_1. We count all valid occurrences we find in this way. Then we find all remaining consecutive occurrences (i, j) where j is a leaf in C_2 by computing $\mathsf{FindConsecutive}_{P_1}(j)$ for all leaves j below $\text{locus}(P_2)$ in C_2. If $\mathsf{FindConsecutive}_{P_1}(j)$ returns a valid occurrence (i, j) we use the inverse suffix array to check if the leaf i is below b_1. This can be done by checking whether i's position in the suffix array is in $\text{range}(b_1)$. If i is below b_1 we count the occurrence, otherwise we discard it.

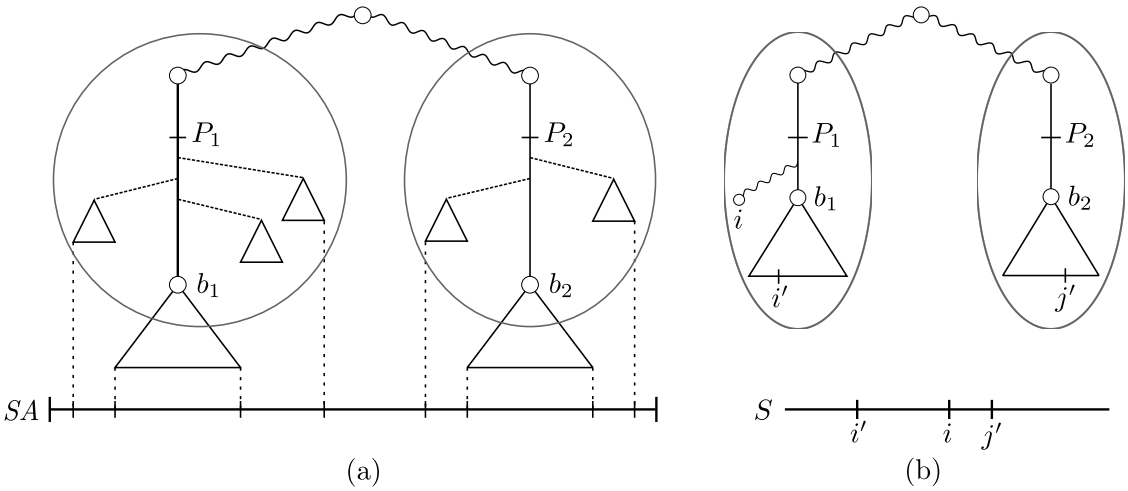

Figure 3.1: (a) Any consecutive occurrences (i, j) of P_1 and P_2 is either also a consecutive occurrence of $\text{str}(b_1)$ and $\text{str}(b_2)$, or i or j are within the respective cluster. The suffix array is shown in the bottom with the corresponding ranges marked. (b) Example of a false occurrence. Here (i', j') is a consecutive occurrence of $\text{str}(b_1)$ and $\text{str}(b_2)$, but not a consecutive occurrence of P_1 and P_2 due to i. The string S is shown in bottom with the positions of the occurrences marked.

Type (ii) occurrences Next, we count the consecutive occurrences (i, j), where both i and j are below b_1 and b_2, respectively. We will use the precomputed table, but we have to be a careful not to overcount. By its construction, $M(b_1, b_2)[\alpha, \min(\lfloor \frac{n}{\tau} \rfloor, \beta)]$ is the number of consecutive occurrences (i', j') of $\text{str}(b_1)$ and $\text{str}(b_2)$, where $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$. However, not all of these occurrence (i', j') are necessarily *consecutive* occurrences of P_1 and P_2, as there could be an occurrence of P_1 in C_1 or P_2 in C_2 which is between i' and j'. We call such a pair (i', j') a *false occurrence*. See Figure 3.1(b). We proceed as follows.

1. Set $c = M(b_1, b_2)[\alpha, \min(\lfloor \frac{n}{\tau} \rfloor, \beta)]$.
2. Construct the lists L_i containing the leaves in C_i that are below $\text{locus}(P_i)$ sorted by text order for $i = 1, 2$. We can obtain the lists as follows. Let $[a, b]$ be the range of $\text{locus}(P_i)$ and $[a', b'] = \text{range}(b_i)$. Sort the leaves in $[a, a' - 1] \cup [b' + 1, b]$ using their local rank.
3. Until both lists are empty iteratively pick and remove the smallest element e from the start of either list. There are two cases.
   - e is an element of L_1.
     - Compute $j' = \mathsf{RangeSuccessor}(\text{range}(b_2), e)$ to get the closest occurrence of $\text{str}(b_2)$ after e.
     - Compute $i' = \mathsf{RangePredecessor}(\text{range}(b_1), j')$ to get the closest occurrence of $\text{str}(b_1)$ before j'.
   - e is an element of L_2.
     - Compute $i' = \mathsf{RangePredecessor}(\text{range}(b_2), e)$ to get the previous occurrence i' of $\text{str}(b_1)$.
     - Compute $j' = \mathsf{RangeSuccessor}(\text{range}(b_1), j')$ to get the following occurrence j' of $\text{str}(b_2)$.

   If $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$ and $i' < e < j'$ decrement c by one. We skip any subsequent occurrences that are also inside (i', j'). As the lists are sorted by text order, all occurrences that are within the same consecutive occurrence (i', j') are handled in sequence.

Finally, we add the counts of the different type of occurrences.

Correctness. Consider a consecutive occurrence (i, j) where $j - i > \frac{n}{\tau}$. Such a pair must span a segment boundary, i.e., i and j cannot be in the same segment. As (i, j) is a *consecutive* occurrence, i is the last occurrence of P_1 in its segment and j is the first occurrence of P_2 in its segment. With the RangePredecessor queries we find all occurrences of P_1 that are the last in their segment. We thus check and count all valid occurrences of large distance in the initial pass of the segments.

If either locus is in a small subtree we use $\mathsf{FindConsecutive}_{P_2}(.)$ or $\mathsf{FindConsecutive}_{P_1}(.)$ on the leaves below that locus, which by the arguments in Section 3.3.1 will find all consecutive occurrences.

Otherwise, both loci are on a spine. To count type (i) occurrences we use $\mathsf{FindConsecutive}_{P_2}(i)$ for all leaves i below locus(P_1) in C_1 and $\mathsf{FindConsecutive}_{P_1}(j)$ for all leaves j below locus(P_2) in C_2. However, any valid occurrence (i, j) where both $i \in C_1$ and $j \in C_2$ is found by both operations. Therefore, whenever we find a valid occurrence (i, j) via $i = \mathsf{FindConsecutive}_{P_1}(j)$ for $j \in C_2$, we only count the occurrence if i is below b_1. Thus we count all type (i) occurrences exactly once.

To count type (ii) occurrences we start with $c = M(b_1, b_2)[\alpha, \min(\lfloor \frac{n}{\tau} \rfloor, \beta)]$, which is the number of consecutive occurrences (i', j') of str(b_1) and str(b_2), where $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$. Each (i', j') is either also a consecutive occurrence of P_1 and P_2, or there exists an occurrence of P_1 or P_2 between i' and j'. Let (i', j') be a false occurrence and let w.l.o.g. i be an occurrence of P_1 with $i' < i < j'$. Then i is a leaf in C_1, since (i', j') is a *consecutive* occurrence of str(b_1) and str(b_2). In step 3 we check for each leaf inside the clusters below the loci, if it is between a consecutive occurrence (i', j') of str(b_1) and str(b_2) and if $\alpha \leq j' - i' \leq \min(\lfloor \frac{n}{\tau} \rfloor, \beta)$. In that case (i', j') is a false occurrence and we adjust the count c. As (i', j') can have multiple occurrences of P_1 and P_2 inside it, we skip subsequent occurrences inside (i', j'). After adjusting for false occurrences, c is the number of type (ii) occurrences.

Time Analysis. We find the loci in $O(|P_1| + |P_2|)$ time. Then we perform a number of range successor and FindConsecutive queries. The time for a FindConsecutive query is bounded by the time to do a constant number of range successor queries. To count the large distances we check at most τ segment boundaries and thus perform $O(\tau)$ range successor and FindConsecutive queries.

For small distances, if either locus is not on a spine we check the leaves below that locus. There are at most τ such leaves due to the clustering. To count type (i) occurrences we check the leaves below the loci that are inside the clusters. There are at most 2τ such leaves in total. To count type (ii) occurrences we check two lists constructed from the leaves inside the clusters below the loci. There are again at most 2τ such leaves in total. For each of these $O(\tau)$ leaves we use a constant number of range successor and FindConsecutive queries. Thus the time for this part is bounded by the time to perform $O(\tau)$ range successor queries.

Using the data structure of Nekrich and Navarro [NN12], each range successor query takes $O(\log^\epsilon n)$ time so the total time for these queries is $O(\tau \log^\epsilon n)$. For type (ii) occurrences we sort two lists of size at most τ from a universe of size τ, which we can do in $O(\tau)$ time. Thus, the total query time is $O(|P_1|+|P_2|+\tau \log^\epsilon n)$.

Setting $\tau = \Theta(n^{2/3})$ we get a data structure that uses $O\left(n + n^3/\tau^3\right) = O(n)$ space and has query time $O(|P_1| + |P_2| + \tau \log^\epsilon n) = O(|P_1| + |P_2| + n^{2/3} \log^\epsilon n)$, for constant $\epsilon > 0$. We answer an Exists query with a Count query, terminating when the first valid occurrence is found. This concludes the proof of Theorem 5(i).

3.4 Reporting

In this section, we describe our data structure for reporting queries. Note that in Section 3.3, we explicitly find all valid occurrences *except* for type (ii) occurrences, where we use the precomputed values. In this section, we describe how we can use a recursive scheme to report these.

The main idea, inspired by fast set intersection by Cohen and Porat [CP10], is to build a recursive binary structure which allows us to recursively divide the problem into subproblems of half the size. Intuitively, the subdivision is a binary tree where every node contains the suffix tree of a substring of S. We use this structure to find type (ii) occurrences by recursing on smaller trees. We define the binary decomposition of the suffix tree next. The details of the full solution follow after that.

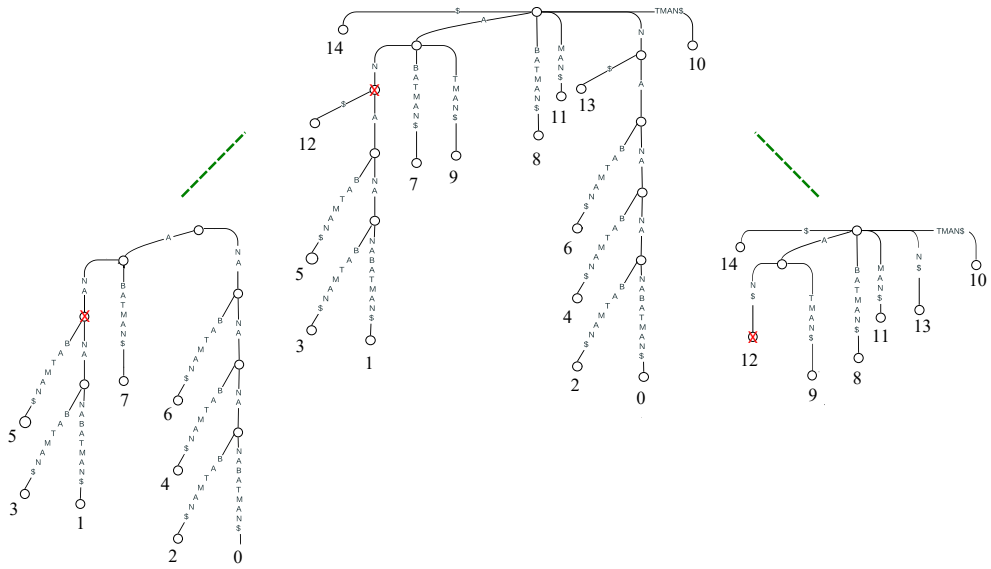

Figure 3.2: The suffix tree of `NANANANABATMAN$` together with its children trees $T[0, 7]$ and $T[8, 14]$. The red crosses show a node in the parent tree and and its successor nodes in the two children trees.

3.4.1 Induced Suffix Tree Decomposition

Let T be a suffix tree of a string S of length n. For an interval $[a, b]$ of *text positions*, we define $T[a, b]$ to be the subtree of T *induced* by the leaves in $[a, b]$: That is, we consider the subtree consisting of leaves in $[a, b]$ together with their ancestors. We then delete each node that has only one child in the subtree and contract its ingoing and outgoing edge. See Figure 3.2.

The *induced suffix tree decomposition of* T now consists of a higher level binary tree structure, the *decomposition tree*, where each node corresponds to an induced subtree of the suffix tree. The root corresponds to $T[0, n-1]$, and whenever we move down in the decomposition tree, the interval splits in half. We also associate a level with each of the induced subtrees, which is their depth in the decomposition tree. In more detail, the decomposition tree is a binary tree such that:

- The root of the decomposition tree corresponds to $T[0, n-1]$ and has level 0.
- For each $T[a, b]$ of level i in the decomposition, if $b - a > 1$, its two children in the decomposition tree are $T[a, c]$ and $T[c+1, b]$ where $c = \lfloor \frac{a+b}{2} \rfloor$; we will sometimes refer to these as "children trees" to differentiate from children in the suffix tree.

The decomposition tree is a balanced binary tree and the total size of the induced subtrees in the decomposition is $O(n \log n)$: There are at most 2^i decomposition tree nodes on level i, each of which corresponds to an induced subtree of size $O\left(\frac{n}{2^i}\right)$, and thus the total size of the trees on each of the $O(\log n)$ levels is $O(n)$.

For each node v in $T[a, b]$, we define the *successor node* of v in each of the children trees of $T[a, b]$ in the following way: If v exists in the child tree, the successor node is v. Else, it is the closest descendant which is present. Note that from the way the induced subtrees are constructed, v has at most one successor node in each child tree.

The induced suffix tree decomposition of S consists of:

- Each $T[a, b]$ stored as a compact trie.
- For each $T[a, b]$ we store a sparse suffix array $SA_{[a,b]}$, that is, the suffix array of $S[a, b]$ with the original indices within S.

- For each node v in $T[a,b]$ we store a pointer from v to its successor nodes in each child tree, if it exists, and the interval in $SA_{[a,b]}$ that corresponds to the leaves below v.

Since we store only constant information per node in any $T[a,b]$, the total space usage of this is $O(n \log n)$.

3.4.2 Data Structure

The reporting data structure consists of:

- The induced suffix tree decomposition for S,
- An orthogonal range successor data structure on the suffix array, and
- The data structure from Section 3.3 for each $T[a,b]$ in the induced suffix tree decomposition with parameters n_i and τ_i, where $n_i = \lfloor \frac{n}{2^i} \rfloor$ and $\tau_i = \Theta(n_i^{2/3})$, such that $n_i/\tau_i = \lfloor n_i^{1/3} \rfloor$. The only change is that we do not store an orthogonal range successor data structure for each of the induced subtrees.

Space Analysis. The data structure from Section 3.3 for each $T[a,b]$ of level i is linear in n_i. By the arguments of Section 3.4.1, the space for the suffix tree decomposition is $O(n \log n)$. Since we already have a space bottleneck of $O(n \log n)$, we use a different tradeoff for orthogonal range successor than in Section 3.3, namely, the $O(n \log \log n)$ space and $O(\log \log n)$ time structure by Zhou [Zho16]. The total space of the data structure is $O(n \log n)$.

3.4.3 Query Algorithm

The main idea behind the algorithm is the following: For large distances, as in Section 3.3, we implicitly segment S to find all consecutive occurrences of at least a certain distance. For small distances, we are going to use the cluster decomposition and counting arrays to decide whether valid occurrences exist. That is, if one of the loci is in a small subtree, we use $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$ to find all consecutive occurrences. Else, we perform a query as in Section 3.3 to decide whether any valid occurrences exist, and if yes, we recurse on smaller subtrees.

The idea here is, that in the induced suffix tree decomposition, the trees are divided in half by *text position* - therefore, a *consecutive* occurrence either will be fully contained in the left child tree, fully contained in the right child tree, or have the property that the occurrence of P_1 is the maximum occurrence in the left child tree and the occurrence of P_2 is the minimum occurrence in the right child tree. We will check the border case each time when we recurse.

In detail, we do the following: We find the loci of P_1 and P_2 in the suffix tree. As in the previous section, we check τ_0 segment boundaries with $\tau_0 = \Theta(n^{2/3})$ to find all consecutive occurrences with distance within $[\max(\alpha, \lfloor n^{1/3} \rfloor), \beta]$. Now, we only have to find consecutive occurrences of distance within $[\alpha, \min(\beta, \lfloor n^{1/3} \rfloor)]$ in $T = T[0, n-1]$. In general, let $n_i = \lfloor \frac{n}{2^i} \rfloor$ and $\beta_i = \min(\beta, \lfloor n_i^{1/3} \rfloor)$ and let $T[a,b]$ be an induced subtree of level i.

To find all consecutive occurrences with distance within $[\alpha, \beta_i]$ in $T[a,b]$ of level i, given the loci of P_1 and P_2 in $T[a,b]$, recursively do the following:

- If any of the loci is not on a spine of a cluster, we find all consecutive occurrences using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$ and check for each of them if they are valid; we report all such, then terminate.
- Else, we use the query algorithm for small distances from Section 3.3 to decide whether a valid occurrence with distance within $[\alpha, \beta_i]$ exists in $T[a,b]$.

  If such a valid occurrence exists, we recurse; that is, set $c = \lfloor \frac{a+b}{2} \rfloor$. We use $\mathsf{RangePredecessor}$ to find the last occurrence of P_1 before and including c, and $\mathsf{RangeSuccessor}$ to find the first occurrence of P_2 after c. Then we check if they are consecutive (again using $\mathsf{RangePredecessor}$ and $\mathsf{RangeSuccessor}$),

and if it is a valid occurrence. If yes, we add it to the output. Then, for both $S[a, c]$ and $S[c+1, b]$, we implicitly partition them into segments of size $\lfloor n_{i+1}^{1/3} \rfloor$ and find and output all valid occurrences of distance $> n_{i+1}^{1/3}$. Then we follow pointers to the successor nodes of the current loci to find the loci of P_1 and P_2 in the children trees $T[a, c]$ and $T[c+1, b]$ and recurse on those trees to find all consecutive occurrences of distance within $[\alpha, \beta_{i+1}]$

Correctness. At any point before we recurse on level i, we check all consecutive occurrences of distance $> n_{i+1}^{1/3}$ by segmenting the current substring of S. By the arguments of the previous section, we will find all such valid occurrences. Thus, on the subtrees of level $i+1$, we need only care about consecutive occurrences with distance in $[\alpha, \beta_{i+1}]$.

By the properties of the induced suffix tree decomposition, a consecutive occurrence of P_1 and P_2 that is present in $T[a, b]$ will either be fully contained in $T[a, c]$, or in $T[c+1, b]$, or the occurrence of P_1 is the last occurrence before and including c and the occurrence of P_2 is the first occurrence after c. We check the border case each time we recurse. Thus, no consecutive occurrences get lost when we recurse. If we stop the recursion, it is either because one of the loci was in a small subtree or that no valid occurrences with distance within $[\alpha, \beta_i]$ exists in $T[a, b]$. In the first case we found all valid occurrences with distance within $[\alpha, \beta_i]$ in $T[a, b]$ by the same arguments as in Section 3.3. Thus, we find all valid occurrences of P_1 and P_2.

Time Analysis. For finding the loci, we first spend $O(|P_1| + |P_2|)$ time in the initial suffix tree $T[0, n-1]$; after that, we spend constant time each time we recurse to follow pointers. The rest of the time consumption is dominated by the number of queries to the orthogonal range successor data structure, which we will count next.

Consider the recursion part of the algorithm as a traversal of the decomposition tree, and consider the subtree of the decomposition tree we traverse. Each leaf of that subtree is a node where we stop recursing. Since we only recurse if we know there is an occurrence to be found, there are at most $O(\text{occ})$ leaves. Thus, we traverse at most $O(\text{occ} \log n)$ nodes.

Each time we recurse, we spend a constant number of RangeSuccessor and RangePredecessor queries to check the border cases. Additionally, we spend $O(n_i^{2/3})$ such queries on each node of level i that we visit in the decomposition tree: For finding the "large" occurrences, and additionally either for reporting everything within a small subtree or doing an existence query. For finding large occurrences, there are $O(n_i^{2/3})$ segments to check. The number of orthogonal range successor queries used for existence queries or reporting within a small subtree is bounded by the number of leaves within a cluster, which is also $O(n_i^{2/3})$.

Now, let x be the number of decomposition tree nodes we traverse and let l_i, $i = 1, \ldots, x$, be the level of each such node. The goal is to bound $\sum_{i=1}^{x} \left(\frac{n}{2^{l_i}}\right)^{2/3}$. By the argument above, $x = O(\text{occ} \log n)$. Note that because the decomposition tree is binary we have that $\sum_{i=1}^{x} \frac{1}{2^{l_i}} \leq \log n$. The number of queries to the orthogonal range successor data structure is thus asymptotically bounded by:

$$\begin{aligned}
\sum_{i=1}^{x} \left(\frac{n}{2^{l_i}}\right)^{2/3} &= n^{2/3} \sum_{i=1}^{x} \left(\frac{1}{2^{l_i}}\right)^{2/3} \cdot 1 \\
&\leq n^{2/3} \left(\sum_{i=1}^{x} \left(\frac{1}{2^{l_i}}\right)^{\frac{2}{3} \cdot \frac{3}{2}}\right)^{2/3} \left(\sum_{i=1}^{x} 1^3\right)^{1/3} \\
&= n^{2/3} \left(\sum_{i=1}^{x} \frac{1}{2^{l_i}}\right)^{2/3} x^{1/3} \\
&= O(n^{2/3} \text{occ}^{1/3} \log n)
\end{aligned}$$

For the inequality, we use Hölder's inequality, which holds for all $(x_1, \ldots, x_k) \in \mathbb{R}^k$ and $(y_1, \ldots, y_k) \in \mathbb{R}^k$

and p and q both in $(1, \infty)$ such that $1/p + 1/q = 1$:

$$\sum_{i=1}^{k} |x_i y_i| \leq \left(\sum_{i=1}^{k} |x_i|^p \right)^{1/p} \left(\sum_{i=1}^{k} |y_i|^q \right)^{1/q} \tag{3.1}$$

We apply (3.1) with $p = 3/2$ and $q = 3$.

Since the data structure of Zhou [Zho16] uses $O(\log \log n)$ time per query, the total running time of the algorithm is $O(|P_1| + |P_2| + n^{2/3} \mathrm{occ}^{1/3} \log n \log \log n)$. This concludes the proof of Theorem 5(ii).

3.5 Lower Bound

We now prove the conditional lower bound from Theorem 6 based on set intersection. We use the framework and conjectures as stated in Goldstein et al. [GKLP17]. Throughout the section, let $\mathcal{I} = S_1, , \ldots, S_m$ be a collection of m sets of total size N from a universe U. The *SetDisjointness problem* is to preprocess $\mathcal{I}$ into a compact data structure, such that given any pair of sets S_i and S_j, we can quickly determine if $S_i \cap S_j = \emptyset$. We use the following conjecture.

Conjecture 1 (Strong SetDisjointness Conjecture)**.** *Any data structure that can answer SetDisjointness queries in t query time must use $\widetilde{\Omega}\left(\frac{N^2}{t^2}\right)$ space.*

3.5.1 SetDisjointness with Fixed Frequency

We define the following weaker variant of the SetDisjointness problem: the *f-FrequencySetDisjointness problem* is the SetDisjointness problem where every element occurs in precisely f sets. We now show that any solution to the f-FrequencySetDisjointness problem implies a solution to SetDisjointness, matching the complexities up to polylogarithmic factors.

Lemma 11. *Assuming the Strong SetDisjointness Conjecture, any data structure answering f-FrequencySetDisjointness queries in time $O(N^\delta)$, for $\delta \in [0, 1/2]$, must use $\widetilde{\Omega}\left(N^{2-2\delta-o(1)}\right)$ space.*

Proof. Assume there is a data structure D solving the f-FrequencySetDisjointness problem in time $O(N^\delta)$ and space $O\left(N^{2-2\delta-\epsilon}\right)$ for constant ϵ with $0 < \epsilon < 1$. Let $\mathcal{I} = S_1, \ldots, S_m$ be a given instance of SetDisjointness, where each S_i is a set of elements from universe U, and assume w.l.o.g. that m is a power of two.

Define the *frequency* of an element, f_e, as the number of sets in $\mathcal{I}$ that contain e. We construct $\log m$ instances $\mathcal{I}_1, \ldots, \mathcal{I}_{\log m}$ of the f-FrequencySetDisjointness problem. The instances are sorted such that $\mathcal{I}_1$ handles the least frequent elements in $\mathcal{I}$, while $\mathcal{I}_{\log m}$ handles the most frequent elements. More precisely, for each j, $1 \leq j \leq \log m$, the instance $\mathcal{I}_j$ contains the following sets:

- For each $i \in [1, m]$ a set S_i^j containing all $e \in S_i$ that satisfy $2^{j-1} \leq f_e < 2^j$;
- 2^{j-1} "dummy sets", which contain extra copies of elements to make sure that all elements have the same frequency. That is, we add every element with $2^{j-1} \leq f_e < 2^j$ to the first $2^j - f_e$ dummy sets. These sets will not be queried in the reduction.

Clearly, $S_i = \bigcup_j S_i^j$. Instance $\mathcal{I}_j$ has $O(m)$ sets and every element occurs exactly 2^j times. Further, the total number of elements in all the instances is at most $2N$. We now build f-FrequencySetDisjointness data structures $D_j = D(\mathcal{I}_j)$ for each of the $\log m$ instances.

To answer a SetDisjointness query for two sets S_{i_1} and S_{i_2}, we query D_j for the sets $S_{i_1}^j$ and $S_{i_2}^j$, for each $1 \leq j \leq \log m$. If there exists a j such that $S_{i_1}^j$ and $S_{i_2}^j$ are not disjoint, we output that S_i and S_j are not disjoint. Else, we output that they are disjoint.

$$S_1 = \{1, 2\} \qquad w_1 = 00$$
$$S_2 = \{3, 4\} \qquad w_2 = 01$$
$$S_3 = \{1, 3\} \qquad w_3 = 10$$
$$S_4 = \{2, 4\} \qquad w_4 = 11$$

$$S = \underbrace{00\$10\$}_{1}\ \$\$\$\$\$\$\ \underbrace{00\$11\$}_{2}\ \$\$\$\$\$\$\ \underbrace{01\$10\$}_{3}\ \$\$\$\$\$\$\ \underbrace{01\$11\$}_{4}\ \$\$\$\$\$\$$$

Figure 3.3: Instance of f-FrequencySetDisjointness problem reduced to Exists. Alphabet $\Sigma = \{0, 1\}$ and fixed frequency $f = 2$, resulting in block size $B = 2 \cdot 2 + 2 = 6$.

If there exists $e \in S_{i_1} \cap S_{i_2}$, let j be such that $2^{j-1} \leq f_e < 2^j$. Then $e \in S^j_{i_1} \cap S^j_{i_2}$, and we will correctly output that the sets are not disjoint. If S_{i_1} and S_{i_2} are disjoint, then, since $S^j_{i_1}$ is a subset of S_{i_1} and $S^j_{i_2}$ is a subset of S_{i_2}, the queried sets are disjoint in every instance. Thus we also answer correctly in this case.

Let N_j denote the total number of elements in $\mathcal{I}_j$. For each j, we have $N_j \leq 2N$ and thus $N_j^{2-2\delta-\epsilon} \leq (2N)^{2-2\delta-\epsilon}$. Thus, the space complexity is asymptotically bounded by

$$\sum_{j=1}^{\lceil \log m \rceil} N_j^{2-2\delta-\epsilon} = O(N^{2-2\delta-\epsilon} \log m).$$

Similarly, we have $N_j^{\delta} = O(N^{\delta})$ and so the time complexity is asymptotically bounded by

$$\sum_{j=1}^{\lceil \log m \rceil} N_j^{\delta} = O(N^{\delta} \log m).$$

This is a contradiction to Conjecture 1.

□

3.5.2 Reduction to Gapped Indexing

We can reduce the f-FrequencySetDisjointness problem to Exists queries of the gapped indexing problem: Assume we are given an instance of the f-FrequencySetDisjointness problem with a total of N elements. Each distinct element occurs f times. Assume again w.l.o.g. that the number of sets m is a power of two. Assign to each set S_i in the instance a unique binary string w_i of length $\log m$. Build a string S as follows: Consider an arbitrary ordering $e_1, e_2, ...$ of the distinct elements present in the f-FrequencySetDisjointness instance. Let \$ be an extra letter not in the alphabet. The first $B = f \cdot \log m + f$ letters are a concatenation of $w_i\$$ of all sets S_i that e_1 is contained in, sorted by i. This block is followed by B copies of \$. Then, we have B symbols consisting of the strings for each set that e_2 is contained in, again followed by B copies of \$, and so on. See Figure 3.3 for an example.

For a query for two sets S_i and S_j, where $i < j$, we set $P_1 = w_i$ and $P_2 = w_j$, $\alpha = 0$, and $\beta = B$. If the sets are disjoint, then there are no occurrences which are at most B apart. Otherwise w_i and w_j occur in the same block, and w_j comes after w_i. The length of the string S is $2N \log m + 2N$: In the block for each element, we have $\log m + 1$ letters for each of its occurrences, and it is followed by a \$ block of the same length.

This means that if we can solve Exists queries in $s(n)$ space and $t(n) + O(|P_1| + |P_2|)$ time, where n is the length of the string, we can solve the f-FrequencySetDisjointness problem in $s(2N \log m + 2N)$ space and $t(2N \log m + 2N) + O(\log m)$ time. Together with Lemma 11, Theorem 6 follows.

3.6 Gapped Indexing for $[0, \beta]$ Gaps

In this section, we consider the special case where the queries are one sided intervals of the form $[0, \beta]$. We give a data structure supporting the following tradeoffs:

Theorem 7. *Given a string of length n, we can*

(i) *construct an $O(n)$ space data structure that supports* Exists$(P_1, P_2, 0, \beta)$ *queries in $O(|P_1| + |P_2| + \sqrt{n}\log^{\epsilon} n)$ time for constant $\epsilon > 0$, or*

(ii) *construct an $O(n \log n)$ space data structure that supports* Count$(P_1, P_2, 0, \beta)$ *and* Report$(P_1, P_2, 0, \beta)$ *queries in $O(|P_1| + |P_2| + (\sqrt{n \cdot \text{occ}}) \log\log n)$ time, where* occ *is the size of the output.*

Note that since the results match (up to log factors) the best known results for set intersection, this is about as good as we can hope for. We mention here that for this specific problem, a similar tradeoff follows from the strategies used by Hon et al. [HSTV14]. The results from that paper include (among others) a data structure for documents such that given a query of two patterns P_1 and P_2 and a number k, one can output the k documents with the closest occurrences of P_1 and P_2. Thus, the problem is slightly different, however, with some adjustments, the results from Theorem 7 follow (up to a log factor). We show a simple, direct solution.

The data structure is a simpler version of the data structure considered in the previous sections. The main idea is that for each pair of boundary nodes u and v, we do not have to store an array of distances, but only one number that carries all the information: the smallest distance of a consecutive occurrence of $\text{str}(u)$ and $\text{str}(v)$. Thus, for existence, we can cluster with $\tau = \sqrt{n}$ to achieve linear space, and we do not need to check large distances separately. For the reporting solution, we store the decomposition from Section 3.4.1, and use the matrix M to decide where to recurse. In the following we will describe the details.

Existence data structure. For solving Exists queries in this setting, we cluster the suffix tree with parameter $\tau = \sqrt{n}$. Again, we store the linear space orthogonal range successor data structure by Nekrich and Navarro [NN12] on the suffix array. For each pair of boundary nodes (u, v), we store at $M(u, v)$ the minimum distance of a consecutive occurrence of $\text{str}(u)$ and $\text{str}(v)$. The total space is linear. To query, we proceed similarly as in Section 3.3 for the "small distances": We find the loci of P_1 and P_2. If any of the loci is not on the spine, we check all consecutive occurrences using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$. If both loci are on the spine, denote b_1, b_2 the lower boundary nodes of the respective clusters. Find $M(b_1, b_2)$. If $M(b_1, b_2) \leq \beta$, we can immediately return YES: If a valid occurrence (i', j') of $\text{str}(b_1)$ and $\text{str}(b_2)$ exists, then either (i', j') is a consecutive occurrence of P_1 and P_2, or there exists a consecutive occurrence of smaller distance. Otherwise, that is if $M(b_1, b_2) > \beta$, all valid occurrences (i, j) have the property that either i is in the cluster of $\text{locus}(P_1)$ or j is in the cluster of $\text{locus}(P_2)$, and we check all such pairs using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$. The running time is $O(|P_1| + |P_2| + \sqrt{n}\log^{\epsilon} n)$.

Reporting data structure. For the reporting data structure, we store the decomposition of the suffix tree as described in Section 3.4.1 and the $O(n \log n)$ space orthogonal range successor data structure by Zhou [Zho16] on the suffix array. For each induced subtree of level i in the decomposition, we store the existence data structure we just described.

Reporting algorithm. The algorithm follows a similar, but simpler, recursive structure as in Section 3.4. We begin by finding the loci of P_1 and P_2. If either of the loci is not on a spine, we find all consecutive occurrences using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$, check if they are valid, report these, and terminate. If both loci are on a spine, we check $M(b_1, b_2)$ for the lower boundary nodes b_1 and b_2. If $M(b_1, b_2) > \beta$, all valid occurrences (i, j) have the property that either i is in the cluster of $\text{locus}(P_1)$ or j is in the cluster of $\text{locus}(P_2)$. We check all such pairs using $\mathsf{FindConsecutive}_{P_2}(.)$ resp. $\mathsf{FindConsecutive}_{P_1}(.)$, report the valid occurrences, and terminate. If $M(b_1, b_2) \leq \beta$, we recurse on the children trees. That is, we check the border case and follow pointers to the loci in the children trees.

Analysis. The space is $O(n \log n)$, just as in Section 3.4.

For time analysis, we spend $O(\sqrt{\frac{n}{2^{l_i}}})$ orthogonal range successor queries on the nodes in the decomposition tree of level l_i where we stop the recursion. For all other nodes we visit in the tree traversal, we only spend a constant number of queries. In total, we visit $O(\text{occ} \log(n/\text{occ}) + \text{occ})$ decomposition tree nodes (by following the analysis in [CP10]), and we spend $O(\sqrt{\frac{n}{2^{l_i}}})$ orthogonal range successor queries on $O(\text{occ})$ many such nodes.

We use the same notation as in Section 3.4. By $x = O(\text{occ})$ we now denote the number of nodes where we stop the algorithm and output. Since each such node can be seen as a leaf in a binary tree, $\sum_{i=1}^{x} \frac{1}{2^{l_i}} \leq 1$. We use the Cauchy-Schwarz inequality (which is a special case of Hölders with $p = q = 2$). We get as an asymptotic bound for the number of orthogonal range successor queries:

$$\begin{aligned}\sum_{i=1}^{x} \sqrt{\frac{n}{2^{l_i}}} &= \sqrt{n} \sum_{i=1}^{x} \sqrt{\frac{1}{2^{l_i}}} \cdot 1 \\ &\leq \sqrt{n} \sqrt{\sum_{i=1}^{x} \frac{1}{2^{l_i}}} \sqrt{\sum_{i=1}^{x} 1} \\ &\leq \sqrt{nx} = O(\sqrt{n \cdot \text{occ}}).\end{aligned}$$

Note that since $\text{occ} \log(n/\text{occ}) = O(\text{occ}\sqrt{n/\text{occ}}) = O(\sqrt{n \cdot \text{occ}})$, this brings the total number of orthogonal range successor queries to $O(\text{occ} + \sqrt{n \cdot \text{occ}})$. Using the data structure by Zhou [Zho16], the time bound from Theorem 7 follows.

3.7 Conclusion

We have considered the problem of gapped indexing for consecutive occurrences. We have given a linear space data structure that can count the number of such occurrences. For the reporting problem, we have given a near-linear space data structure. The running time for both includes an $O(n^{2/3})$ term, which forms a gap of $O(n^{1/6})$ to the conditional lower bound of $O(\sqrt{n})$. Thus, the most obvious open question is whether we can close this gap, either by improving the data structure or finding a stronger lower bound.

Further, we have used the property that there can only be few consecutive occurrences of large distances. Thus, our solution cannot be easily extended to finding *all* pairs of occurrences with distance within the query interval. An open question is if it is possible to get similar results for that problem. Lastly, document versions of similar problems have concerned themselves with finding all documents that contain P_1 and P_2 or the top-k of smallest distance; conditional lower bounds for these problems are also known. It would be interesting to see if any of these results be extended to finding all documents that contain a (consecutive) occurrence of P_1 and P_2 that has a distance within a query interval.

Chapter 4

Sliding Window String Indexing in Streams

Sliding Window String Indexing in Streams

Philip Bille
Technical University of Denmark
phbi@dtu.dk

Inge Li Gørtz
Technical University of Denmark
inge@dtu.dk

Max Pedersen
Technical University of Denmark
mhrpe@dtu.dk

Tord Joakim Stordalen
Technical University of Denmark
tjost@dtu.dk

Abstract

Given a string S over an alphabet Σ, the *string indexing problem* is to preprocess S to subsequently support efficient pattern matching queries, that is, given a pattern string P report all the occurrences of P in S. In this paper we study the *streaming sliding window string indexing problem*. Here the string S arrives as a stream, one character at a time, and the goal is to maintain an index of the last w characters, called the *window*, for a specified parameter w. At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and all occurrences of P within the current window must be returned. The streaming sliding window string indexing problem naturally captures scenarios where we want to index the most recent data (i.e. the window) of a stream while supporting efficient pattern matching.

Our main result is a simple $O(w)$ space data structure that uses $O(\log w)$ time with high probability to process each character from both the input string S and any pattern string P. Reporting each occurrence of P uses additional constant time per reported occurrence. Compared to previous work in similar scenarios this result is the first to achieve an efficient worst-case time per character from the input stream with high probability. We also consider a delayed variant of the problem, where a query may be answered at any point within the next δ characters that arrive from either stream. We present an $O(w+\delta)$ space data structure for this problem that improves the above time bounds to $O(\log(w/\delta))$. In particular, for a delay of $\delta = \epsilon w$ we obtain an $O(w)$ space data structure with constant time processing per character. The key idea to achieve our result is a novel and simple hierarchical structure of suffix trees of independent interest, inspired by the classic log-structured merge trees.

4.1 Introduction

The *string indexing problem* is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P, report all occurrences of P within S. In this paper, we introduce a basic variant of string indexing called the *streaming sliding window string indexing (SSWSI) problem*. Here, the string S arrives as a stream one character at a time, and the goal is to maintain an index of a *window* of the last w characters, for a specified parameter w. At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and we need to report the occurrences of P within the current window. The goal is to compactly maintain the index while processing the characters arriving in either stream efficiently. We consider two variants of the problem: a *timely* variant where each query must be answered immediately, and a *delayed* variant where it may be answered at any point within the next δ characters arriving from either stream, for a specified parameter δ. See Section 4.1.1 for precise definitions.

The SSWSI problem naturally captures scenarios where we want to index the most recent data (i.e. the window) of a stream while supporting efficient pattern matching. For instance, monitoring a high-rate data stream system where we cannot feasibly index the entire stream but still want to support efficient queries. Depending on the specific system we may require immediate answers to queries, or we may be able to afford a delay that allows for more efficient queries and updates.

The SSWSI problem has not been explicitly studied before in our precise formulation, but for the timely variant several closely related problem are well-studied. In particular, the *sliding window suffix tree problem* [FG89,Lar99,Sen05,BJ18,NAIP03] is to maintain the *suffix tree* of the current window (i.e., the compact trie of the suffixes of the window) as each character arrives. With appropriate augmentation the suffix tree can be used to process pattern matching queries efficiently, leading to a solution to the timely SSWSI problem. For constant-sized alphabets, the best of these solutions [BJ18] maintains the sliding window suffix tree in constant *amortized* time per character while supporting efficient pattern matching queries. The worst-case time for updates is $\Omega(w)$. The other solutions achieve similar amortized time bounds. This amortization cannot be avoided since explicitly maintaining the suffix tree after the arrival of a new character may incur $\Omega(w)$ changes.

Another closely related problem is the *online string indexing problem* [AKLL05, Kop12, BI13, Kos94, AN08, KN17, FG05, AFG+14]. Here the goal is to process S one character at a time (in either left-to-right or right-to-left order), while incrementally building an index of the string read so far. The best of these solutions update the index in either constant time per character for constant-sized alphabets [KN17] or $O(\log\log n + \log\log|\Sigma|)$ time for any alphabet where each character fits in a constant number of machine words [Kop12]. These solutions all heavily rely on processing the string in right-to-left order to avoid the inherent linear time suffix tree updates due to appending, as mentioned above. Therefore they cannot be applied in our left-to-right streaming setting. Alternatively, we can instead apply these solutions on the reverse of the string S, but then each pattern must be processed in reverse order, which also cannot be done in our setting. Also, note that these solutions index the entire string read so far. It is not clear if they can be adapted to efficiently index a sliding window.

Another line of work shows how to maintain a fully dynamic suffix array under insertions and deletions [AB20, AB21, SLLM10, KK22]. These can also be used to solve SSWSI but are more general and lead to polylogarithmically slower bounds than our results while being more complicated.

Our main result is an efficient and simple solution to the SSWSI problem in both the timely and delayed variant. Let w denote the size of the window. For the timely variant, we present a string index that uses $O(w)$ space and processes a character from the stream S in $O(\log w)$ time. Each pattern matching query P is also supported in $O(\log w)$ time per character with additional $O(\text{occ})$ time incurred after receiving the last character of P, where occ is the number of occurrences of P in the current window. The index is randomized and both time bounds hold with high probability. Compared to previous suffix tree based approaches for indexing a sliding window, we improve the worst-case time bounds per character in the stream from $\Omega(w)$ to $O(\log w)$ with high probability. This is particularly important in the above mentioned applications, such as high-rate data stream systems. Our solution generalizes to the delayed variant of the problem. If we allow a delay of δ before answering each query we achieve $O(w+\delta)$ space while improving the above time bounds to $O(\log(w/\delta))$. In particular, if we allow a delay of $\delta = \epsilon w$ for any constant $\epsilon > 0$, we achieve linear space and optimal constant time (reporting the occurrences still takes $O(\text{occ})$ time, and we do not count the reporting time towards the delay). Note that $\delta \leq w$ is sufficient delay for optimal time bounds and we can assume $O(w+\delta) = O(w)$. The results hold on a word RAM and for any alphabet size, assuming that each character fits into a constant number of machine words.

The key idea to achieve our result is a novel and simple hierarchical structure of suffix trees inspired by log-structured merge trees [OCGO96b]. Instead of maintaining a single suffix tree on the window we maintain a collection of suffix trees of exponentially increasing sizes that cover the current window. We show how to efficiently maintain the structure as new characters from the stream arrive by incrementally "merging" suffix trees, while supporting efficient pattern matching queries within the window.

Our solution uses randomization to construct suffix trees in linear time with high probability. Plugging in a deterministic construction algorithm such as the one by Ukkonen [Ukk95], we obtain a solution

using $O(\log w \log |\Sigma|)$ time for both queries and updates. With more recent deterministic suffix tree solutions [FG05, CKL15, BGS17] we can improve this to obtain $O(\log w \log \log n)$ time per character for both queries and updates. Note that the $O(\log \log |\Sigma|)$ in the time bounds of [FG05] has been replaced by $O(\log \log n)$ here due to an additional sorting step using [Han02].

4.1.1 Setup and Results

We formally define the problem as follows. Let S be a stream over any alphabet Σ where each character fits in a constant number of machine words. For given integer parameters $w \geq 1$ and $\delta \geq 0$, the *δ-delayed streaming sliding window string indexing ((w, δ)-SSWSI)* problem is to maintain a data structure that, after receiving the first i characters of S, supports

- Report(P): report all the occurrences of P in $S[i - w + 1, i]$ before an additional δ characters have arrived from either stream.
- Update(): process the next character in the stream S.

In the Report(P) query the pattern string P is also streamed. When P is streamed it interrupts the stream S, arrives one character at a time, and all characters of P arrive before the streaming of S resumes. Furthermore, we do not assume that we know the length of P before the arrival of its last character. Although P is streamed we assume random access to its characters after they arrive, as any pattern that fits in the window is at most w characters long and we can afford to store it. The delay is counted from after the last character of P arrives. Characters from S and from new patterns count towards the delay, while reported occurrences do not (otherwise it would be impossible to answer the query in time if there are more than δ occurrences).

We define the *timely streaming sliding window string indexing (w-SSWSI) problem* to be $(w, 0)$-SSWSI, that is, queries must be answered immediately as the last character of the pattern arrives.

We show the following general main result.

Theorem 8. *Let S be a stream and let $w \geq 1$ and $\delta \geq 0$ be integers. We can solve the (w, δ)-SSWSI problem on S with an $O(w+\delta)$ space data structure that supports Update and Report in $O(\log \frac{w}{\delta+1})$ time per character with high probability. Furthermore, Report uses additional worst-case constant time per reported occurrence.*

Here, with high probability means with probability at least $1 - \frac{1}{w^d}$ for any constant d. Theorem 8 provides a trade-off in the delay parameter δ. In particular, plugging in $\delta = 0$ in Theorem 8 we obtain a solution to the timely SSWSI problem that uses $O(w)$ space and $O(\log w)$ time per character for both Update and Report. Compared to the previous work on sliding window stream indexing [FG89, Lar99, Sen05, BJ18, NAIP03, ISTA04, SD08] this improves the worst-case bounds on the Update operation from $\Omega(w)$ to $O(\log w)$ with high probability and also removes the restriction on the alphabet. At the other extreme, plugging in $\delta = \epsilon w$ for constant $\epsilon > 0$ in Theorem 8 we obtain a solution to the delayed SSWSI problem that uses $O(w)$ space and optimal constant time per character with high probability. All our results hold on a word RAM where each machine word has at least $\log w$ bits, and where each character of the alphabet fits into a constant number of machine words.

4.1.2 Techniques

We obtain our result for the timely variant, but without high probability guarantees, as follows. At all times we maintain at most $\log w$ suffix trees that do not overlap and together cover the window. The trees are organized by the *log-structured merge technique* [OCGO96b], where the rightmost tree is the smallest and their sizes increase exponentially towards the left. For each new character that arrives we append its suffix tree to the right side of our data structure. Whenever there are two trees of the same size next to each other we "merge" them by constructing a new suffix tree covering them both. Each character from S is involved in at most $\log w$ merges and each merge takes expected linear time, so we spend expected amortized $O(\log w)$ time per character in S. We deamortize the updates by temporarily keeping both trees while merging them

in the background. Note that for each adjacent pair of suffix trees we also store a suffix tree approximately covering them both, referred to as *boundary trees* (see details below).

We find the occurrences of a pattern P in the window by querying each of these trees, which takes $O(\log w)$ time per character in P. For adjacent pairs of trees larger than $|P|$ we find the occurrences of P crossing from one into the other using the boundary trees. The remaining trees cover a suffix of the window of length $O(|P|)$, and we grow a suffix tree to answer queries in this suffix *at query time*. Our data structure has some "overhang" on the left side of the window, and we use range maximum queries to report only the occurrences that start inside the window.

This solution is generalized to incorporate a delay of δ as follows. We store the $O(\log(w/\delta))$ largest trees from the timely solution and leave a suffix of size $\Theta(\delta)$ of the window uncovered by suffix trees. We answer queries as follows. If $|P| > \delta/4$ we say that P is *long*, and otherwise it is *short*. For long patterns we do as in the timely case; the suffix tree we grow at query time now must also contain the uncovered suffix, but it still has size $O(|P|)$ since the uncovered part of the window has length $O(\delta) = O(|P|)$. We show how to do this in $O(\log(w/\delta))$ time per character in P. For short patterns we utilize that they are smaller than the delay to temporarily buffer the queries and later batch process them. We buffer up to $O(\delta \log(w/\delta))$ work and deamortize it over $\Theta(\delta)$ characters, obtaining the same bound as for long patterns. Updates run in the same bound since each character from S is involved in at most $O(\log(w/\delta))$ merges before it leaves the window.

Finally, we improve the time bounds by proving that for any substring S' of our window, we can construct the suffix tree over S' in $O(|S'|)$ time with probability $1 - w^{-d}$ for any constant $d > 1$. We do so by reducing the alphabet $\Sigma' = \{c \in S'\}$ of S' to rank-space $\{1, 2, \ldots, |\Sigma'|\}$ from which the algorithm by Farach-Colton et al. [FFM00] can construct the suffix tree in worst-case linear time. For large strings ($|S'| > w^{1/5}$) we pick a hash function from $\Sigma \to [0, w^c]$ that with high probability is injective on S', and then we use radix sort to reduce to rank-space in linear time. For small strings ($|S'| \leq w^{1/5}$) we pick a hash function from $\Sigma \to [0, w/\log w]$ that is injective with (almost) high probability, and use this to manually construct a mapping into rank space in $O(S')$ time. This mapping algorithm uses additional $O(w/\log w)$ space, but we construct at most $O(\log w)$ suffix trees at any time so the total space is linear.

4.1.3 Outline

In Section 4.2 we cover the preliminaries, including some useful facts about suffix trees. In Section 4.3 we give a solution to the timely SSWSI problem that supports each operation in expected logarithmic time per character. In Section 4.4 we show how to generalize this to incorporate delay, and in Section 4.5 we show how to get good probability guarantees, proving Theorem 8.

4.2 Preliminaries

Given a string X of length n over an alphabet Σ, the ith character is denoted $X[i]$ and the substring starting at $X[i]$ and ending at $X[j]$ is denoted $X[i, j]$. The substrings of the form $X[i, n]$ are the *suffixes* of X.

A *segment* of X is an interval $[i, j] = \{i, i + 1, \ldots, j\}$ for $1 \leq i \leq j \leq n$. We will sometimes refer to segments as strings, i.e., the segment $[i, j]$ refers to the string $X[i, j]$. The definition differs from "substring" by being specific about position; even if $X[1, 2] = X[3, 4]$ we have $[1, 2] \neq [3, 4]$. A *segmentation* of X is a decomposition of X into disjoint segments that cover it. For instance, $x_1 = [1, i]$ and $x_2 = [i + 1, n]$ is a segmentation of X into two parts. The two segments x_1 and x_2 are *adjacent* since x_2 starts immediately after x_1 ends, and for a pair of adjacent segments we define the *boundary* (x_1, x_2) to be the implicit position between i and $i + 1$.

The *suffix tree* [Wei73] T over X is the compact trie of all suffixes of $X\$$, where $\$ \notin \Sigma$ is lexicographically smaller than any letter in the alphabet. Each leaf corresponds to a suffix of X, and the leaves are ordered from left to right in lexicographically increasing order. The suffix tree uses $O(n)$ space by implicitly representing the string associated with each edge using two indices into X. Farach-Colton et al. [FFM00] show that the optimal construction time for T is $\text{sort}(n, |\Sigma|)$, i.e., the time it takes to sort n elements from the universe Σ. For alphabets of the form $\Sigma = \{0, \ldots, n^c\}$ for constant $c \geq 1$ this implies that T can be built in worst-case

$O(n)$ time using radix sort. For larger alphabets we can reduce to the polynomial case in expected linear time using hashing, building T in expected linear time (see Section 4.5 for details).

The *suffix array* L of X is the array where $L[i]$ is the starting position of the ith lexicographically smallest suffix of X. Note that $L[i]$ corresponds to the ith leaf of T in left-to-right order. Furthermore, let v be an internal node in T and let s_v be the string spelled out by the root-to-v path. The descendant leaves of v exactly correspond to the suffixes of X that start with s_v, and these leaves correspond to a consecutive range $[\alpha, \beta]_v$ in L.

We augment the suffix tree to support efficient pattern matching queries as follows. First, we use the well-known FKS perfect hashing scheme [FKS84] to store the edges of the suffix tree, so we can for any node determine if there is an outgoing edge matching a character $a \in \Sigma$ in worst-case constant time. Note that this construction takes *expected* linear time. Furthermore, we also build a *range maximum query* data structure over L. This data structure supports range maximum queries, i.e., given a range $[\alpha, \beta]$ return the $j \in [\alpha, \beta]$ maximizing $L[j]$. It also supports range minimum queries, defined analogously. The data structure can be built in linear time and supports queries in constant time [GBT84]. Finally, we preprocess the suffix tree in linear time such that each internal node v stores the range $[\alpha, \beta]_v$ into L corresponding to the occurrences of s_v.

We can use this structure to efficiently find all the occurrences of P in $O(|P| + \text{occ})$ time, where occ is the number of occurrences, or the leftmost and rightmost occurrence of P in $O(|P|)$ time. The *locus* of a string P is the minimum depth node v such that P is a prefix of s_v. First we find the locus by walking downwards in the suffix tree, matching each character in P in worst-case constant time using the dictionary. Once we have found v we can report all the occurrences in $[\alpha, \beta]_v$ in $O(\text{occ})$ time. Alternatively, we can find the rightmost occurrence of P in constant time by doing a range maximum query on the range $[\alpha, \beta]_v$ in L, which returns the $j \in [\alpha, \beta]_v$ maximizing the *string position* $L[j]$. We can also find the leftmost occurrence by doing a range minimum query.

Finally, note that it is possible to deamortize algorithms with *expected* running time using the standard technique of distributing the work evenly. Specifically, if an algorithm runs in expected λn time we can do λ work for $n - 1$ steps; by linearity of expectation only expected λ work remains for the last step.

4.3 The Timely SSWSI Problem

Here we present a solution for the timely variant that matches the bounds in Theorem 8 in expectation. Section 4.5 shows how to get the bounds with high probability. Throughout this section we assume without loss of generality that w is a power of two. Section 4.3.3 briefly mentions how to generalize to arbitrary w.

The main idea is as follows. We maintain a suffix of S of length at least w. This suffix is segmented into at most $\log w$ segments whose sizes are distinct powers of two, in increasing order from right to left. The length of the suffix we store is at most $2^0 + \ldots + 2^{\log w} = 2w - 1$. When a new character arrives, we append a new size-one segment to our data structure and merge equally-sized segments until they all have distinct sizes again. We also discard the largest segment when it no longer intersects the window. For each segment we store a suffix tree, and for every pair of adjacent segments we store a *boundary tree* approximately covering them both (see below). To support queries we query the suffix tree for each individual segment, and also each boundary tree. For the segments larger than the pattern, the boundary trees are sufficient to find the occurrences crossing the respective boundary. The remaining trees cover a suffix of S that is $O(|P|)$ long, and we grow a suffix tree at query time to find the remaining occurrences in this suffix.

4.3.1 Data Structure

At any point, the data structure contains a suffix s of S of length $w \leq |s| \leq 2w - 1$ and a segmentation of s into at most $\log w$ segments. Specifically, if $|s| = 2^{b_1} + \ldots + 2^{b_k}$ for integers $b_1 < \ldots < b_k$ then we have the segmentation $s_1, \ldots, s_k$ where $|s_i| = 2^{b_i}$, and s is the concatenation of the strings $s_k, s_{k-1}, \ldots, s_1$, in that order. The set $\{b_1, \ldots, b_k\}$ is unique and corresponds to the 1-bits in the binary encoding of $|s|$. Three different configurations can be seen in Figure 4.1.

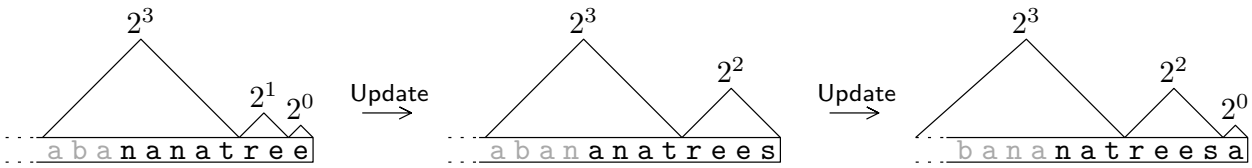

Figure 4.1: Example of updating the data structure with a window size of $w = 8$. Here we illustrate the segments by the suffix trees built over them. Characters outside of the window are gray. As the character `s` arrives we construct a new suffix tree of size one, which is then immediately merged with the existing size-one suffix tree over `e` into a size-two suffix tree over `es`, which is then merged to into the final size-four suffix tree over `rees`. After receiving `a` we again have a size-one suffix tree. Note that after three more updates the suffix tree of size eight will no longer overlap the window and will be discarded.

For each segment s_i we store the suffix tree T_i over s_i, along with a range maximum query data structure over the suffix array of s_i. For each boundary (s_{i+1}, s_i) we store the *boundary tree* B_i, which is the suffix tree over the substring centered at the boundary and extending $|s_i|$ characters in both directions. We augment B_i with an additional data structure that we will use for reporting occurrences across the boundary. Let BL_i be the suffix array corresponding to B_i. We define the *modified suffix array* BL'_i as

$$BL'_i[j] = \begin{cases} BL_i[j] & \text{if } BL_i[j] \text{ corresponds to a suffix starting in } s_{i+1} \\ -\infty & \text{if } BL_i[j] \text{ corresponds to a suffix starting in in } s_i \end{cases}$$

We store a range maximum query data structure over BL'_i. Each of the data structures use $O(s_i)$ space, so the whole data structures uses $O(s) = O(w)$ space.

We note a few properties of the data structure. Let $S[n]$ be the most recent character to arrive and let $W_n = S[n - w + 1, n]$ be the current window. Then W_n is a suffix of s since $|s| \geq w$. The largest, and leftmost, segment s_k always has size $2^{\log w} = w$; it is not larger since $\log w$ bits are sufficient to represent $|s| \leq 2w - 1$, and it is always there since $|s| \geq w$ cannot be represented with $\log w - 1$ bits. For the same reason, s_k always intersects at least partially with W_n, and each of $s_1, \ldots, s_{k-1}$ are fully contained in W_n.

4.3.2 Queries

The idea is as follows, as exemplified in Figure 4.2. Any occurrence of a pattern P that is fully contained in a segment is found using the suffix tree over that segment. Similarly, any occurrence that only crosses a single boundary far enough away from the end of the window is found in the respective boundary tree. Note that in the leftmost segment we must be careful to not report any occurrences that start before the left window boundary. The remaining occurrences are not contained in any of the trees in the data structure (either because they cross multiple boundaries or because they cross a single boundary (s_{i+1}, s_i) but start more than $|s_i|$ characters to the left of the boundary). However, these occurrences are all located within a substring of size $O(m)$ ending at position $S[n]$, so we build, at query time, a suffix tree to find these occurrences.

Let P be the length-m pattern being queried, $S[n]$ be the most recent character to arrive, and let W_n, the suffix s, the segmentation $s_1, \ldots, s_k$, and the indices $b_1 < \ldots < b_k$ be defined as above. As mentioned, any occurrence of P in W_n must either be fully contained within one of the segments, or it must cross the boundary between two adjacent segments. We will show how to handle each of these cases separately.

Fully Contained in a Segment Fix a specific segment s_i. As each character of P arrives we match it in T_i. When the last character arrives we have a (possibly empty) range $[\alpha, \beta]$ into the suffix array of s_i corresponding to the occurrences of P. If s_i is not the leftmost segment then it is fully contained in W_n and we report all the occurrences. Otherwise, $s_i = s_k$ is the leftmost segment, which might overlap only partially with W_n, and it may contain occurrences of P that are not contained in the window. However, note that the intersection between W_n and s_k is a suffix of s_k. Therefore, if an occurrence of P in s_k *starts* inside W_n it

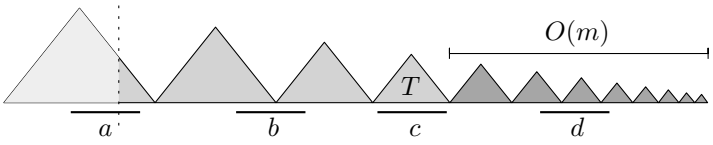

Figure 4.2: Illustration of how we answer queries for a pattern P of length m. The lines denoted a, b, c, and d indicate occurrences of P. The segmentation is illustrated by the trees over the segments. The leftmost window boundary is marked with a vertical dashed line. Note that the leftmost segment intersects only partially with the window. The tree T marks the smallest segment of size at least m. The segments to the right of T are all smaller than m, so they cover at most $m + m/2 + \ldots + 1 = O(m)$ characters. To answer the query we match P in the tree over each segment and in each boundary tree, and we also build a suffix tree over the segments smaller than m at query time. We find b because the respective boundary tree is sufficiently large. We find c because it is fully contained in a segment. We find d in the suffix tree that we build at query time. Note that a is not contained in the window; we avoid reporting it by recursively using range maximum queries to find the *rightmost* occurrence of P in the leftmost segment.

also ends inside W_n. We find all such occurrences as follows. Let L_k be the suffix array of s_k. As described in Section 4.2 we find the index j of the rightmost occurrence of P by doing a range maximum query on the range $[\alpha, \beta]$ in L_k. If $L_k[j]$ is not inside W_n then none of the occurrences are, and we are done. Otherwise we recurse on $[\alpha, j-1]$ and $[\beta, j-1]$. Matching P in the trees of all the segments takes $O(\log w)$ overall time per character of P. Reporting each occurrence takes constant time since range maximum queries run in constant time.

Crossing a Boundary We now show how to report the occurrences of P that span a boundary. The main idea is as follows, as illustrated in Figure 4.3. Let s_i be the smallest segment where $|s_i| \geq m$. Consider any boundary (s_{j+1}, s_j) to the left of s_i, i.e., where $j \geq i$. Since both of these segments have size at least $|s_i| \geq m$, the boundary tree B_j extends at least m characters in both directions from the boundary. Therefore, all the occurrences of P crossing the boundary are contained in B_j, and none of them can cross another boundary as well. Now consider the suffix $\mathcal{R}$ of s containing the $m-1$ last characters of s_i and extending to the end of s. This substring contains all the other boundary-crossing occurrences. Furthermore, all the occurrences in $\mathcal{R}$ cross at least one boundary since the longest consecutive part of a single segment in $\mathcal{R}$ is the $m-1$ characters in s_i. Note that the length of $\mathcal{R}$ is at most $m-1+|s_{i-1}|+|s_{i-1}|/2+\ldots+1 < m-1+2|s_{i-1}| < 3m$ since $|s_{i-1}| < m$. Thus, the number of boundary-crossing occurrences of P equals the number of occurrences in $\mathcal{R}$ plus the number of occurrences crossing the boundaries $(s_k, s_{k-1}), (s_{k-1}, s_{k-2}), \ldots (s_{i+1}, s_i)$.

The algorithm for finding the occurrences in the sufficiently large boundary trees is as follows. Fix a boundary (s_{x+1}, s_x). We match each character of P in B_x as it arrives. When the last character arrives we know if $|s_x| \geq m$, and also the range $[\alpha, \beta]$ corresponding to the occurrences of P in the boundary tree. If $|s_x| \geq m$ (hence $x \geq i$) we report the occurrences as follows. As above we do a range maximum query to find the j maximizing $BL'_x[j]$. If $BL'_x[j] = -\infty$ then all occurrences of P start in s_x, and there are no occurrences crossing the boundary. Otherwise, $BL'_x[j]$ corresponds to the starting position of the rightmost occurrence of P in s_{x+1}. Since all of P has arrived and we now know m, we know that this occurrence crosses the boundary if and only if $BL'_x[j] \geq |s_x| - m + 2$ (recall that B_x extends $|s_x|$ characters in both directions from the boundary). If it does not cross the boundary, then none of the other occurrences do either. Otherwise we report $BL'_x[j]$ and recurse on $[\alpha, j-1]$ and $[j+1, \beta]$ to find the remaining occurrences. Matching P in all boundary trees takes $O(\log w)$ overall time per character, and reporting each occurrence with range maximum queries takes constant time.

We now show how to find the occurrences of P in $\mathcal{R}$ with the same bounds. Assume that we know that $2^\ell \leq m < 2^{\ell+1}$ for some integer ℓ. We build the suffix tree over the last $3 \cdot 2^{\ell+1}$ characters of s, deamortized over receiving the first $2^{\ell-1}$ characters of P. Over the next $2^{\ell-1}$ characters we match P in the tree, at a rate of two characters per new character from P. Then, when the 2^ℓth character arrives, we have caught up to

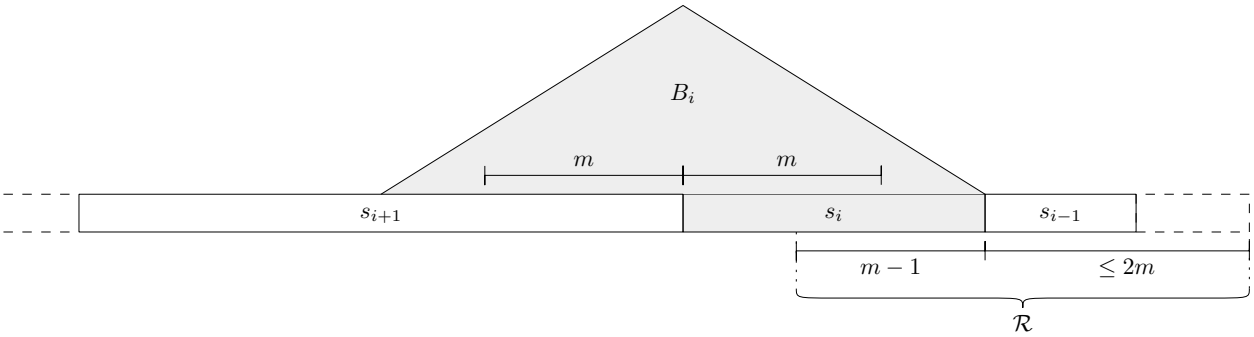

Figure 4.3: The segment s_i is the smallest segment where $|s_i| \geq m$. For each boundary (s_{j+1}, s_j) where $j \geq i$, the tree B_j is large enough to find all occurrences of P across the boundary. All other occurrences of P that cross a boundary must be in $\mathcal{R}$, the string covering the $m-1$ rightmost characters of s_i and extending to the end of the window. The length of $\mathcal{R}$ is no more than $m - 1 + |s_{i-1}| + |s_{i-1}|/2 + \ldots + 1 < 3m$.

the stream P, and we match the remaining $m - 2^\ell$ characters as they arrive. When the last character arrives we have matched P in a tree of size at least $3m$, and we can start reporting occurrences. Note that we are overestimating the size of the tree, and it potentially includes some occurrences of P that are contained in s_i. To avoid reporting these, we also build a range maximum query data structure over the suffix array such that we can use recursive range maximum queries. When deamortized, we construct the tree in expected constant time per character of P. Matching P also takes constant time per character. We know that $m \leq w$, so we run this algorithm simultaneously for each of the $\log w$ different choices for ℓ, using expected $O(\log w)$ time per character in P. Note that the trees use $O(w)$ space in total since the sum of the space is a geometric sum where the largest term is $O(w)$.

4.3.3 Amortized Updates

We show how to support updates in amortized $O(\log w)$ time. Let $S[n]$ be the last character to arrive and as in the description of the data structure let $b_1 < b_2 < \ldots < b_k$ be the positions of the 1-indices in the binary encoding of $|s|$. When the new character $c = S[n+1]$ arrives, we update s and the segmentation $s_1 \ldots s_k$ to create the new suffix s' with the new segmentation $s'_1, \ldots s'_{k'}$. See Figure 4.1 for an example.

If $|s| < 2w - 1$ then we set $s' = sc$. The segmentation of s' corresponds to the unique binary encoding of $|s'| = |s| + 1$, so we update the segmentation analogously to a "binary increment". One way to do so is as follows. We create a new segment of size one over c. If there was not already a segment of size one, then we add the new segment and we are done. Otherwise we *merge* (see below) the two size-one segments to create a segment of size two. The process cascades until we reach a size 2^b that does not exist in the segmentation of s (i.e., the smallest index $b \notin \{b_1, \ldots, b_k\}$). At this point we replace all of the segments $s_{b-1}, \ldots, s_1$ with s'_1 covering the last 2^b characters of s'. The remaining segments for s' are the same as the segments $s_{b+1}, \ldots, s_k$. If $|s| = 2w - 1$ then there is a segment of each size $2^0, 2^1, \ldots, 2^{\log w}$. Since the segments have decreasing size from left to right, the $\log w - 1$ rightmost segments cover the last $2^0 + \ldots + 2^{\log w - 1} = w - 1$ characters of s. Thus, after c arrives, the leftmost segment of size $2^{\log w} = w$ no longer intersects the window. We remove it by setting $s' = s[w+1, |s|]c$, and update the segmentation as above.

Let s_a, s_b and s_c be three adjacent segments, in that order. To *merge* s_b and s_c we combine them into a new segment s_d that spans them both, construct the suffix tree over s_d, and construct a range maximum query data structure on the suffix array of s_d. Furthermore, since s_a and s_d are now adjacent we also construct the boundary-spanning suffix tree for the boundary (s_a, s_d) that extends $|s_d|$ characters in each direction. The construction of all of these data structures takes expected $O(|s_d|)$ time (see Section 4.2). Thus, it takes expected constant time per character every time it moves into a new, larger segment. Each character is contained in at most $\log w$ segments before it leaves the window, so the amortized update time is expected $O(\log w)$ per character.

Note that all but the last merge are unnecessary to actually compute s'_1; in the amortized setting we can simply determine where the cascade will end and immediately construct the suffix tree over the corresponding segment. However, the cascading merges will come into play in the deamortized variant.

Also note that if w is not a power of two we can use a similar scheme where we allow either two simultaneous trees of size $2^{\lfloor \log w \rfloor}$, or one tree of size $2^{\lceil \log w \rceil}$. In both cases, there are some straightforward edge cases for when to remove the leftmost segment.

4.3.4 Deamortized Updates

We now show how to deamortize the updates. Unfortunately the previous construction cannot be directly deamortized since the suffix tree construction algorithm by Farach-Colton et al. [FFM00] requires access to the whole string. Therefore, if a new character c causes a cascade of merges resulting in a new segment of size 2^i we have to build the suffix tree over that segment when c arrives.

Instead, we modify the structure slightly. When two segments of size 2^i become adjacent we temporarily keep both while deamortizing the cost of merging them over the *next* 2^i characters of S, doing expected constant work per character. Note that queries are unaffected, with one exception for reporting occurrences across the boundaries; there might now be two adjacent segments s_{i+1} and s_i of the same size that are both the smallest segment at least as large as $|P|$. In this case the suffix $\mathcal{R}$ extends only $m-1$ characters into the rightmost segment s_i. The boundary tree for (s_{i+1}, s_i) is large enough to report all occurrence crossing that boundary since both segments have size at least $|P|$. Furthermore, $\mathcal{R}$ potentially becomes twice as long, so we adjust the constants of the trees that we grow at query time.

To bound the time for updates we show that we are constructing at most $\log w$ suffix trees at any point, from which it follows that the update time is expected $O(\log w)$. To do so we show the following lemma.

Lemma 12. *When the construction of a segment of size 2^i finishes there is exactly one segment of each size $2^{i-1}, \ldots, 2^0$.*

Proof. The proof is by induction on i. For $i = 1$, when two size-one segments become adjacent we merge them when the next character c from S arrives. This results in a segment of size two, as well as a size-one segment containing c, proving the base case.

Inductively, consider the first time two segments of size 2^i become adjacent. By the induction hypothesis, there is one segment of each size $2^0, 2^1, \ldots, 2^{i-1}$ to the right of these two segments. For another segment of size 2^i to be constructed, we must first receive one more character, which triggers a merge that eventually cascades through all $i-1$ of these segments. For this to happen, $1+(2^0+2^1+\ldots+2^{i-1}) = 2^i$ more characters from S must arrive, where the 1 is for the next character to arrive, and 2^j is the amount of characters the jth merge is deamortized over. However, at this point the merge of the two segments of size 2^i is complete, so we constructed two new segments, one of size 2^{i+1} and one of size 2^i. By the induction hypothesis, there is also one segment of each size $2^0, \ldots, 2^{i-1}$, concluding the proof. □

Lemma 12 implies that there are never more than two segments of the same size adjacent to each other, and therefore at most one merging process for each segment size $2^0, 2^1, \ldots, 2^{\log w}$. To see this, consider the first time two segments a and b of size 2^i are adjacent. At this point, there are $2^0 + 2^1 + \ldots + 2^{i-1} = 2^i - 1$ characters to the right of b. When the next segment c of size 2^i arrives there are $2^i - 1$ characters to the right of that, too. But then there are $|c| + 2^i - 1 = 2^i + 2^i - 1$ characters to the right of b. Thus 2^i new characters must have arrived in the meanwhile, and the merging of a and b is done.

We obtain the following theorem.

Theorem 9. *Let S be a stream and let $w \geq 1$ be an integer. We can solve the w-SSWSI problem on S with an $O(w)$ space data structure that supports* Update *and* Report *in expected $O(\log w)$ time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

4.4 The Delayed SSWSI Problem

In this section we show how to improve the result from Section 4.3 if we are allowed a delay of δ. The main idea is as follows. As before, we maintain suffix trees of exponentially increasing sizes, although only the $O(\log(w/\delta))$ largest of them. As a result there are fewer trees to query, but also an *uncovered* suffix of size $\Theta(\delta)$ of the window for which we do not have any suffix trees. As in Section 4.3 we denote the part of S covered by suffix trees by s and we denote the uncovered suffix by t. As above, s is segmented into $s_1, \ldots, s_k$.

We will first explain how to solve the problem when all patterns are *long*, that is, $|P| > \delta/4$, and then when all patterns are *short*, that is, $|P| \leq \delta/4$. Finally we show how to combine these solutions. When all the patterns are long we can afford to construct, at query time, a suffix tree covering t. On the other hand, when all the patterns are short we can do both updates and queries in an offline fashion; we buffer queries and updates until we have approximately $\delta/2$ operations to do, at which point we can afford to construct a suffix tree over t in a deamortized manner. See Figure 4.4 for an example.

Throughout this section we assume without loss of generality that δ is a power of two. Otherwise we instead use a more restrictive delay of $\delta' = 2^{\lfloor \log \delta \rfloor}$ and achieve the same asymptotic bounds.

4.4.1 Long Patterns

We first show how to support queries if all patterns have a length $m > \delta/4$. We modify the data structure from Section 4.3 slightly. The smallest tree now has size $\delta/2$ as opposed to 1, so there are $\Theta(\log w - \log(\delta/2)) = O(\log(w/\delta))$ segments and boundary trees. The uncovered suffix t has length at most δ.

We answer queries the same way as in Section 4.3.2, with only small modifications. Let P be a pattern of length $m > \delta/4$. As before, let s_i be the smallest and rightmost segment with $|s_i| \geq m$. We find any occurrence within a segment or crossing a single boundary by using the suffix trees over each segment and the boundary trees to the left of s_i, as before. The remaining occurrences we again find by growing suffix trees of exponentially increasing sizes from the right window boundary. The only change is that we now grow the trees faster, as we must also cover t, and we can afford to let the smallest tree have size δ since we have $m > \delta/4$ characters in the pattern to deamortize the work over. As above, let $\mathcal{R}$ be the string covering the $m-1$ last characters of s_i and extending to the right window boundary, which now also includes t. As $|t| < \delta$ the length of $\mathcal{R}$ is $|\mathcal{R}| < 3m + \delta < 7m$. Assuming $2^\ell \leq m < 2^{\ell+1}$, we build the suffix tree of size $7 \cdot 2^{\ell+1}$ and match P in it, amortized over the characters of P. As we have $m > \delta/4$ characters to deamortize the work over, we only do this for each choice of ℓ where $2^{\ell+1} \geq \delta$, which results in $O(\log w - \log \delta) = O(\log(w/\delta))$ work per character in P. As in Section 4.3.2 we use recursive range maximum queries to avoid double reporting any occurrences of P that are also in s. As there are also only $O(\log(w/\delta))$ segments and boundary trees we spend $O(\log(w/\delta))$ time per character in P. Note that we answer these queries without delay.

Updates are performed as follows. For each segment of $\delta/2$ characters that arrives we construct the suffix tree over it, deamortized over the next $\delta/2$ characters of S. We merge suffix trees as before, also deamortized over new characters of S. The induction proof from Section 4.3.4 still works by modifying the base case; the merging of two trees of size $\delta/2$ takes $\delta/2$ characters, at which point another tree of size $\delta/2$ is constructed. The inductive step follows from the fact that δ is a power of two. Thus, we spend expected $O(\log(w/\delta))$ time per update.

4.4.2 Short Patterns

We now show how to support queries if all patterns have a length $m \leq \delta/4$. We extend the data structure with a buffer of size δ. This buffer will contain queries that we have not yet answered and characters for S that we have not yet processed. The total space is still $O(w + \delta) = O(w)$.

Whenever a character from S arrives we append it to both t and to the buffer. When a pattern arrives we append the full pattern to the buffer, and along with it we store the current position of the right window boundary. Once the buffer has more than $\delta/2$ characters (patterns and text combined) we immediately allocate a new buffer of size δ and *flush* the old buffer as follows. Note that at this point there are strictly less $\frac{3}{4}\delta$ characters in the buffer since each pattern is short.

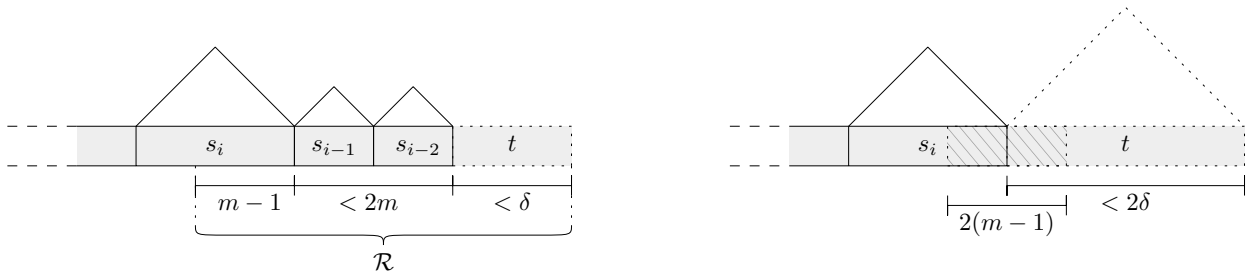

Figure 4.4: *Left:* Example of a query with a long pattern. Here s_i is the smallest and rightmost segment with $|s_i| \geq m$. Note that the non-indexed suffix t is less than $\delta < 4m$ characters long. *Right:* Example of a query with a short pattern. Note that for short patterns, s_i is always the rightmost segment. Any occurrence in s cross at most a single boundary and is found using the constructed trees. Any occurrence in t is found by the suffix tree over t that we construct when we flush the buffer. Any occurrence that cross the boundary (s, t) is found by the KMP automaton we build over the substring the extends $m-1$ characters in both directions from the boundary, which is hatched in the figure.

When we flush the buffer, we first answer all the buffered queries, and then we process all the buffered updates. We deamortize this work over the next $\delta/4$ characters that arrive from either stream. To answer the buffered queries we do as follows. Let $P_1, \ldots, P_\ell$ be the patterns in the buffer, let $m_i = |P_i|$, and let $M = \sum_{1 \leq i \leq \ell} m_i$. We have $M < \delta$. We start by building a suffix tree over t, along with a range maximum query data structure over the suffix array of t. This takes expected $O(\delta)$ time. An occurrence of P_i is either contained in s, or it crosses the boundary (s, t), or it is contained in t. Since P_i is smaller than each segment s_j we can find all the occurrences within s using the suffix trees over the segments and the boundary trees in $O(m_i \log(w/\delta))$ time. To find the occurrences crossing the boundary we build the KMP matching automaton [KJP77] for P_i. In it we match the string that is centered at the boundary (s, t) and extends $m_i - 1$ characters in each direction. This takes $O(m_i)$ time. To find the occurrences in t we match P_i in the suffix tree over t in $O(m_i)$ time. In total, this takes $O(M \log(w/\delta)) = O(\delta \log(w/\delta))$ time for all the patterns, or expected $O(\log(w/\delta))$ time per character when deamortized. Note however, that after P_i arrived more characters from S could have arrived and been appended to t. We must therefore take care not to report any occurrences of P_i that extend past what *was* the right window boundary when P_i arrived. The KMP automaton finds the occurrences in left-to-right order, and in t we avoid reporting too far right using recursive range minimum queries.

Finally, we process each update in the buffer in the order they arrived, using the same procedure as for long patterns. This takes $O(\log(w/\delta))$ time per update and $O(\delta \log(w/\delta))$ time in total. Thus flushing the buffer takes expected $O(\log(w/\delta))$ time per character since we deamortize the expected $O(\delta \log(w/\delta))$ work over $\delta/4$ characters. Since we allocate a new buffer immediately when we begin flushing, we will complete the flush before the next flush begins.

4.4.3 Both Long and Short Patterns

We now show how to combine the solutions for short and long patterns, to obtain a solution that handles patterns of any length. The data structure is the same as for small patterns above. As above, we append each new character to the buffer. However, whenever we start streaming a pattern we also proceed as if P were long. If P turns out to fit in the buffer without triggering a flush (which might also happen if P is long), we simply discard the work we did for the long-pattern case. However, if adding P to the buffer results in more than $\frac{3}{4}\delta$ characters being in the buffer, then P must be long. We immediately start flushing the buffer (ignoring the characters related to P) and also continue processing P as a long pattern. Note that since we are potentially streaming a long pattern while batch processing the updates in the buffer, the data structure might change while we are matching in it. However, it only changes when a merge finishes, replacing a pair of suffix trees by a larger tree. If this happens we keep the old trees in memory until we are done processing

the pattern, at which point we discard them.

We obtain the following theorem.

Theorem 10. *Let S be a stream and let $w \geq 1$ and $\delta \geq 1$ be integers. We can solve the (w, δ)-SSWSI problem on S with an $O(w)$ space data structure that supports* Update *and* Report *in expected $O(\log(w/\delta))$ time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

4.5 Obtaining High Probability

In this section we show how to improve the time bounds to $O(\log(w/\delta))$ with probability $1 - w^{-d}$ for any constant $d \geq 1$.

The expectation in the time bounds in Section 4.4 comes from the construction of suffix trees (recall that we also build suffix trees at query time). Below, in Lemma 13, we prove that given a string $\mathcal{K}$ of length $k = O(w)$ we can construct the suffix tree over $\mathcal{K}$ in $O(k)$ time with probability $1 - 1/w^{1+\epsilon}$, using additional $O(w/\log w)$ space. We use this algorithm to construct suffix trees during updates and queries, deamortizing them as before and doing $O(\log(w/\delta))$ work per character that arrives. When a new character arrives from S or P, at most $O(\log(w/\delta)) = O(\log w)$ suffix tree constructions will finish. At this point, we finish constructing those trees that did not finish in time, that is, used more more time than what was allotted to them. By the union bound, the probability that any of them fail to finish in time (and thus incurring extra construction cost) is no more than $c \log w / w^{1+\epsilon}$ for some constant c which is no more than $1/w$ for large w. Thus, for each character from S or P we spend $O(\log(w/\delta))$ time with high probability in w. We obtain the $1 - 1/w^d$ probability bound by probability boosting, running $d = O(1)$ independent copies of the construction algorithm simultaneously. The algorithm from Lemma 13 uses additional $O(w/\log w)$ space, but we are never constructing more than $O(\log w)$ suffix trees, so the space usage is $O(w)$ in total.

Furthermore, as mentioned in Section 4.2, we previously used an FKS dictionary [FKS84] to store the edges to support reporting queries in worst-case constant time per character in the pattern. The construction time of this dictionary is expected linear, so it can no longer be used. Instead we use a dictionary by Dietzfelbinger and Meyer auf der Heide [DadH90]. If there are n elements in the dictionary it supports searches in worst-case constant time and any sequence of $\frac{1}{2}n$ updates takes constant time per update with probability $1 - 1/n^{d'}$ for any constant $d' \geq 1$. We store all the edges of all the suffix trees in one such dictionary. At all times, we keep $\Theta(w)$ dummy-elements in the dictionary to ensure that we get good probability bounds in terms of w, and we choose d' large enough that any sequence of $O(w)$ operations (e.g., the construction of any one of our suffix trees) runs in $O(w)$ time with probability $1 - 1/w^{d+\epsilon}$.

Universal Hashing Before we prove Lemma 13 we restate some basic facts about universal hashing, introduced by Carter and Wegman [CW79]. Let $M, m > 0$ be integers, $\mathcal{H}$ be a set of functions $[0, M] \to [0, m]$, and $h \in \mathcal{H}$ be selected uniformly at random. Then $\mathcal{H}$ is *universal* if $P[h(x) = h(y) \mid x \neq y] \leq 1/m$. Let $R \subseteq [0, M]$ and $|R| = r$. It follows from the union bound that h has a *collision* on R with probability at most

$$P[h(x) = h(y) \text{ for some } x \neq y] \leq \sum_{x \neq y \in R} P[h(x) = h(y)] = \frac{r(r-1)}{2} \cdot \frac{1}{m} < \frac{r^2}{m}. \tag{4.1}$$

In particular, if $m = r^c$ for constant $c \geq 1$ then h is *injective* (i.e., has no collisions) on R with probability at least $1 - 1/r^{c-2}$. Carter and Wegman gave several classes of universal hash functions from which we can sample a function uniformly at random in constant time.

Fast Suffix Tree Construction We now prove Lemma 13, showing how to construct our suffix trees in linear time with high probability.

Lemma 13. *Given a string $\mathcal{K}$ of length $k \leq 2w$ there is an algorithm that uses $O(k + w/\log w)$ space and constructs the suffix tree over $\mathcal{K}$ in $O(k)$ time with probability $1 - 1/w^{1+\epsilon}$ for some $\epsilon > 0$.*

Proof. Let $\sigma = \{\mathcal{K}[i] \mid i \in [1,k]\} \subseteq \Sigma$ be the alphabet of $\mathcal{K}$. We show how to, in $O(k)$ time, find a function $h : \Sigma \to [1, k^{O(1)}]$ such that h is injective on σ with probability at least $1 - 1/w^{1+\epsilon}$. If h is injective on σ, we can construct the suffix tree over $\mathcal{K}'$ where $\mathcal{K}'[i] = h(\mathcal{K}[i])$ in time $O(\text{sort}(k, k^{O(1)})) = O(k)$ using radix sort. After the tree is constructed we can substitute for the original alphabet in linear time. Therefore, the construction algorithm finishes in $O(k)$ time with probability at least $1 - 1/w^{1+\epsilon}$ (otherwise we make no guarantee on the construction time and we can build the suffix tree in any way).

For some m to be determined later, let $f : \Sigma \to [1, m]$ be chosen uniformly at random from a class of universal hash functions. By Equation 4.1, the probability that f has a collision on σ is

$$P[f \text{ has collisions on } \sigma] < \frac{|\sigma|^2}{m} \leq \frac{k^2}{m}.$$

We divide into the cases of large trees ($k \geq w^{1/5}$) and small trees ($k < w^{1/5}$). If k is large then $w^{1/5} \leq k \leq 2w$, and we set $m = w^4$ so the probability that f has a collision is at most

$$\frac{k^2}{m} \leq \frac{(2w)^2}{w^4} = \frac{4}{w^2} \leq \frac{1}{w^{1+\epsilon}}$$

for some $\epsilon > 0$. We check whether f is injective by sorting the set $\{(x, f(x)) \mid x \in \sigma\}$ with respect to the $f(\cdot)$-values and checking if two consecutive elements $(x, f(x))$ and $(y, f(y))$ have $x \neq y$ and $f(x) = f(y)$. This takes time $O(\text{sort}(k, w^4)) = O(k)$ using radix sort since $k \geq w^{1/5}$. If f is injective we set $h = f$, concluding the proof of the large case.

If k is small then we allocate an array A of length $w/\log w$ in constant time. For simplicity we assume that A is initialized such that $A[i] = 0$ for all i. This can be avoided using standard constant-time initialization schemes; assume each entry in A contains an arbitrary value initially. We maintain two other arrays B and C such that if we have written a value to $A[i]$ at least once then $A[i]$ is a pointer to some $B[j]$, $B[j]$ is a pointer to $A[i]$, and $C[j]$ stores the value most recently written to $A[i]$. From this we can determine if $A[i]$ has been initialized (check if the pointers match), and if it has not we can initialize it in constant time.

Then we set $m = w/\log w$ such that the probability that f has a collision is no more than

$$\frac{k^2}{m} < \frac{w^{2/5}}{w/\log w} = \frac{\log w}{w^{3/5}} = \frac{\log w}{w^{1/2}} \cdot \frac{1}{w^{1/10}} \leq \frac{1}{w^{1/10}}$$

for $w \geq 16$. We check if f is injective on σ by for each character x in $\mathcal{K}$ setting $A[f(x)] = x$ and seeing if two distinct characters hash to the same index. If f is injective we then arbitrarily assign the values $1, \ldots, |\sigma|$ to the now non-zero indices of A and let $h(x) = A[f(x)]$ (at this point we know σ since it is equal to the number of entries in A that we modified). To boost the probability of success we run this algorithm up to eleven times with independent choices for f. The probability that all of them fail is at most $1/w^{11/10} \leq 1/w^{1+\epsilon}$ concluding the proof for the small case. □

In conjunction with Theorems 9 and 10, this proves Theorem 8.

4.6 Conclusion and Future Work

We have studied two variants of the streaming sliding window string indexing problem; the timely variant, where queries must be answered immediately, and the delayed variant where a query may be answered at any point within the next δ characters received, for a specified parameter δ. For a sliding window of size w we have given an $O(w)$ space data structure that, in the timely variant, supports updates in $O(\log w)$ time with high probability and queries in $O(\log w)$ time with high probability per character in the pattern; each occurrence is reported in additional constant time. For the delayed variant we improved these bounds to $O(\log(w/\delta))$, where each occurrence is still reported in constant time.

One open problem is whether these bounds can be improved. Another is to find efficient solutions when queries may be interleaved with new updates to the stream. That is, while you are streaming a pattern, new characters of S might arrive that move the current window.

Chapter 5

New Advances in Rightmost Lempel-Ziv

New Advances in Rightmost Lempel-Ziv

Philip Bille
Technical University of Denmark
phbi@dtu.dk

Inge Li Gørtz
Technical University of Denmark
inge@dtu.dk

Max Pedersen
Technical University of Denmark
mhrpe@dtu.dk

Tord Joakim Stordalen
Technical University of Denmark
tjost@dtu.dk

Abstract

The Lempel-Ziv (LZ) 77 factorization of a string is a widely-used algorithmic tool that plays a central role in compression and indexing. For a length-n string over a linearly-sortable alphabet, e.g., $\Sigma = \{1, \ldots, \sigma\}$ with $\sigma = n^{O(1)}$, it can be computed in $O(n)$ time. It is unknown whether this time can be achieved for the *rightmost* LZ parsing, where each referencing phrase points to its rightmost previous occurrence. The currently best solution takes $O(n(1 + \log \sigma/\sqrt{\log n}))$ time (Belazzougui & Puglisi SODA2016). We show that this problem is much easier to solve for the LZ-End factorization (Kreft & Navarro DCC2010), where the rightmost factorization can be obtained in $O(n)$ time for the greedy parsing (with phrases of maximal length), and in $O(n + z\sqrt{\log z})$ time for any LZ-End parsing of z phrases. We also make advances towards a linear time solution for the general case. We show how to solve multiple non-trivial subsets of the phrases of any LZ-like parsing in $O(n)$ time. As a prime example, we can find the rightmost occurrence of all phrases of length $\Omega(\log^{6.66} n/\log^2 \sigma)$ in $O(n/\log_\sigma n)$ time and space.

5.1 Introduction

The Lempel-Ziv (LZ) 77 factorization [LZ76] of a string S decomposes it into a series of phrases $S = f_1 f_2 \ldots f_z$. Each phrase is either the leftmost occurrence of an alphabet symbol (a *literal phrase*), or the longest substring that can be read at an earlier position in the string (a *referencing phrase*). Compression can be achieved by replacing each referencing phrase with an integer pair consisting of the length and the distance to an earlier occurrence of the phrase. Further compression is possible by encoding the integers, e.g., by applying a universal code. Variable length codes often assign longer codewords to larger integers, and thus it is beneficial if every referencing phrase knows not only any of its previous occurrences, but the rightmost one (at the smallest distance).

In an LZ-*like* factorization, referencing phrases do not need to be of maximal length. The encoding works in the same way as for the exact LZ factorization.

Related Work. LZ(-like) parsings are well-studied, and there are fast factorization algorithms in multiple settings (we only list a few examples for each) including parallel [CR91, FM95, Nao91, SZ13, Shu18], online [OS08, Sta12, YIB+14] and external memory algorithms [KKP14]. In the sequential setting, there are several linear-time solutions [GB13, KKP13b, GB14, FIK15], and some that compute the parsing in small space [OG11, OS08, Sta12, YIB+14, KKP13a, Kos15, KS16, BP16], with the overall best using only $O(n \log \sigma)$ bits and running in $O(n)$ time [FIKS18] for a string of length n over integer alphabet $[0, \sigma)$.

LZ-End, introduced by Kreft and Navarro [KN10, KN13], is a family of LZ-like parsings where each referencing phrase must have a previous occurrence aligned with the end of a phrase, i.e., for f_k there must be $k' < k$ such that f_k is a suffix of $f_1 f_2 \dots f_{k'}$. This has beneficial properties that lead to efficient compressed text indices (e.g., [KS22]). The uniquely defined *greedy* LZ-End parsing, in which each referencing phrase is of maximal length, can be computed in linear time [KK17b], and the number of phrases is within an $O(\log^2 n)$ factor of the exact LZ factorization [KS22]. Bannai et al. [BFK+23] proved that computing the optimal LZ-End parsing (with minimal number of phrases) is NP-hard and gave a lower bound of 2 for the approximation ratio of optimal LZ-End to greedy LZ-End.

The first theoretical result on computing the rightmost LZ parsing is by Amir et al. [ALU02] and uses $O(n \log n)$ time and working space. Larsson et al. [Lar14] presented an online algorithm in the same time and space. The first approximation algorithm was by Crochemore et al. [CLM13] and runs in $O(n \log n)$ time and $O(n)$ space, and finds the rightmost *equal-cost* position for each phrase, meaning it takes the same number of bits to encode as the rightmost position. Later, Bille et al. [BCFG17] gave an $(1+\epsilon)$-approximation algorithm of the rightmost parsing in $O(n(\log z + \log\log n))$ time and linear working space. The first exact algorithm to achieve $o(n \log n)$ time is by Ferragina et al. [FNV13] and runs in $O(n(1 + \log\sigma/\log\log n))$ time and $O(n)$ words of space. This was improved by Belazzougui and Puglisi [BP16] with an algorithm using only $O(n \log\sigma)$ bits of space and achieving $O(n(\log\log\sigma + \log\sigma/\sqrt{\log n}))$ deterministic time or $O(n(1 + \log\sigma/\sqrt{\log n}))$ time with randomization, which is the current state of the art.

Our Contributions. We present time-efficient deterministic algorithms for rightmost LZ parsings, summarized by Theorems 11 and 12 below.

Theorem 11. *Let $S \in [0,\sigma)^n$. Given an LZ-End factorization $S = f_1 \dots f_z$, we can compute its rightmost LZ-End parsing in $O(n + z\sqrt{\log z})$ time and $O(n)$ words of space. For the greedy LZ-End factorization, we achieve $O(n)$ time.*

Theorem 12. *Let $S \in [0,\sigma)^n$. Unless explicitly stated otherwise, the space complexity is $O(n)$ words. Given any LZ-like factorization $S = f_1 \dots f_z$, we can compute the rightmost previous occurrence of all referencing phrases*

- *(a) of length $\Omega(\log^{6.66} n/\log^2\sigma)$ in $O(n/\log_\sigma n)$ time and words of space*
- *(b) f_k with $k \in F \subseteq [1,z]$ in $O(n + |F|\, d^\epsilon)$ time, where $d = |\{f_{k'} \mid k' \in F\}| \le |F|$*
- *(c) f_k with $|\{k' \in [1,z] \mid f_{k'} = f_k\}| = O(\log n)$ in $O(n)$ time*
- *(d) with rightmost previous occurrence at distance $O(\log n)$ in $O(n)$ time*

We provide the solution for rightmost parsings of LZ-End factorizations (Theorem 11) in Section 5.3. The algorithms for subsolutions of general rightmost LZ-like parsings (Theorem 12) are presented in Section 5.4.

5.2 Preliminaries

Strings and Model of Computation. For $i, j \in \mathbb{N}$, we write $[i,j] = [i, j+1)$ rather than $\{k \in \mathbb{N}^+ \mid i \le k \le j\}$. A string $S = S[1..n] = S[1]S[2]\dots S[n]$ of length $|S| = n$ is a sequence of n symbols from an alphabet Σ. For $i, j \in [1,n]$, the substring $S[i..j] = S[i..j+1)$ is the sequence $S[i]S[i+1]\dots S[j]$ (or the empty string ε if $j < i$). A substring shorter than S is *proper*. Substrings $S[1..i]$ and $S[i..n]$ are respectively called prefix and suffix of S. The reversal of S is $\mathsf{rev}(S) = S[n]S[n-1]\dots S[1]$. The concatenation of two string S_1 and S_2 is S_1S_2. We only consider alphabets Σ that are totally ordered, which induces a lexicographical order over the set of all strings in the usual way. We write $S_1 \prec S_2$ to denote that S_1 is lexicographically smaller than S_2. We say that S_1 is *co*-lexicographically smaller than S_2 if $\mathsf{rev}(S_1) \prec \mathsf{rev}(S_2)$. For strings S and P, an *occurrence* of P in S is a position i such that P is a prefix of $S[i..|S|]$. For the occurrence i of substring $S[i..i+\ell)$ in S, a *previous occurrence* is an occurrence j of $S[i..i+\ell)$ in S with $j < i$. We assume that the string $S[1..n]$ is over integer alphabet $[0,\sigma)$ with $\sigma = n^{O(1)}$, and we use a word RAM of width $w = \Theta(\log n)$

bits (see, e.g., [Hag98]). Each symbol is stored in $\lceil \log \sigma \rceil$ bits, and thus the string occupies $O(n/\log_\sigma n)$ words of space. From now on, space complexities are given in number of words.

We assume that the reader is familiar with tries [Fre60]. The *suffix tree* [Wei73] of S is the compact trie of all suffixes of $S\$$, where $\$ = -\infty$ is smaller than all symbols from the alphabet. Each leaf corresponds to a suffix of S and is labeled with the start position of this suffix. The outgoing edges of each node are arranged in increasing order of the first symbol of the respective edge label. Hence the leaves are ordered from left to right in lexicographical order of suffixes. In the present model of computation, the suffix tree can be computed in $O(n)$ time and space [MM93a]. The suffix array SA of S is the unique permutation of $[1, n]$ that lexicographically sorts the suffixes, i.e., $\forall i \in [1, n) : S[\mathsf{SA}[i]..n] \prec S[\mathsf{SA}[i+1]..n]$. Equivalently, it consists of the leaf-labels of the suffix tree in left-to-right order and can therefore be constructed from the suffix tree in linear time.

Lempel-Ziv Parsings. The unique *LZ (77) factorization* $S = f_1 f_2 \dots f_z$ decomposes S into z substrings called *phrases*. Each phrase f_k at *destination* $i = 1 + \sum_{j=1}^{k-1} |f_j|$ is either the leftmost occurrence of $S[i]$ (a *literal phrase*), or the longest prefix of $S[i..n]$ that has a previous occurrence (a *referencing phrase*). A previous occurrence $j \in [1, i)$ of a referencing phrase f_k is called a *source* of f_k. (This is Storer and Szymanski's version of the factorization [SS82].) An *LZ-like factorization* is defined exactly like the LZ factorization, but without the requirement that referencing phrases are of maximal length. The *rightmost parsing* of an LZ(-like) factorization annotates each referencing phrase with its rightmost source, i.e., f_k at destination i is annotated with the maximal $j \in [1, i)$ such that f_k is a prefix of $S[j..n]$.

A source j of some phrase f_k in an LZ-like factorization is *LZ-End aligned* if $S[1..j + |f_k|) = f_1 f_2 \dots f_{k'}$ for some $k' \in [1, k)$ (i.e., f_k equals the suffix of $f_1 f_2 \dots f_{k'}$ that starts at position j). An *LZ-End factorization* is an LZ-like factorization in which all referencing phrases have an LZ-End aligned source. (This is slightly different from [KN10] and leads to a simpler description; the presented results can be easily modified to work for the original definition.) The *greedy LZ-End factorization* is the unique LZ-End factorization in which each f_k at destination i is the longest prefix of $S[i..n]$ that is a suffix of $f_1 f_2 \dots f_{k'}$ for some $k' \in [1, k)$. We could define the rightmost parsing for LZ-End in the same way as for arbitrary LZ-like factorizations (i.e., annotate each phrase with its rightmost source), but this is undesirable because the rightmost source might not be LZ-End aligned. Hence the rightmost parsing of an LZ-End factorization annotates each referencing phrase with its rightmost LZ-End aligned source.

From now on, we use z (commonly used to denote the number of phrases in the exact LZ 77 factorization) to denote the number of phrases in the factorization at hand, even if it is an LZ-like or LZ-End factorization. Instead of saying that we compute the rightmost source of f_k, we simply say that we resolve f_k.

5.3 Computing Rightmost LZ-End Parsings

In this section, we provide the solutions for Theorem 11. We exploit the fact that an LZ-End phrase only has to choose from less than z sources, while a general LZ-like phrase has to consider up to $\Omega(n)$ possible sources. This makes the computation significantly easier for LZ-End factorizations.

5.3.0.1 Rightmost Greedy LZ-End Parsing.

We start by computing an arbitrary LZ-End aligned source for each referencing phrase f_k. We build the suffix array of the reversed text $\mathsf{rev}(S)$, and use filtering and rank reduction to obtain in $O(n)$ time the unique permutation co of $[1, z]$ that satisfies $\forall k' \in [1, z) : \mathsf{rev}(f_1 f_2 \dots f_{\mathsf{co}(k')}) \prec \mathsf{rev}(f_1 f_2 \dots f_{\mathsf{co}(k'+1)})$. (This permutation rearranges the prefixes that end at phrase boundaries in co-lexicographical order.) We also compute its inverse permutation co^{-1}. Any referencing phrase f_k has a previous occurrence as a suffix of $f_1 f_2 \dots f_{k'}$, where k' and k are neighbors in co (because the co-lexicographical order groups together prefixes that share a long suffix). More precisely, if $\mathsf{co}^{-1}(k) = 1$ then $k' = \mathsf{co}(2)$. If $\mathsf{co}^{-1}(k) = z$ then $k' = \mathsf{co}(z-1)$. Otherwise, $k' \in \{k^-, k^+\}$ with $k^- = \mathsf{co}(\mathsf{co}^{-1}(k)-1)$ and $k^+ = \mathsf{co}(\mathsf{co}^{-1}(k)+1)$. In the latter case, we naively check if f_k is a suffix of $f_1 f_2 \dots f_{k^-}$. If this is the case, then we use $k' = k^-$. Otherwise, we use $k' = k^+$.

Hence we can compute a suitable k' for each referencing phrase f_k in total time $O(n+z+\sum_{j=1}^{z}|f_j|) = O(n)$. We then report $|f_1 f_2 \dots f_{k'}| - |f_k| + 1$ as an LZ-End aligned source of f_k.

The computed sources are already rightmost for all phrases that only have a single LZ-End aligned source. It remains to correct the sources of phrases that have multiple LZ-End aligned sources, for which we observe the following.

Proposition 1. *Let f_k be a referencing phrase in the greedy LZ-End factorization, and let $k', k'' \in [1, k)$ with $k'' < k'$ be such that f_k is a suffix of both $f_1 f_2 \dots f_{k'}$ and $f_1 f_2 \dots f_{k''}$. Then f_k is a suffix of $f_{k'-1} f_{k'}$.*

Proof. If f_k is a suffix of $f_1 f_2 \dots f_{k'}$ but not of $f_{k'-1} f_{k'}$, then $f_{k'-1} f_{k'}$ is a suffix of f_k. Since f_k is a suffix of $f_1 f_2 \dots f_{k''}$, this implies that $f_{k'-1} f_{k'}$ is a suffix of $f_1 f_2 \dots f_{k''}$. Hence $f_{k'-1} f_{k'}$ has a previous occurrence that satisfies the LZ-End property. Thus, $f_{k'-1}$ is not of maximal length, which contradicts the definition of the greedy LZ-End factorization. □

We compute a compacted trie that contains for each $k' \in [2, z]$ the string $\mathsf{rev}(f_{k'-1} f_{k'})$. Note that the total length of the strings is less than $2n$. We make the respective nodes that spell $\mathsf{rev}(f_{k'})$ and $\mathsf{rev}(f_{k'-1} f_{k'})$ explicit (if they are not explicit already), and store pointers to these nodes. We will not need fast navigation on the trie; in fact, we only need the parent operation. Hence we can construct the trie in $O(n)$ deterministic time using standard techniques (e.g., from the suffix array of $\mathsf{rev}(f_1 f_2 \# f_2 f_3 \# \dots \# f_{z-1} f_z)$ where $\#$ is a special separator symbol). Now we process the phrase pairs $f_{k'-1} f_{k'}$ with $k' \in [2, z]$ from right to left. Whenever we finish processing a pair, we annotate the node that spells $\mathsf{rev}(f_{k'})$ with k' (indicating that the rightmost LZ-End aligned source of $f_{k'}$ has not been found yet). Before adding this annotation, we first check if $f_{k'-1} f_{k'}$ resolves other phrases. For this purpose, we traverse the path from the leaf that spells $\mathsf{rev}(f_{k'-1} f_{k'})$ to the root of the trie. For each node on the path, we check if it has been annotated with some value k. If we find such an annotation, then the corresponding node spells $\mathsf{rev}(f_k)$, and f_k is a suffix of $f_{k'-1} f_{k'}$. Hence we store $|f_1 f_2 \dots f_{k'}| - |f_k| + 1$ as the maximal LZ-End aligned source of f_k, and remove the annotation of the node. By Proposition 1 and the right-to-left order of processing, we correctly find the rightmost LZ-End aligned source of any phrase that has multiple LZ-End aligned sources.

A node might spell the reversal of a phrase that has multiple occurrences in the parsing. Nevertheless, each node has at most one annotation at any given point in time. This is because we annotate the node that spells $\mathsf{rev}(f_{k'})$ only after we finish processing pair $f_{k'-1} f_{k'}$. If the node is already annotated with some $k > k'$ (because $f_k = f_{k'}$), then we also find the source $|f_1 f_2 \dots f_{k'}| - |f_k| + 1$ of f_k while processing pair $f_{k'-1} f_{k'}$, and hence we remove annotation k before adding annotation k'.

We need $O(n)$ time for computing the trie. Processing a pair $f_{k'-1} f_{k'}$ takes time linear in the depth of the node that spells $\mathsf{rev}(f_{k'-1} f_{k'})$. This is limited by $O(|f_{k'-1} f_{k'}|)$, which sums to $O(n)$ over all phrase pairs. The space for the trie is $O(n)$. Hence we have shown Theorem 11 for the greedy LZ-End factorization.

5.3.0.2 Rightmost (Arbitrary) LZ-End Parsing.

If the given LZ-End factorization does not satisfy the greedy property, then Proposition 1 no longer holds. However, each referencing phrase f_k is still a suffix of some $f_1 f_2 \dots f_{k'}$ with $k' \in [1, k)$, which limits the number of possible sources. We will again exploit properties of the co-lexicographical order of prefixes.

We compute a compacted trie that contains for each $k' \in [1, z]$ the reversed prefix $\mathsf{rev}(f_1 f_2 \dots f_{k'})$ of the text. We make the respective nodes that spell $\mathsf{rev}(f_{k'})$ and $\mathsf{rev}(f_1 f_2 \dots f_{k'})$ explicit (if they are not explicit already), and store pointers to these nodes. We annotate the node that spells $\mathsf{rev}(f_1 f_2 \dots f_{k'})$ with its co-lexicographical rank $\mathsf{co}^{-1}(k')$ (defined as before). Additionally, we annotate the node that spells $\mathsf{rev}(f_{k'})$ with its co-lexicographical range, which is given by the respectively smallest and largest co-lexicographical ranks $c_{k'}^{\min}$ and $c_{k'}^{\max}$ that were used to annotate any of its descendants (or itself). Again, we do not need fast navigation on the trie; for writing the annotations, it suffices if we can perform a preorder traversal in linear time. Hence we can construct the trie and its annotations in $O(n)$ deterministic time using standard techniques (e.g., from the suffix array of $\mathsf{rev}(S)$).

Now we show how to find the rightmost LZ-End aligned source of referencing phrase f_k. We have annotated the node that spells $\mathsf{rev}(f_k)$ with the co-lexicographical range $[c_k^{\min}, c_k^{\max}]$. We store the permutation

co (defined as before) in an array. Note that, by design of the trie, the range $\mathsf{co}[c_k^{\min}, c_k^{\max}]$ contains exactly all the k' for which f_k is a suffix of $f_1 f_2 \dots f_{k'}$. Hence finding the rightmost LZ-End aligned source of f_k is equivalent to answering the following so-called *range predecessor query*. Given the range $[c_k^{\min}, c_k^{\max}] \subseteq [1, z]$ and the threshold k, find the largest value $k' < k$ in $\mathsf{co}[c^{\min}, c^{\max}]$. Then, the rightmost LZ-End aligned source of f_k is $|f_1 f_2 \dots f_{k'}| - |f_k| + 1$.

Belazzougui and Puglisi show how to compute a data structure in $O(z\sqrt{\log z})$ time and $O(z)$ space that answers range predecessor queries on a permutation of $[1, z]$ in $O(\log^\epsilon z)$ time (for any constant $0 < \epsilon < 1$). We issue less than z queries, and thus the total construction and query time is $O(z\sqrt{\log z})$. The total time for computing the rightmost parsing (including the construction of the trie) is $O(n + z\sqrt{\log z})$, and the total space is $O(n)$. Hence we have shown Theorem 11 for an arbitrary LZ-End factorization.

5.4 Partially Solving Rightmost LZ-Like Parsings

In this section, we show how to efficiently compute the rightmost sources for some subsets of the phrases of an LZ-like factorization (Theorem 12).

5.4.1 Long Phrases

Belazzougui and Puglisi [BP16] find the rightmost sources of all phrases of length $\Omega(\log^5 n)$ in $O(n)$ time and $O(n/\log_\sigma n)$ space. We show a similar result for resolving all phrases of length $\Omega(\log^{33/5+\epsilon} n/\log^2 \sigma)$ in $O(n/\log_\sigma n)$ time and space. The main contribution here is that we achieve sublinear time. The solution works for an arbitrary LZ-like factorization $S = f_1 f_2 \dots f_z$.

Let $\delta = \Omega(\log^2 n/\log \sigma)$ be a parameter to be fixed later. We start by performing a preprocessing as follows. In $O(n/\log_\sigma n)$ time, we compute the reversed text $\mathsf{rev}(S)$ as described in [BP16, Section 6.2] (essentially, we use a precomputed lookup table to reverse the text one half-word rather than one symbol at a time). We consider a set $\mathcal{D} = \{d \in [1, n] \mid d \equiv 0 \pmod{\delta}\}$ of $m = |D| = O(\frac{n}{\delta})$ regularly sampled positions. We construct the respectively unique permutations pref and suf of $[1, m]$ such that for every $h \in [1, m)$ it holds $S[\mathsf{suf}(h)\delta..n] \prec S[\mathsf{suf}(h+1)\delta..n]$ and $\mathsf{rev}(S[1..\mathsf{pref}(h)\delta]) \prec \mathsf{rev}(S[1..\mathsf{pref}(h+1)\delta])$ (these are sparse suffix arrays of the string and its reversal). We use comparison sorting and obtain the permutations with $O(m \log m) \subseteq O(\frac{n}{\delta} \log n) \subset O(n/\log_\sigma n)$ lexicographical comparisons between suffixes of either S or $\mathsf{rev}(S)$. With an LCE data structure by Kempa and Kociumaka [KK19] (constructed for both S and $\mathsf{rev}(S)$), each lexicographical comparison takes constant time. The data structure can be constructed in $O(n/\log_\sigma n)$ time and space. We use $O(m \log m) \subseteq O(\frac{n}{\delta} \log n) \subset O(n \log \sigma)$ bits of space to store pref, suf, and their respective inverse permutations $\mathsf{pref\text{-}rank}$ and $\mathsf{suf\text{-}rank}$.

A long phrase is of length at least $\gamma > \delta$, where γ is another parameter. When resolving a long phrase f_k with rightmost source j and destination i, we will use the fact that $j + q$ with $q = (\delta - (j \bmod \delta)) \in [1, \delta]$ is a sample position. For now, assume that we know the value of q in advance (we will later simply try all the possible values of q). Finding the rightmost source of f_k means that we have to find the rightmost sample position $h\delta < i + q$ with $S[h\delta - q..h\delta] = S[i..i + q]$ and $S[h\delta..h\delta - q + |f_k|) = S[i + q..i + |f_k|)$. Note that the co-lexicographical order groups together prefixes that share a long suffix, and hence all the values of h for which $S[i..i + q]$ is a suffix of $S[1....h\delta]$ form a consecutive interval $\mathsf{pref}[p_1..p_2]$ (we treat the permutations like arrays). We can find the boundaries p_1 and p_2 by binary searching in pref for the respectively co-lexicographically minimal and maximal prefixes of S that have suffix $S[i..i + q]$. This takes $O(\log m)$ time because we can perform each LCE computation and lexicographical comparison in constant time using the same LCE data structure as before. Similarly, it takes $O(\log m)$ time to compute the interval $\mathsf{suf}[s_1..s_2]$ that contains exactly the values of h for which $S[i + q..i + |f_k|)$ is a prefix of $S[h\delta..n]$.

We associate a three-dimensional point $(\mathsf{pref\text{-}rank}(h), \mathsf{suf\text{-}rank}(h), h)$ with each sample position. For resolving the phrase, we have to find the point $(p, s, \hat{h})$ with $p \in [p_1, p_2]$, $s \in [s_1, s_2]$, and maximal value $\hat{h}\delta < i + q$ (or equivalently $h < \frac{i+q}{\delta}$). Given this point, it is easy to compute the rightmost source $\hat{h}\delta - q$ of f_k. For solving the geometric query, we use a data structure for three-dimensional orthogonal range searching [CT18, Theorem 4]. For our m points from $[1, m]^3$, it can be constructed in $O(m \log^{8/5+\epsilon} m)$ time

and space (for any constant $\epsilon \in \mathbb{R}^+$). Given a three-dimensional six-sided orthogonal query range, it returns a point in the range or reports that it is empty in $O(\log^2 m)$ time (the precise bound is slightly better, but not needed for our purposes). For our queries, we have to find the point with maximal coordinate in the third dimension. Thus, we binary search for this point with $O(\log n)$ queries to the geometric data structure, which increases the query time to $O(\log^3 n)$. Note that this dominates the $O(\log m)$ time needed to compute the query range. Finally, we do not actually know the value of q in advance. Hence we try all the possible $q \in [1, \delta]$. For each of them, we compute the query range and find the rightmost admitted source in $O(\log^3 n)$ time. Thus, the time needed per phrase is $O(\delta \cdot \log^3 n)$.

We need $O(n/\log_\sigma n)$ time for computing the (co-)lexicographically sorted permutations of samples, $O(\frac{n}{\delta} \log^{8/5+\epsilon} n)$ time for computing the geometric data structure, and $O(\frac{n\delta}{\gamma} \cdot \log^3 n)$ time for actually resolving the phrases. We want δ to be small in order to minimize the time for resolving phrases. On the other hand, the time needed for computing the geometric data structure should become $O(n/\log_\sigma n)$. Hence we use $\delta = \Theta(\log^{13/5+\epsilon} n/\log\sigma)$, which achieves the desired construction time and implies that we take $O(\frac{n}{\gamma} \cdot \log^{28/5+\epsilon} n/\log\sigma)$ time for resolving phrases. Thus, in order to achieve $O(n/\log_\sigma n)$ time, long phrases have to be of length at least $\gamma = \Omega(\log^{33/5+\epsilon} n/\log^2\sigma) \subset \Omega(\log^{6.66} n/\log^2\sigma)$. For all steps (including the geometric data structure), the space is linear in the time spent, and hence it is $O(n/\log_\sigma n)$. This concludes the proof of Theorem 12(a).

5.4.2 Arbitrary Subsets of Phrases

Now we show how to solve an arbitrary subset of phrases of any LZ-like factorization $S = f_1 f_2 \dots f_z$. The subset is given by $F \subseteq [1, z]$, and the time complexity depends on $d = |\{f_k \mid k \in F\}| \leq F$, i.e., on the number of distinct phrases in the subset. In a slight abuse of terminology, we will say that f_k is a phrase from F if $k \in F$. We show how to resolve all phrases from F in $O(\frac{n}{\epsilon} + |F|\, d^\epsilon)$ time and $O(\frac{n}{\epsilon})$ space for arbitrary $\epsilon \in \mathbb{R}^+$ with $\epsilon \leq \frac{1}{2}$, or $O(n + |F|\, d^\epsilon)$ time and $O(n)$ space for constant ϵ. If the string is highly compressible, say, $z = O(n^{1-\epsilon})$, then the time is $O(n)$. The idea is to use range maximum data structures to find the rightmost sources. We note that this solution is very similar to [FNV13], and mostly differs in the choice of the range maximum data structure.

We start with the following preprocessing. We arrange the distinct phrases of F into a tree of d nodes, and we start using the terms node and phrase interchangeably (even though multiple phrases may refer to the same node). The parent of phrase f_k is the longest phrase $f_{k'}$ from F that is a proper prefix of f_k (or the artificial root node ϵ if $f_{k'}$ does not exist), and we call this tree the *phrase trie*. This is a slight abuse of terminology, since the tree is only similar to a trie. An example is provided in Figure 5.1. We annotate f_k with its *preorder number* p_k, which is the rank of f_k in a preorder traversal of the phrase trie, as well as the maximal preorder number q_k of a descendant of f_k. We also annotate each text position i with the preorder number of the longest phrase from F that is a prefix of $S[i..n]$, if any. This concludes the preprocessing.

In order to resolve the phrases, we traverse S from left to right and track in an array $A[1..d]$ the last position at which we encountered each preorder number as an annotation. When we reach the destination i of some phrase f_k from F, the rightmost previous occurrence will be at position $\max_{p \in [p_k, q_k]} A[p]$ (the solution of a *range maximum query*), as any occurrence of f_k is annotated with either p_k or the preorder number $p_{k'}$ of a phrase $f_{k'}$ that is a descendant of f_k in the phrase trie. Hence, if we have a dynamic data structure for range maximum queries, then we can compute each rightmost occurrence with one query.

The phrase trie can be obtained in linear time as follows. We compute the suffix tree for the string $S' = S\#_0 f_1 \#_1 f_2 \#_2 \dots \#_{z-1} f_z \#_z$, where each $\#_k$ is a unique separator symbol. This takes $O(n)$ time. For any f_k from F, the parent of the leaf that spells suffix $f_k \#_k \dots$ is exactly the node that spells f_k. Thus, we can mark the d nodes that spell phrases from F in $O(|F|)$ time. It is then easy to compute the nearest marked ancestor of each node in $O(n)$ time. The phrase trie is obtained by creating a new tree that contains only the marked nodes and an artificial root. The new parent of a marked node is its nearest marked ancestor (or the artificial root node if it does not exist). Finally, we compute the preorder numbers in the phrase trie, and also annotate the corresponding marked nodes in the suffix tree with these numbers. Then, the annotation of text position i is the annotation of the nearest marked ancestor of the leaf that corresponds

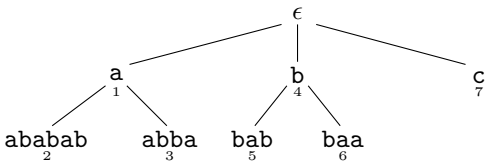

Figure 5.1: The phrase trie for the LZ factorization a|b|b|a|ab|ababab|bab|c|abba|baa where F is all the distinct phrases. Below each node is the preorder number.

to text position i in the suffix tree. Hence we obtain the annotations in $O(n)$ time.

Finally, we solve dynamic range maximum queries (RMQ) for A. The updates are incremental in the sense that every update is the new global maximum (i.e., the rightmost text position processed so far). Therefore, we can maintain a dynamic RMQ data structure for A with $O(\frac{1}{\epsilon})$ time updates and $O(d^\epsilon)$ time queries using the standard technique of square-root decomposition, generalized to arbitrary ϵ. For $\epsilon = \frac{1}{2}$, we split A into blocks of size $\Theta(\sqrt{d})$ and maintain the maximum of each block, which we can update in constant time whenever we update an entry of A. To answer queries we need to scan at most $O(\sqrt{d})$ elements in A that are in blocks that are only partially overlapped by the query range. Then, we also scan the $O(\sqrt{d})$ maxima of blocks that are fully contained in the query range. Thus, we take $O(\sqrt{d})$ time. This generalizes to smaller ϵ by recursively subdividing the blocks into $\frac{1}{\epsilon}$ layers, leading to $O(\frac{1}{\epsilon})$ update time and $O(d^\epsilon)$ query time. Each phrase in F incurs a range query and each text position an update. We perform $|F|$ range queries and n updates in $O(\frac{n}{\epsilon} + |F|\, d^\epsilon)$ time. This concludes the proof of Theorem 12(b).

5.4.3 Infrequent Phrases

Given an LZ-like parsing $S = f_1 \dots f_z$, we say that a phrase f_k is *infrequent* if $|\{k' \in [1, z] \mid f_{k'} = f_k\}| = O(\log n)$, i.e., if it occurs at most $O(\log n)$ times in the parsing. We now show how to resolve all infrequent phrases in $O(n)$ time, and we begin by establishing a data structure that is crucial for our solution.

Lemma 14. *Let $m, n \in [1, 2^w]$. For a tree of m nodes, labeled with preorder numbers from $[1, m]$, after an $O(m) + o(n)$ time preprocessing, and in $O(m) + o(n)$ space, we can maintain a data structure for nearest marked ancestor queries with the following operations.*

- *mark/unmark a node $i \in [1, m]$ with d_i descendants in $O(1 + d_i/\log n)$ time*
- *check if a node $i \in [1, m]$ is marked in $O(1)$ time*
- *check if a node $i \in [1, m]$ has a marked ancestor in $O(1)$ time*
- *output the nearest marked ancestor j of a node $i \in [1, m]$ in $O(1 + d_j/\log n)$ time, where d_j is the number of descendants of j.*

Proof. We compute the balanced parenthesis sequence [Nav16, Chapter 7] (BPS) $B[1..2m]$ of the tree by re-running the traversal used to obtain preorder numbers (with an artificial parent edge for the root to start the traversal). When we walk down the edge to node i, we append i's opening parenthesis to B, when we walk up the edge from node i we append its closing one. The ith opening parenthesis (in left to right order) belongs to node i, and between i's opening and closing parentheses there are exactly all the parentheses corresponding to descendants of i. We preprocess B such that given node $i \in [1, m]$ we can lookup the positions $\mathsf{open}(i)$ and $\mathsf{close}(i)$ of its respective opening and closing parentheses in B in constant time. This is possible with a simply linear scan in $O(m)$ time and space. For open, we also compute the inverse mapping $\mathsf{prenum}(\mathsf{open}(i)) = i$.

We use two additional bitvectors $A[1..2m]$ and $R[1..2m]$, both initialized with zeroes. When asked to mark node i, we set the bits $A[\mathsf{open}(i)]$ and $A[\mathsf{close}(i)]$ (marking the respective parentheses in B as *active*), and additionally we set the entire range $R[\mathsf{open}(i) + 1..\mathsf{close}(i)]$ one word at a time (indicating that nodes

whose opening parentheses lie in this region have a marked ancestor). If i has d_i descendants, then it holds $\mathsf{close}(i) - \mathsf{open}(i) = 1 + 2d_i$, and thus the procedure takes $O(1 + d_i/w)$ time. A node i is marked if and only if $A[\mathsf{open}(i)]$ is set, and it has a marked ancestor if and only if $R[\mathsf{open}(i)]$ is set (we do not consider a node to be its own ancestor). Both can be tested in constant time. Finding the nearest marked ancestor of i is more involved, and we explain it later.

When unmarking a node i, we unset the bits $A[\mathsf{open}(i)]$ and $A[\mathsf{close}(i)]$. If i currently has a marked ancestor, then there is no need to unset the range in R associated with i. Otherwise, we cannot simply unset the entire range $R[\mathsf{open}(i)+1..\mathsf{close}(i)]$ because it may have also been set by descendants of i. Hence we have to leave segments corresponding to marked nodes untouched. Starting at position $k = \mathsf{open}(i) + 1$, we scan $A[k..\mathsf{close}(i)]$ from left to right and keep track of the excess of opening active parentheses, which is initially $e = 0$. We perform the scan in blocks of size $w' = \lfloor \log n/7 \rfloor$. Processing $A[k..k + w')$ works as follows. We scan the block from left to right. For each position $A[j]$ in the block, we first check if currently $e = 0$. If yes, then we unset bit $R[j]$. Afterwards, if $A[j] = 1$, we increment e if $B[j]$ is an opening parenthesis, and decrement e otherwise. Once we reach the end of the block, we increase k by w' and continue with the next block, until we reach position $\mathsf{close}(i)$. This way, we avoid unsetting parts of R that have to remain active. However, the procedure takes $O(d_i)$ time, or $O(w')$ time per block.

The processing of block $A[k..k + w')$ depends only on $A[k..k + w')$, $B[k..k + w')$, $R[k..k + w')$ and $\min(e, w')$ (if $e > w'$, then the excess cannot reach 0 while processing the block). Thus it depends on $3w' + \log w' \leq \log n/2$ bits of information, and in principle there are fewer than $2^{\log n/2} = \sqrt{n}$ distinct instances of the procedure. In a lookup table, we precompute for each possible $A[k..k + w')$, $B[k..k + w')$, $R[k..k + w')$, and $\min(e, w')$ the result of the procedure, i.e., the total increment or decrement that we have to apply to e, and the new value of $R[k..k + w')$. The lookup table has $O(\sqrt{n})$ entries, and each of them can be computed naively in $O(\text{polylog}(n))$ time. Using the table, an entire block $A[k..k + w')$ can be processed in constant time (and handling the last block that is possibly shorter than w' can be solved with additional lookup tables for each shorter block length). Thus, we can unmark a node in $O(1 + d_i/w') = O(1 + d_i/\log n)$ time.

We have already shown how to check if i has a marked ancestor in constant time. If we also want to output the nearest marked ancestor, then we start at position $o = \mathsf{open}(i)$. Similarly to the technique for unmarking nodes, we now scan $A[1..o]$ and $B[1..o]$ from *right to left* and keep track of the excess of active *closing* parentheses. As soon as the excess becomes negative, we have found the opening parenthesis of the nearest marked ancestor. If this parenthesis is at position o', then the ancestor is $j = \mathsf{prenum}(o')$. We can implement this procedure with lookup tables (similar to unmarking nodes), and thus it takes $O(1 + d_j/\log n)$ time, where d_j is the number of descendants of j. □

Resolving the Phrases. Now we are ready to resolve the infrequent phrases. We first build the phrase trie including only the infrequent phrases, and compute the mapping from phrases to preorder numbers. We also annotate each text position i with the preorder number corresponding to the longest infrequent phrase that is a prefix of $S[i..n]$ (this works just like in Section 5.4.2). We prepare the phrase trie for nearest marked ancestor queries with Lemma 14.

Now we scan S from right to left. For each text position i, we first try to resolve phrases, which we explain in a moment. After that, if i is the destination of a phrase f_k with preorder number p_k, we mark node p_k in the phrase trie (indicating that the phrase needs to be resolved). We also store $P[p_k] = k$ in an array of size at most z. This is necessary because the preorder numbers correspond to the *distinct* infrequent phrases, and thus the mapping from preorder numbers to phrases is not necessarily injective. Later, we resolve f_k by discovering that node p_k is marked, and we will then need to be able to lookup $k = P[p_k]$. Note that we never try to resolve two phrases with the same preorder number at the same time, since the one further to the left would have already resolved the other one.

For every text position i, with annotation q_i, we check if q_i has a marked ancestor. If it does, we obtain the nearest marked ancestor p of q_i, corresponding to phrase $f_{P[p]}$. By the construction of the phrase trie and the annotations of text positions, $f_{P[p]}$ is a prefix of $T[i..n]$. Since we have not unmarked the node yet, and due to the right-to-left processing order, it follows that i is the rightmost source of $f_{P[p]}$. We unmark p.

Analyzing the Complexity. The preprocessing for the nearest marked ancestor structure takes $O(z) + o(n)$ time and space. For each text position, annotated with q_i, we check if q_i has a marked ancestor in overall $O(n)$ time. Whenever this is the case, we also find its nearest marked ancestor. However, we will then also immediately unmark the nearest marked ancestor, and thus the total time for finding marked ancestors is the same as the time for unmarking nodes, which is bounded by the time for marking them.

Now we analyze the total time for marking nodes. Let m be the number of nodes in the phrase trie (or equivalently the number of distinct infrequent phrases). We mark nodes $O(z)$ times, and thus the total time is $O(z)$ plus the sum of all the $O(d_i/\log n)$ terms. For now, we assume that each node gets marked exactly once. Then the time is $O(\frac{1}{\log n} \cdot \sum_{i=1}^{m} d_i)$. Let a_i denote the number of ancestors of a node i, and observe that $\sum_{i=1}^{m} d_i = \sum_{i=1}^{m} a_i$ (because in both sums each combination of descendant and ancestor contributes value 1 to the sum). If node i corresponds to a phrase f_k, then the number of ancestors of i is bounded by $a_i < |f_k|$, since each ancestor represents a phrase that is a proper prefix of f_k. Hence the time is $O(\frac{1}{\log n} \cdot \sum_{i=1}^{z} |f_k|) = O(n/\log n)$. We assumed that each node gets marked exactly once. Since we only consider infrequent phrases, each node gets marked $O(\log n)$ times, and thus the time is $O(n)$. This concludes the proof of Theorem 12(c).

5.4.4 Close Phrases

Given an LZ-like parsing $S = f_1 \dots f_z$, we say that a phrase f_k with destination i is *close* if its rightmost source is j and $i - j = O(\log n)$. We now show how to resolve all close phrases in $O(n)$ time. Let $\gamma = \Theta(\log n)$. If a phrase at destination i is of length at least γ, then we can afford $O(\log n)$ time to resolve it. We consider each $j \in [i - r, i)$ with $r = O(\log n)$ as a potential source. Checking if j is a source of i takes constant time with an LCE data structure (e.g., [KK19]). Thus we can resolve all close phrases of length at least γ in $O(n)$ time.

For the phrases of length less than γ, we extract copies of overlapping segments $s_0, \dots, s_{\lfloor n/2\gamma \rfloor}$ where $\forall i \in [1, \lfloor n/2\gamma \rfloor] : s_i = S[1 + 2(i-1)\gamma \dots \min(2(i+1)\gamma, n)]$. We modify each segment s_i by rank-reducing the alphabet of s_i to (a subset of) $[1, 4\gamma]$, which takes $O(n)$ total time by radix sorting all segments in batch. Then, we offset the alphabets such that s_i is over alphabet $[1 + 4(i-1)\gamma, 4i\gamma]$. We concatenate all segments s_i into $S' = s_0 s_1 \dots s_{\lfloor n/2\gamma \rfloor}$.

Each phrase of length less than γ is fully contained in the right half of at least one segment (apart from possible phrases with destination in the first 2γ position of S, which we solve with the LCE data structure in $O(\text{polylog}(n))$ time). We map each phrase of length less than γ to a corresponding destination in S' such that if the destination is within some segment s_j then the phrase is fully contained in the right half of s_j. This results in a subset of an LZ-like factorization of S'. Since the segments have disjoint alphabets, all phrases in the subset are infrequent an can be solved with Theorem 12(c). We only have to map the sources back to original text positions, which is easily done in linear time. Hence we have shown Theorem 12(d).

Chapter 6

Faster Compression of Deterministic Finite Automata

Faster Compression of Deterministic Finite Automata

Philip Bille phbi@dtu.dk
Inge Li Gørtz inge@dtu.dk
Max Rishøj Pedersen mhrpe@dtu.dk

Abstract

Deterministic finite automata (DFA) are a classic tool for high throughput matching of regular expressions, both in theory and practice. Due to their high space consumption, extensive research has been devoted to compressed representations of DFAs that still support efficient pattern matching queries. Kumar et al. [SIGCOMM 2006] introduced the *delayed deterministic finite automaton* (D^2FA) which exploits the large redundancy between inter-state transitions in the automaton. They showed it to obtain up to two orders of magnitude compression of real-world DFAs, and their work formed the basis of numerous subsequent results. Their algorithm, as well as later algorithms based on their idea, have an inherent quadratic-time bottleneck, as they consider every pair of states to compute the optimal compression.

In this work we present a simple, general framework based on locality-sensitive hashing for speeding up these algorithms to achieve sub-quadratic construction times for D^2FAs. We apply the framework to speed up several algorithms to near-linear time, and experimentally evaluate their performance on real-world regular expression sets extracted from modern intrusion detection systems. We find an order of magnitude improvement in compression times, with either little or no loss of compression, or even significantly better compression in some cases.

6.1 Introduction

Regular expressions are a widely used tool for signature detection and play a key role in modern network applications such as monitoring, balancing and intrusion detection. The standard solution to achieve high throughput is converting the regular expressions into *deterministic finite automata* (DFA), as they are very efficient in practice and have asymptotically optimal query time. Unfortunately DFAs require a lot of memory, which makes them unsuitable in many real-world applications where the regular expression sets are prohibitively large and complex. A key technique to make DFAs feasible for real-word data sets is to exploit the massive redundancy via *compression* [BC08, BTC06, KSE08, TSCV04, AFS+12, BC07a, KDY+06, FGP+08, FPG+11, AFS+15, QWF+11, SLHW17, MPN+10, PLT14, LT14, BC13, KTW06, BC07b, LSL+17, MMK18, GWX+23].

A classic, well-cited result in this line of work is the *delayed deterministic finite automaton* [KDY+06], with several subsequent based on this technique [MPN+10, PLT14, LT14, BC13, KTW06, BC07b, LSL+17, MKMK18, GWX+23]. The basic observation is that many states share a large number of *equivalent transitions*, that is, they have identical labels and destination states. If two states share a significant number of equivalent transitions, the common transitions can be replaced with single unlabeled *default transition*. For real-world DFAs this can achieve compression by two orders of magnitude. To decide which labeled transitions to replace, current algorithms count the number of equivalent transitions between every pairs of states. They therefore take $\Omega(n^2)$ time to compress a DFA of n states. This is infeasible for many modern data sets where automatons can have thousands or millions of states. In this paper we present a general framework for circumventing this bottleneck, based on locality-sensitive hashing. Using a simple, efficient hashing scheme that is locality-sensitive with regards to the similarity of states, i.e., their pair-wise compressibility, we effectively sample a sub-quadratic number of candidate default transitions that each have high probability of contributing significantly to the compression.

6.1.1 Our Contribution

We present a simple, general framework for efficient compression of DFAs with default transitions. This enables faster compression times for the class of algorithms based on the general idea of D^{2}FAs. We apply our technique to several such compression algorithms, and experimentally evaluate the resulting algorithms on collections of regular expressions extracted from real-world intrusion detection system. We find an order of magnitude improvement in compression time, with either no or minor loss of compression. In some cases we even achieve significantly better compression. Specifically we show the following:

- We speed up the general D^{2}FA construction algorithm [KDY$^+$06] from quadratic to near-linear time. Experimentally this improves the running time by an order of magnitude without loss of compression, for all data sets we tested.
- Kumar et al. [KDY$^+$06] presented a variant of the general compression algorithm that bounds the worst-case query time in the resulting D^{2}FA. Given a positive integer L they construct a D^{2}FA with a *longest delay* of L. Here longest delay refers to the maximum number of default transition traversed when matching any single character in the D^{2}FA. We present an algorithm for constructing D^{2}FAs with the same guarantees, using our framework. On our tested data sets we construct such D^{2}FAs up to 50 times faster while obtaining significantly better compression for the same longest delay bound.
- Becchi and Crowley [BC07b,BC13] presented an alternative approach to obtain query time guarantees, called the A-DFA. They construct D^{2}FAs with bounded *matching delay*. When matching a pattern P, any single character could traverse multiple default transitions, but over the entire pattern at most $|P|$ are traversed. We apply our technique to obtain a linear-time algorithm for constructing D^{2}FAs with the same guarantees. In our experiments this improves the running time up to 37 times with only minor loss of compression.

6.1.2 Related Work

A lot of work has been done on compressing DFAs. For an overview see surveys [XCS$^+$16, PS17]. One approach is compressing the alphabet and thereby reducing the size of the transition table [BC08, BTC06, KSE08, BC13, TJD$^+$17]. Here, if some sets of characters always cause the same transitions throughout the DFA they can be replaced by a single character [BC08,BTC06,BC13], and even if they occasionally diverge in some states [KSE08]. The alphabet can also be reduced by replacing infrequent characters with sequences of frequent characters [TJD$^+$17]. Becchi and Cambadi [BC07a] showed how states with similar sets of outgoing transitions could be merged into one, thereby compressing the set of states. Finally, another popular approach is compressing the set of transitions [MPN$^+$10, TSCV04, AFS$^+$12, BC07a, KDY$^+$06, KTW06, BC07b, FGP$^+$08, FPG$^+$11, BC13, PLT14, LT14, AFS$^+$15, QWF$^+$11, LSL$^+$17, MMK18, SLHW17]. A well-cited result in this line of work is the D^{2}FA [KDY$^+$06] which compresses equivalent transitions between states, with several subsequent results utilizing default transitions [MPN$^+$10, PLT14, LT14, BC13, KTW06, BC07b, LSL$^+$17, MMK18, MKMK18, GWX$^+$23] A significant weakness of the D^{2}FA is the quadratic construction time, which is the topic of this paper. Patel et al. [PLT14] proposed a framework that circumvents this issue for D^{2}FAs constructed from sets of regular expressions, where each individual rule produces small DFAs. We note this technique is orthogonal to ours.

The idea of using locality-sensitive hashing to compress collections is not new [DI03, OMST02, DAS10, KDLT04, PWZ11, SHWH12, KH15, XJFH11, BGPT23]. To the best of our knowledge, however, we are the first to apply the technique to the problem of compressing deterministic finite automata.

6.2 Preliminaries

A *graph* $G = (V, E)$ is a set of *nodes* V (also called vertices) and a set of *edges* $E : V \times V$ between nodes. We call the two nodes of an edge its *endpoints*. If the edges have direction, i.e. $(u, v) \neq (v, u)$, then the graph is

directed, otherwise it is *undirected*. Edges can have an associated *weight*, in which case the graph is *weighted*. Edges can also have an associated *label*, in which case the graph is *labeled*. We denote an edge from u to v with label c as $(u, v)_c$ and say (u, v) is *c-labeled*. A path of length k between two nodes u_0 and u_k is a sequence of nodes $u_0, \ldots, u_k$ such that $(u_i, u_{i+1}) \in E$ for $0 \leq i < k$. It can also be viewed as the sequence of edges $(u_0, u_1), \ldots, (u_{k-1}, u_k)$. If $u_0 = u_k$ then p is a *cycle*. A set of nodes where every pair of nodes have a path between them is called a *connected component*. If a graph has only one connected component it is *connected*, otherwise it is *disconnected*.

A *tree* is a connected graph that contains no cycles. Due to this property, a tree with n nodes has $n - 1$ edges. A graph consisting of several trees is a *forest*. If a node has only one incident edge it is called a *leaf*. For a node v in a tree its *radius* is the length of the longest path from v to a leaf. The *diameter* of a tree is the length of the longest path between any two nodes in the tree. A *spanning tree* of a graph is a subgraph that contains all the nodes and is a tree. If the graph is weighted the *value* of a spanning tree is the sum of the weights of the edges in the tree. A *maximum spanning tree* (MST) is a spanning tree of maximum value.

A family of hash functions is *locality-sensitive*, for some similarity measure, if the probability of two objects hashing to the same value is *high* (lower-bounded for some parameter) when they are *similar* (similarity above some threshold) and, conversely, *low* when they are *dissimilar*. For simplicity we omit formal definitions, but see e.g. [HIM12] for details. There are different families of locality-sensitive hash functions for different distance or similarity measures, with some of the most popular being *simhash* [Cha02], *MinHash* [BCFM00] and *sdhash* [Rou10]. As an example, the MinHash of a set is the minimum element according to a uniformly random permutation. The probability that two sets A and B hash to the same value is exactly their Jaccard similarity $(|A \cap B|)/(|A \cup B|)$. As the probability of collision can be appropriately upper- and lower-bounded for chosen similarity thresholds, MinHash is locality-sensitive w.r.t. Jaccard similarity.

6.3 Delayed Deterministic Finite Automata

A *deterministic finite automaton* (DFA) is a 5-tuple $D = (Q, \Sigma, \delta, q_0, A)$ where Q is a set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the initial state and $A \subseteq Q$ is a set of accepting states. Throughout we let $n = |Q|$ denote the number of states. A DFA can be thought of as a *labeled directed graph* where Q is the set of nodes and each transition $\delta(u, c) = v$ is a labeled, directed edge $(u, v)_c$. See Figure 6.1 (A) for an example. For simplicity we assume every state has exactly one labeled transition for each character in the alphabet, i.e., δ is *total*, as in previous work. Given a pattern string P and a path p in D we say that p matches P if the concatenation of the labels of p equals P. We say a path that starts in q_0 and ends in A is *accepting*. We say D *accepts* a string P if there exists an accepting path that matches P. The *language* of D is the set of strings it accepts.

A *delayed deterministic finite automaton* (D^{2}FA) [KDY+06] is a deterministic finite automaton that is augmented with unlabeled *default transitions*. Formally a D^{2}FA is a 6-tuple $D^2 = (Q, \Sigma, \delta, q_0, A, F)$. Again Q is the set of states, Σ is the alphabet, δ is the transition functions, $q_0 \in Q$ is the initial state and $A \subseteq Q$ is the set of accepting states. Here $F : Q \to Q$ is the *default transition function*. Viewed as a graph, default transitions are ϵ-labeled directed edges, where ϵ is the empty string, and each state has at most one outgoing default transition. See Figure 6.1 (C) for an example. To transition from a state u according to a character c we follow a c-labeled transition if it exists, or otherwise follow the default transition:

$$\delta(u, c) = \begin{cases} v & \text{if } (u, v)_c \in D^2 \\ \delta(F(u), c) & \text{otherwise} \end{cases}$$

Note that for δ to be well-defined it must always be possible to reach a state from u that has a c-labeled transition. This implies that any cycle of default transitions must have an outgoing c-labeled transition, for any character c. To transition from a state u according a string $P = c_1 \ldots c_m$ we recursively transition according to each character:

$$\delta(u, c_1 c_2 \ldots c_m) = \delta(\delta(u, c_1), c_2 \ldots c_m).$$

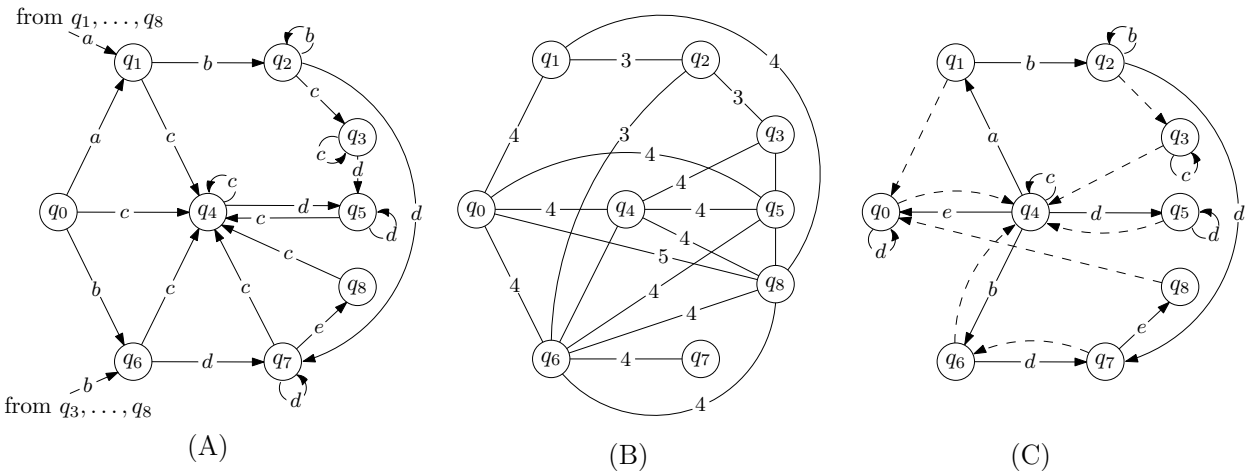

Figure 6.1: Example from [KDY+06]. (A) DFA D for regular expression `.*((ab+c+)|(cd+)|bd+e)`. Edges to q_0 are omitted. (B) Space reduction graph for D with edges annotated with similarity. Edges with similarity less than 4 omitted, except those connecting q_2 to avoid disconnecting the graph. (C) D^{2}FA equivalent to D. All transitions are shown, default transitions are dashed.

Given a character c and a path $p = u_1, \ldots, u_k$ we say p matches c if $\delta(u_1, c) = u_k$ and all but the last transition is default, i.e., $F(u_i) = u_{i+1}$ for $1 \leq i < k$. Note that then the concatenation of the labels of p equals c. Given a string $P = c_1 \ldots c_m$ and a path p we say p matches P if p is the concatenation of the paths matching the individual characters $c_1, \ldots, c_m$. Note that then the concatenation of the labels of p is P. We define acceptance as before.

We say two automata are *equivalent* if the language of both is the same. We say two transitions are *equivalent* if they have the same destination and label, i.e., $(u, w)_c$ and $(v, w)_c$ are equivalent. Given a DFA we can compress it by replacing sets of equivalent transitions with single default transitions to obtain an equivalent D^{2}FA with fewer total transitions. We define the *similarity* of two states u and v, denoted $sim(u, v)$, to be their number of equivalent transitions, that is, $sim(u, v) = |\{c \in \Sigma \mid \delta(u, c) = \delta(v, c)\}|$. See Figure 6.1 (B) for an illustration. Inserting a default transition (u, v) and removing the equivalent transitions from u does not affect the language but saves $sim(u, v) - 1$ transitions. Each transition we can remove without affecting the language we say is *redundant*.

Following a default transition does not consume an input character which introduces a *delay* when matching. We define the *longest delay* of D^2 to be maximum number of default transitions in any path matching a single character. That is, if the delay is d we must follow at most d default transitions to match any single character. Given a pattern P we define the *matching delay* of P in D^2 to be the number of default transitions in the path starting in q_0 and matching P.

6.4 Compression of DFAs with Default Transitions

In this section we outline the algorithm of Kumar et al. [KDY+06] that given a DFA D constructs an equivalent D^{2}FA. Let D^2 be an initially empty D^{2}FA with the same set of states as D. The algorithm then constructs the transitions of D^2 as follows.

Step 1: Space Reduction Graph Construct the complete graph over the states of D and to each edge (u, v) assign weight $sim(u, v)$. This is the *space reduction graph* (SRG). See Figure 6.1 (B) for an example.

Step 2: Maximum Spanning Tree Build a maximum spanning tree over the SRG. Root the spanning

tree in a central node, that is, a node of minimal radius, and direct all edges towards the root to obtain a directed spanning tree.

Step 3: Transitions For each edge (u, v) in the tree insert the default transition (u, v) into D^2. Copy every labeled transition from D into D^2 that is not redundant in D^2.

Step 1 takes $\Theta(n^2|\Sigma|)$ time as the complete graph over n states has $n(n-1)/2 = \Theta(n^2)$ edges and for each edge (u, v) they calculate $sim(u, v)$ by comparing the $|\Sigma|$ outgoing transitions of u and v. In Step 2 they construct the maximum spanning tree with Kruskal's algorithm [Kru56] which takes $O(m \log m) = O(n^2 \log n)$ time for the $m = \Theta(n^2)$ edges of the SRG. Step 3 takes $O(n|\Sigma|)$ time as they consider every transition in D. Thus the total running time is $O(n^2(\log n + |\Sigma|))$.[1]

The space reduction graph exactly captures the amount of transitions we can remove by inserting a default transition between each pair of states. Therefore it is the natural basis for most algorithms that use default transitions. Constructing default transitions along leaf-to-root paths in a tree ensures that there are no cycles and that no state has more than one outgoing default transition. A maximum spanning tree of the space reduction graph maximizes the number of edges made redundant by the default transitions, and thus the compression achieved. Directing the default transitions toward a central node reduces the longest delay of D^2 by minimizing the length of paths of default transitions (leaf-to-root paths) in the tree.

6.5 Fast Compression

We now show how to use locality-sensitive hashing to speed up the the above algorithm for constructing D^{2}FAs. The idea is to reduce the number of edges in the space reduction graph, i.e., *sparsifying* it.

Let again D be the input DFA. Let r and k be two positive integer constants. We replace Step 1 of the algorithm (constructing the complete SRG) and instead construct a sparse SRG (explained below) with $m = O(rn) = O(n)$ edges in $O(n(rk + |\Sigma|) = O(n|\Sigma|)$ time. The remainder of the algorithm then takes $O(m \log m) = O(n(|\Sigma| + \log n))$ time, which is also the total time complexity. The resulting D^{2}FA is not necessarily as small as the one achieved by constructing the complete SRG, but our experiments show that it essentially is in practice (see Section 6.8 for details).

6.5.1 Constructing a Sparse SRG

We begin with an initially empty graph $G = (Q, E)$ where the nodes are the states of D. First we add edges to connect the initial state to all other states, that is $E = \{(q_0, u) \mid u \neq v_0\}$. We then add edges to the graph in r rounds, where a single round is as follows:

We pick k unique random characters $c_1, \ldots, c_k \in \Sigma$. For each state $v \in Q$ we construct the sequence of k states $V = \delta(v, c_1), \ldots, \delta(v, c_k)$. We hash V into a single hash value $h(v)$ using a standard vector hashing scheme of Black et al. [BHK$^+$99]. We insert v into a table with key $h(v)$. For each unique hash value h_i, consider the set of states C_i that hash to h_i. For each state $u \in C_i$ we pick another state $v \in C_i$ uniformly at random and insert (u, v) into E, if it does not already exist.

After r rounds the algorithm terminates and we assign weights to each edge of G equal to the similarity of the endpoint states. We call G the *sparse space reduction graph*.

Hashing a single states takes $O(k)$ time so hashing all states takes $O(kn)$ time. Iterating over each state and sampling an edge takes $O(n)$ time so the r rounds take $O(rkn)$ time in total. Each round inserts at most a single edge per state so G has $m = O(rn)$ edges. Calculating similarity between two states takes $\Theta(|\Sigma|)$ time so assigning weight to all edges takes $O(rn|\Sigma|)$ time. The total time to construct G is thus $O(rn(k + |\Sigma|)) = O(n|\Sigma|)$.

We note that this hashing scheme is very similar to the scheme of Har-Peled et al. [HIM12] which is locality-sensitive w.r.t. the Hamming distance between bit-vectors. They hash a bit-vector by uniformly sampling k bits into a k-bit hash value. We hash a state by uniformly sampling k outgoing transitions into

[1] We note that the running time is not explicitly stated in their paper, but follows from their description.

a k-element vector, which is then compressed into a single hash value. By an analysis similar to theirs, this scheme is locality-sensitive w.r.t. the similarity of states. Any edges we sample are between states that hash to the same value, in some round, which requires (ignoring collisions in final vector hash) their transitions to be equivalent for each of the k characters sampled from Σ. The probability of this is high only when they are they are very similar, and therefore any edge in G is likely of high weight and can contribute significantly to the compression.

6.6 Compression with Bounded Longest Delay

We now present an efficient algorithm for constructing D^{2}FAs with bounded longest delay. The problem is: given a DFA D and a positive integer L, construct a D^{2}FA D^2 equivalent to D, such that the longest delay of D^2 is at most L. Recall that the longest delay of D^2 is at most the length of the longest path of default transitions in D^2. The algorithm we present is based on the similar cubic-time algorithm by Kumar et al. [KDY$^+$06], which we outline in the following section. After outlining their algorithm we present ours.

6.6.1 Bounded Longest Delay by Iteratively Constructing Small Trees

We now review the algorithm by Kumar et al. [KDY$^+$06]. It is a modification of the algorithm presented in Section 6.4, as follows.

They first construct the complete SRG, as before. Then they construct a spanning forest where each tree has diameter at most $\Delta = 2L$. To do this they again use Kruskal's algorithm, but now ignore any edges that would cause a tree diameter to exceed Δ. Furthermore, among the edges with maximum similarity they select one that causes a minimum increase to any tree diameter. After constructing the forest they root each tree in a central node, direct edges toward each root and construct transitions, as before.

Each tree has diameter at most Δ and is rooted in a central node. Thus any path of default transitions has length at most $\lceil \Delta/2 \rceil = L$, and the longest delay is at most L. As before, Kruskal's algorithm adds up to $n-1$ edges to the forest after sorting the $m = O(n^2)$ edges of the SRG. For each edge they check if the new tree would exceed the diameter constraint. To facilitate this check they maintain the radius of each node as the spanning forest is constructed. Each time an edge is added, merging two trees, the radius of every node in the new tree can change. In the worst case, each of the $O(n)$ edge additions causes every radii to change, so in total it takes $O(\sum_{i=1}^{n-1} i) = O(n^2)$ time to maintain the radii. We note that in practice each tree typically remains small and thus fewer updates are necessary. Nonetheless, the total time complexity is at least quadratic.[2]

Combining our sparsification technique with this algorithm does not improve the asymptotic running time. Though a sparse SRG has fewer edges, maintaining the radii still takes quadratic time. In the next section we present an alternative algorithm that does combine effectively with sparsification.

6.6.2 Fast Compression with Bounded Longest Delay

We now present an algorithm that uses sparsification to efficiently construct D^{2}FAs with bounded longest delay. Here we also construct a forest where each tree has a bounded diameter, but instead of iteratively constructing small trees, we instead construct one large maximum spanning tree and then remove (cut) edges until each tree in the resulting forest is small. Let again D be the input DFA and L be a given positive integer. The algorithm is then as follows:

Step 1: Construct SRG Construct a sparse SRG for D as in Section 6.5.1.

Step 2: Construct MST Construct a maximum spanning tree T_0 over the SRG using Kruskal's algorithm. Pick a central node v_0 in T_0 and then discard T_0. Construct a new maximum spanning tree T using

[2]It is unclear from the description of the algorithm exactly how edge selection is performed, which could affect the time complexity.

Prim's algorithm [Pri57] with v_0 as the initial node. When queuing a new edge (u, v), where u is the node already in T, assign weight $w'_{u,v} = sim(u, v) - 2^{d_v}$ where d_v is the distance from v_0 to v in T.

Step 3: Cut Edges Cut a minimum number of edges in T to obtain a forest with each tree diameter at most $\Delta = 2L$. Then direct the edges in each tree towards the root.

Step 4: Construct Transitions As in Step 3 in Section 6.4, create default transitions along edges in the trees and then copy in every labeled transitions from D that is not redundant in D^2.

After cutting, each tree has diameter at most $\Delta = 2L$. We orient edges towards each root so the longest delay of D^2 is at most $\lceil \Delta/2 \rceil = L$.

Step 1 takes $O(rn(k + |\Sigma|))$ time, as before, to construct an SRG with $m = O(rn)$ edges. Step 2 takes $O(m \log m) = O(rn + \log rn)$ time, as Prim's algorithm has the same complexity as Kruskal's. Step 3 takes $O(n)$ time by using the algorithm of Farley et al. [FHP81] to cut the necessary edges in a bottom up traversal of T. Step 4 takes $O(n|\Sigma|)$ time as it considers every edge in D. Thus the total construction time is $O(rn(k + |\Sigma| + \log rn)) = O(n(|\Sigma| + \log n))$.

Each edge (u, v) we cut results in $sim(u, v) - 1$ more labeled transitions in D^2, as that default transition is then not constructed. Intuitively, the lower the diameter of T, the fewer edges we cut to get each tree below the bound. Therefore we use an edge weight that trades similarity for lower diameter, as the lost similarity is often outweighed by the fewer cuts. We found the simple heuristic of w' performed well in practice. A similar idea was used in the implementation of the algorithm by Kumar et al [KDY+06]. Because T is not a maximum spanning tree w.r.t. similarity, the choice of initial node v_0 affects the total similarity of the final tree. We found that picking v_0 to be a central node in a MST w.r.t. similarity (T_0) yielded the best compression in practice. Note that we cut the minimum *number* of edges to uphold the diameter constraint. Alternatively we could cut edges of minimum *total similarity*, which could result in better compression. However, the chosen approach is simple and fast in practice, and because each edge in the SRG has near-maximum weight, the difference in compression is negligible.

6.7 Compression with Bounded Matching Delay

We now show an efficient construction algorithm for D^{2}FAs with bounded matching delay. The problem is: given a DFA D, construct a D^{2}FA D^2 such that matching a pattern P in D^2 traverses at most $|P|$ default transitions. The algorithm is based on the A-DFA [BC07b, BC13] algorithm, which we first outline.

6.7.1 Bounded Matching Delay by the A-DFA Algorithm

We now review the A-DFA algorithm of Becchi and Crowley [BC07b]. Let again D be the input DFA. Let the *depth* $d(v)$ of a state $v \in Q$ be the length of the shortest path from the initial state q_0 to v. That is, q_0 has depth zero, all neighbours of q_0 have depth one, all neighbours of those states (that are not also neighbours of q_0) have depth two, etc. The idea is to only add default transitions from higher-depth states to lower-depth states, i.e. they point towards the initial state.

Let D^2 be a D^{2}FA with initially no default transitions. Then the algorithm is as follows:

Step 1: Calculate Depth Calculate the depth $d(v)$ of each state $v \in Q$ by a breadth-first traversal of D.

Step 2: Construct Default Transitions For each state $u \in Q$ add default transition (u, v) to D^2, where v is the state such that $sim(u, v)$ is maximum and $d(v) < d(u)$.

Step 3: Construct Labeled Transitions Copy in every labeled transition from D that is not redundant in D^2.

Step 1 takes $O(n|\Sigma|)$ time to traverse D. Step 2 takes $O(n^2|\Sigma|)$ time as it calculates similarity for every pair of states. Step 3 takes $O(n|\Sigma|)$ time as it considers every edge in D. Thus the total running time is $O(n^2|\Sigma|)$.

Matching a pattern P in D^2 requires following at most $|P|$ default transitions. To see why, consider the depth of the current state as the pattern is matched. Starting in q_0 it is zero, and when a character is matched, i.e., by following a labeled transition, the depth increases by at most one. When a default transition is followed the depth decreases, as default transitions only go from higher-depth to lower-depth states. Over the entire pattern the depth can increase at most $|P|$ and therefore decrease at most $|P|$ (states only have positive depth). Thus we follow at most $|P|$ default transitions and the total delay of D^2 is at most $|P|$.

Every state has exactly one outgoing default transition after Step 2. Because all default transition point towards the root, there cannot be any cycles of default transitions, and thus D^2 is a valid D^{2}FA. Note that the algorithm does not construct the complete SRG. However, selecting a maximum-similarity edge in Step 2 is equivalent to selecting a maximum-weight incident edge in the complete SRG (that also leads to lower-depth state), and asymptotically takes as much time as constructing the edges beforehand. Therefore this algorithm can be thought of as *implicitly* using the SRG.

6.7.2 Fast Compression with Bounded Matching Delay

We now speed up the A-DFA algorithm using sparsification. As the algorithm only implicitly uses the SRG we do not construct a sparse SRG as before, however, the idea is similar. Let again D be the input DFA, and let r and k be positive integer constants. We begin with a D^{2}FA D^2 that has no default transitions, i.e., $F(u) = u$ for $u \in Q$. Then the algorithm runs for r rounds, where a single round is as follows:

Pick k unique random characters $c_1, \ldots, c_k \in \Sigma$. For each state $v \in Q$ we construct the sequence $V = \delta(v, c_1), \ldots, \delta(v, c_k)$, and hash V into a single hash value $h(v)$. We insert v into a table with key $h(v)$. For each unique hash value h_i we consider the set of states C_i that hash to that value. For each state $u \in C_i$ we pick another state $v \in C_i$ uniformly at random. If v has lower depth and the default transition (u, v) compresses better than the current default transition of u, i.e., $d(v) < d(u)$ and $sim(u, v) > sim(u, F(u))$, we update the default transition of u to point to v, otherwise we just continue. After r rounds the algorithm terminates.

In a single round we spend $O(k|\Sigma|)$ time per state to combine k transitions and calculate similarity of the potential new default transition. Therefore the r rounds take $O(rkn|\Sigma|) = O(n|\Sigma|)$ time.

6.8 Experimental Evaluation

We implemented the above algorithms and evaluated their performance on regular expression rulesets extracted from real-world intrusion detection systems. In this section we present the experimental results.

6.8.1 Setup

Experiments were run on a machine with an Intel Xeon Gold 6226R 2.9GHz processor and 128GB of memory. The operating system was Scientific Linux 7.9 kernel version 3.10.0-1160.80.1.el7.x86_64. Source code was compiled with `g++` version 9.4 with options `-Wall -O4`. The input to each algorithm is a DFA constructed from a set of regular expressions. We measured the time for constructing an equivalent D^{2}FA for the input DFA, using the `clock` function of the C standard library. Source code and data sets are available upon request.

6.8.2 Data Sets

We extracted updated versions of the data sets used in prior work from publicly available regular expression sets in real-world intrusion detection systems. These systems are *Snort* [3], *Suricata* [4] and *Zeek* [5].

[3] `https://www.snort.org/`
[4] `https://suricata.io/`
[5] `https://zeek.org/`

Name	States	Rules	Average length of rules	% using wildcards (`*`, `+`, `?`)	% using length restrictions (`{,k,+}`)
`Snort1`	9,435	23	34.5	60.9	39.1
`Snort2`	19,350	60	28.6	40.0	25.0
`Snort3`	40,012	70	27.9	40.0	21.4
`Snort4`	6,577,094	99	27.7	51.5	16.2
`Suricata1`	13,557	32	117.2	90.6	0.0
`Suricata2`	29,278	37	111.2	89.2	2.7
`Suricata3`	75,596	49	105.7	85.7	2.0
`Suricata4`	7,122,336	196	98.5	86.7	1.0
`ZeekNet`	4,874,200	23	48.0	47.8	26.1
`ZeekFile`	4,975,198	41	57.8	19.5	34.1

Table 6.1: Input DFAs and characteristics of the corresponding regular expression set.

Algorithm	Description
D^2FA	The algorithm of [KDY+06], described in Section 6.4.
D^2FA-Ld	The algorithm of [KDY+06] for bounding longest delay, described in Section 6.6.1.
D^2FA-Ld-Cut	The algorithm presented in Section 6.6.2 for bounding longest delay, but using the complete SRG instead of a sparse SRG.
D^2FA-Md	The algorithm of [BC13] for bounding matching delay, described in Section 6.7.
Sparse-D^2FA	The algorithm presented in Section 6.5.
Sparse-D^2FA-Ld	As D^2FA-Ld but using a sparse SRG instead of the complete SRG.
Sparse-D^2FA-Ld-Cut	The algorithm presented in Section 6.6.2 for bounding longest delay.
Sparse-D^2FA-Md	The algorithm presented in Section 6.7.2 for bounding matching delay.

Table 6.2: The algorithms included in the experiments.

We filtered out some rules that use advanced regex features, similar to prior work, as well as any rules that correspond to a DFA of more than 12,000 states. We generated DFAs of varying size by selecting prefixes of each expression collection, so, for example `Snort3` includes all the rules of `Snort1` and `Snort2`. Table 6.1 lists the input DFAs along characteristics of the corresponding regular expression set. All regular expressions are ASCII and have alphabet size $\Sigma = 256$.

6.8.3 Algorithms Tested

The algorithms we compare are listed in Table 6.2. For all algorithms we ignore SRG edges of low similarity, as in previous work, as they have minimal impact on the final compression.

We evaluated four locality-sensitive hashing schemes: the scheme presented in Section 6.5.1, with and without replacement, and MinHash over the set of outgoing transitions, either with one or k random permutations of the universe. The best performing scheme, both in terms of speed and compression, was the one presented in Section 6.5.1 (without replacement), and we use that in all experiments. We also investigated the effect of the parameters k and r. We found $k = 8$ gave the best compression and increasing r resulted in better compression but increased the running time linearly. All experiments in this section were run with $k = 8$ and $r = 512$.

6.8.4 General Compression

Here we evaluate the impact of sparsification on the general compression algorithm. We compare algorithms D^2FA and Sparse-D^2FA. The results are shown in Table 6.3.

Algorithm	Time	Comp.	Time	Comp.	Time	Comp.
	`Snort1`		`Snort2`		`Snort3`	
D^{2}FA	2.78	**2.82**	16.81	**2.20**	95.14	**2.10**
Sparse-D^{2}FA	**1.69**	**2.82**	**4.14**	**2.20**	**8.08**	**2.10**
	`Suricata1`		`Suricata2`		`Suricata3`	
D^{2}FA	6.38	**0.93**	46.14	**0.92**	347.15	**0.89**
Sparse-D^{2}FA	**3.00**	**0.93**	**7.16**	**0.92**	**20.36**	**0.89**
	`Snort4`		`Suricata4`			
Sparse-D^{2}FA	**1950.73**	**2.18**	**3079.39**	**0.93**		
	`ZeekNet`		`ZeekFile`			
Sparse-D^{2}FA	**1221.08**	**7.23**	**858.64**	**2.85**		

Table 6.3: Performance of general compression. *Time* is the number of seconds to construct the D^{2}FA and *Comp.* is the ratio between the number of transitions in the input DFA and the resulting D^{2}FA. Experiments that failed to terminate in at most 12 hours are omitted.

We observe that Sparse-D^{2}FA achieves the same compression but is significantly faster, for all the tested DFAs. The running time is improved by a factor two for the smallest inputs and up to roughly a factor 17 for the larger inputs. D^{2}FA failed to terminate in 12 hours for any of the large DFAs (and is therefore omitted from the table) while Sparse-D^{2}FA terminated in around 50 minutes for the largest. As D^{2}FA did not terminate we can not directly evaluate the impact on compression. However, note that the compression of `Snort4` is very close to that of `Snort3`, and similarly for `Suricata4` and `Suricata3`.

6.8.5 Compression with Bounded Longest Delay

Here we evaluate the impact of sparsification on algorithms for constructing D^{2}FAs with bounded longest delay. We set the longest delay bound to $L = 2$ and compare the algorithms: D^{2}FA-Ld, Sparse-D^{2}FA-Ld, D^{2}FA-Ld-Cut and Sparse-D^{2}FA-Ld-Cut. The results are shown in Table 6.4.

We observe that Sparse-D^{2}FA-Ld-Cut significantly out-performs D^{2}FA-Ld in both running time and compression. Compared to D^{2}FA-Ld it is around 14 times faster for the smaller inputs and around 50 times faster for the largest, while simultaneously obtaining a D^{2}FA that is two to four times smaller. Notably it achieves compression that is no more than a factor two worse than the D^{2}FA algorithm for D^{2}FAs without delay bound (see Table 6.3). It is also the only algorithm that terminated in less than 12 hours for the large inputs. For the largest DFA, with 7 million states, it terminated in 81 minutes and achieved more than 98% compression.

We note that Sparse-D^{2}FA-Ld obtains worse compression than D^{2}FA-Ld, constructing a D^{2}FA that is around 10–15% larger for the small Snort data sets and around 250% for the small Suricata data sets. Here sparsification incurs a loss of compression. We suggest that this approach to bounding longest delay is particularly sensitive to the set of edges available, requiring more options for each edge than other approaches. This could also explain why (Sparse-)D^{2}FA-Ld-Cut generally performs better than (Sparse-)D^{2}FA-Ld.

Interestingly Sparse-D^{2}FA-Ld-Cut obtains slightly better compression than D^{2}FA-Ld-Cut for the two smallest inputs, even though SRG edges considered by the former is a strict subset of the edges considered by the latter. We verified this to be Sparse-D^{2}FA-Ld-Cut by chance constructing an initial spanning tree that requires fewer cuts to satisfy the path bound constraint.

Algorithm	Time	Comp.	Time	Comp.	Time	Comp.
	Snort1		Snort2		Snort3	
D^{2}FA-Ld	28.94	10.79	148.22	10.19	502.41	9.74
Sparse-D^{2}FA-Ld	17.47	12.41	76.20	11.44	285.40	10.87
D^{2}FA-Ld-Cut	3.81	4.81	20.97	**4.02**	98.67	**3.81**
Sparse-D^{2}FA-Ld-Cut	**2.02**	**4.62**	**4.69**	4.04	**10.14**	3.89
	Suricata1		Suricata2		Suricata3	
D^{2}FA-Ld	55.70	4.61	253.19	5.55	1266.20	5.59
Sparse-D^{2}FA-Ld	46.60	12.16	166.79	12.21	990.55	10.83
D^{2}FA-Ld-Cut	7.21	1.49	47.00	**1.31**	368.94	**1.39**
Sparse-D^{2}FA-Ld-Cut	**3.57**	**1.20**	**8.12**	1.45	**23.48**	1.40
	Snort4		Suricata4			
Sparse-D^{2}FA-Ld-Cut	**2182.68**	**3.91**	**4880.69**	**1.39**		
	ZeekNet		ZeekFile			
Sparse-D^{2}FA-Ld-Cut	**934.80**	**7.94**	**961.73**	**3.50**		

Table 6.4: Performance of algorithms that produce a D^{2}FA with a longest delay of at most $L = 2$. *Time* is the number of seconds to construct the D^{2}FA and *Comp.* is the ratio between the number of transitions in the input DFA and the resulting D^{2}FA. Experiments that failed to terminate in at most 12 hours have been omitted.

6.8.6 Compression with Bounded Matching Delay

Here we evaluate the impact of sparsification on the algorithms for constructing DFAs with bounded matching delay. We compare algorithms D^{2}FA-Md and Sparse-D^{2}FA-Md. The results are shown in Table 6.5.

We find that Sparse-D^{2}FA-Md is an order of magnitude faster with only minor loss of compression. For the small inputs Sparse-D^{2}FA-Md compresses slightly worse than D^{2}FA-Md. For the Snort DFAs the difference is less than 10%, while for the Suricata DFAs it is between 48% and 90%. This suggests that for some inputs the algorithm is more sensitive to sparsification. However, Sparse-D^{2}FA-Md runs significantly faster than D^{2}FA-Md. For the smallest inputs it is around twice as fast and for the largest it is 16–38 times faster. We attempted to evaluate the algorithms for the very large DFAs but failed to achieve good compression, and were unable to determine if it was due to sparsification or the A-DFA algorithm, as only Sparse-D^{2}FA-Md terminated.

6.9 Conclusion

We have shown that locality-sensitive hashing can be utilized to improve the running time of DFA compression algorithms based on default transitions. This is achieved by sparsifying the space reduction graph of the DFAs, reducing the number of edges from $\Theta(n^2)$ to $O(rn)$, where n is the number of states and r is a constant parameter. We applied this technique to obtain efficient construction algorithms for D^{2}FAs, as well as D^{2}FAs with bounded longest delay and D^{2}FAs with bounded matching delay. We implemented and experimentally evaluated the algorithms on real-world regular expression sets, comparing against the non-sparsified algorithms. For general D^{2}FAs we found up to 17 times improvement in compression time without any loss of compression. For D^{2}FAs with bounded longest delay our algorithm was up to 50 times faster while obtaining significantly better compression for the same longest delay bound. For D^{2}FAs with bounded matching delay we are up to a factor 37 times faster, with only slightly worse compression.

Patel et al. [PLT14] proposed an orthogonal technique for circumventing the quadratic construction time of D^{2}FAs. An interesting open problem is whether our techniques can be effectively combined.

Algo.	Time	Comp.	Time	Comp.	Time	Comp.
	`Snort1`		`Snort2`		`Snort3`	
D^{2}FA-Md	2.32	**6.63**	15.64	**6.77**	90.36	**6.64**
Sparse-D^{2}FA-Md	**0.90**	6.89	**2.36**	7.27	**5.63**	7.18
	`Suricata1`		`Suricata2`		`Suricata3`	
D^{2}FA-Md	5.57	**0.94**	45.77	**0.94**	355.58	**0.92**
Sparse-D^{2}FA-Md	**1.12**	1.39	**3.43**	1.38	**9.29**	1.74

Table 6.5: Performance of algorithms that construct a D^{2}FA with total delay of at most $|P|$ for a given pattern P. *Time* is the number of seconds to construct the D^{2}FA and *Comp.* is the ratio between the number of transitions in the input DFA and the resulting D^{2}FA.

Bibliography

[AB20] Amihood Amir and Itai Boneh. Update query time trade-off for dynamic suffix arrays. In *Proc. 31st ISAAC*, volume 181, pages 63:1–63:16, 2020.

[AB21] Amihood Amir and Itai Boneh. Dynamic suffix array with sub-linear update time and polylogarithmic lookup time. *CoRR*, abs/2112.12678, 2021.

[ACLL14] Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In *Proc. 41st ICALP*, pages 114–125, 2014.

[AFG+14] Amihood Amir, Gianni Franceschini, Roberto Grossi, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Managing Unbounded-Length Keys in Comparison-Driven Data Structures with Applications to Online Indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014.

[AFS+12] Rafael Antonello, Stenio F. L. Fernandes, Djamel Sadok, Judith Kelner, and Géza Szabó. Deterministic finite automaton for scalable traffic identification: The power of compressing by range. In *Proc. NOMS 2012*, pages 155–162, 2012.

[AFS+15] Rafael Antonello, Stenio F. L. Fernandes, Djamel Fawzi Hadj Sadok, Judith Kelner, and Géza Szabó. Design and optimizations for efficient regular expression matching in DPI systems. *Comput. Commun.*, 61:103–120, 2015.

[AHdLT97] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th ICALP*, pages 270–280, 1997.

[AHT00] Stephen Alstrup, Jacob Holm, and Mikkel Thorup. Maintaining center and median in dynamic trees. In *Proc. 7th SWAT*, pages 46–56, 2000.

[AKL+16] Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap. In *Proc. 27th ISAAC*, pages 12:1–12:12, 2016.

[AKLL05] Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *Proc. 12th SPIRE*, volume 3772, pages 67–78. Springer, 2005.

[ALU02] Amihood Amir, Gad M. Landau, and Esko Ukkonen. Online timestamped text indexing. *Inf. Process. Lett.*, 82(5):253–259, 2002.

[AN08] Amihood Amir and Igor Nor. Real-time indexing over fixed finite alphabets. In *Proc. 19th SODA*, pages 1086–1095, 2008.

[APS04] Alberto Apostolico, Cinzia Pizzi, and Giorgio Satta. Optimal discovery of subword associations in strings. In *Proc. 7th DS*, volume 3245, pages 270–277, 2004.

[APU11] Alberto Apostolico, Cinzia Pizzi, and Esko Ukkonen. Efficient algorithms for the discovery of gapped factors. *Algorithms Mol. Biol.*, 6:5, 2011.

[AR02] Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In *Proc. 13th SODA*, pages 947–953, 2002.

[AS09] Alberto Apostolico and Giorgio Satta. Discovering subword associations in strings in time linear in the output size. *J. Discrete Algorithms*, 7(2):227–238, 2009.

[BB94] P Bucher and A Bairoch. A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In *Proc. 2nd ISMB*, pages 53–61, 1994.

[BC07a] Michela Becchi and Srihari Cadambi. Memory-efficient regular expression search using state merging. In *Proc. 26th INFOCOM*, pages 1064–1072, 2007.

[BC07b] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proc. ANCS 2007*, pages 145–154, 2007.

[BC08] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *Proc. ANCS 2008*, pages 50–59, 2008.

[BC13] Michela Becchi and Patrick Crowley. A-DFA: A time- and space-efficient DFA compression algorithm for fast regular expression evaluation. *ACM Trans. Archit. Code Optim.*, 10(1):4:1–4:26, 2013.

[BCFG17] Philip Bille, Patrick Hagge Cording, Johannes Fischer, and Inge Li Gørtz. Lempel-Ziv compression in a sliding window. In *Proc. 28th CPM*, volume 78, pages 15:1–15:11, 2017.

[BCFM00] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[BFGLO09] Gerth Stølting Brodal, Rolf Fagerberg, Mark Greve, and Alejandro López-Ortiz. Online sorted range reporting. In *Proc. 30th ISAAC*, pages 173–182, 2009.

[BFK+23] Hideo Bannai, Mitsuru Funakoshi, Kazuhiro Kurita, Yuto Nakashima, Kazuhisa Seto, and Takeaki Uno. Optimal LZ-end parsing is hard. *CoRR*, abs/2302.02586, 2023. To appear at CPM 2023.

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[BG11] Philip Bille and Inge Li Gørtz. The tree inclusion problem: In linear space and faster. *ACM Trans. Algorithms*, 7(3):1–47, 2011.

[BG14] Philip Bille and Inge Li Gørtz. Substring range reporting. *Algorithmica*, 69(2):384–396, 2014.

[BGP16] Johannes Bader, Simon Gog, and Matthias Petri. Practical variable length gap pattern matching. In *Proc. 15th SEA*, pages 1–16, 2016.

[BGP+20a] Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, Eva Rotenberg, and Teresa Anna Steiner. String Indexing for Top-k Close Consecutive Occurrences. In *Proc. 40th FSTTCS*, volume 182, pages 14:1–14:17, 2020.

[BGP20b] Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *Proc. SODA 2020*, pages 607–626, 2020.

[BGPS21] Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Teresa Anna Steiner. Gapped indexing for consecutive occurrences. In *Proc. 32nd CPM*, pages 10:1–10:19, 2021.

[BGPT23] Philip Bille, Inge Li Gørtz, Simon J. Puglisi, and Simon R. Tarnow. Hierarchical relative lempel-ziv compression. In *Proc. 21st SEA*, 2023.

[BGS17] Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic Indexing for Packed Strings. In *Proc. 28th CPM*, volume 78, pages 6:1–6:11, 2017.

[BGST18] Sudip Biswas, Arnab Ganguly, Rahul Shah, and Sharma V Thankachan. Ranked document retrieval for multiple patterns. *Theor. Comput. Sci.*, 746:98–111, 2018.

[BGVV14] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014.

[BGVW12] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind. String matching with variable length gaps. *Theoret. Comput. Sci.*, 443, 2012. Announced at SPIRE 2010.

[BHK+99] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. UMAC: fast and secure message authentication. In *Proc. 19th CRYPTO*, volume 1666, pages 216–233, 1999.

[BI13] Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013.

[BJ18] Andrej Brodnik and Matevz Jekovec. Sliding suffix tree. *Algorithms*, 11(8):118, 2018.

[BP16] Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th SODA*, pages 2053–2071, 2016.

[BTC06] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *Proc. 33rd ISCA*, pages 191–202, 2006.

[Cha02] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. 34th STOC*, pages 380–388, 2002.

[Cha13] Timothy M Chan. Persistent predecessor search and orthogonal point location on the word ram. *ACM Trans. Algorithms*, 9(3):1–22, 2013.

[CKL15] Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix Trays and Suffix Trists: Structures for Faster Text Indexing. *Algorithmica*, 72(2):450–466, 2015.

[CLM13] Maxime Crochemore, Alessio Langiu, and Filippo Mignosi. The rightmost equal-cost position problem. In *Proc. DCC 2013*, pages 421–430, 2013.

[CP09] Hagai Cohen and Ely Porat. Range non-overlapping indexing. In *Proc. 20th ISAAC*, pages 1044–1053, 2009.

[CP10] Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theor. Comput. Sci.*, 411(40-42):3795–3800, 2010.

[CPZ20] Manuel Cáceres, Simon J Puglisi, and Bella Zhukova. Fast indexes for gapped pattern matching. In *Proc. 46th SOFSEM*, pages 493–504, 2020.

[CR91] Maxime Crochemore and Wojciech Rytter. Efficient parallel algorithms to test square-freeness and factorize strings. *Inf. Process. Lett.*, 38(2):57–60, 1991.

[CT18] Timothy M. Chan and Konstantinos Tsakalidis. Dynamic orthogonal range searching on the ram, revisited. *J. Comput. Geom.*, 9(2):45–66, 2018.

[CW79] Larry Carter and Mark N. Wegman. Universal Classes of Hash Functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[DadH90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A New Universal Class of Hash Functions and Dynamic Hashing in Real Time. In *Proc. 17th ICALP*, pages 6–19, 1990.

[DAS10] Shuai Ding, Josh Attenberg, and Torsten Suel. Scalable techniques for document identifier assignment in inverted indexes. In *Proc. 19th WWW*, pages 311–320, 2010.

[DI03] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *Proc. USENIX ATC, General Track 2003*, pages 113–126, 2003.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[FFM00] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[FG89] Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989.

[FG05] Johannes Fischer and Pawel Gawrychowski. Alphabet-Dependent String Searching with Wexponential Search Trees. In *Proc. 26th CPM*, pages 160–171, 2005.

[FG08] Kimmo Fredriksson and Szymon Grabowski. Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. *Inf. Retr.*, 11(4):335–357, 2008.

[FGP+08] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved DFA for fast regular expression matching. *Comput. Commun. Rev.*, 38(5):29–40, 2008.

[FHP81] Arthur M. Farley, Stephen T. Hedetniemi, and Andrzej Proskurowski. Partitioning trees: Matching, domination, and maximum diameter. *Int. J. Parallel Program.*, 10(1):55–61, 1981.

[FIK15] Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel Ziv computation in small space (LZ-CISS). In *Proc 26th CPM*, pages 172–184, 2015.

[FIKS18] Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiko Sadakane. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018.

[FKMS03] Paolo Ferragina, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Two-dimensional substring indexing. *J. Comput. Syst. Sci.*, 66(4):763–774, 2003.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[FM95] Martin Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression (extended abstract). In *Proc. 7th SPAA*, page 244–253, 1995.

[FNV13] Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of Lempel-Ziv compression. *SIAM J. Comput.*, 42(4):1521–1541, 2013.

[FPG+11] Domenico Ficara, Andrea Di Pietro, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, and Gianni Antichi. Differential encoding of dfas for fast regular expression matching. *IEEE/ACM Trans. Netw.*, 19(3):683–694, 2011.

[Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[Fre97] Greg N Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997.

[GB13] Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In *Proc. DCC 2013*, pages 133–142, 2013.

[GB14] Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Proc. DCC 2014*, pages 163–172, 2014.

[GBT84] Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th STOC*, pages 135–143. ACM, 1984.

[GKLP17] Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In *Proc. 15th WADS*, pages 421–436, 2017.

[GST20] Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Succinct non-overlapping indexing. *Algorithmica*, 82(1):107–117, 2020.

[GWX+23] Lei Gong, Chao Wang, Haojun Xia, Xianglan Chen, Xi Li, and Xuehai Zhou. Enabling fast and memory-efficient acceleration for pattern matching workloads: The lightweight automata processing engine. *IEEE Trans. Computers*, 72(4):1011–1025, 2023.

[Hag98] Torben Hagerup. Sorting and searching on the word ram. In *Proc. 15th STACS*, pages 366–398, 1998.

[HAKT18] Sahar Hooshmand, Paniz Abedin, M. Oguzhan Külekci, and Sharma V. Thankachan. Non-overlapping indexing - cache obliviously. In *Proc. 29th CPM*, pages 8:1–8:9, 2018.

[Han02] Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. In *Proc. 34th STOC*, pages 602–608, 2002.

[HBFB99] K Hofmann, P Bucher, L Falquet, and A Bairoch. The PROSITE database, its status in 1999. *Nucleic Acids Res*, 27(1):215–219, 1999.

[HIM12] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory Comput.*, 8(1):321–350, 2012.

[HPS+13] Wing-Kai Hon, Manish Patil, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Indexes for document retrieval with relevance. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 351–362, 2013.

[HPSW10] Wing-Kai Hon, Manish Patil, Rahul Shah, and Shih-Bin Wu. Efficient index for retrieving top-k most frequent documents. *J. Discrete Algorithms*, 8(4):402–417, 2010.

[HSSSS11] Tuukka Haapasalo, Panu Silvasti, Seppo Sippu, and Eljas Soisalon-Soininen. Online dictionary matching with variable-length gaps. In *Proc. 10th SEA*, pages 76–87, 2011.

[HSTV14] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *J. ACM*, 61(2):1–36, 2014. Announced at 50th FOCS.

[HTSV13] Wing-Kai Hon, Sharma V. Thankachan, Rahul Shah, and Jeffrey Scott Vitter. Faster compressed top-k document retrieval. In *Proc. 23rd DCC*, pages 341–350, 2013.

[IR09] Costas S Iliopoulos and M Sohel Rahman. Indexing factors with gaps. *Algorithmica*, 55(1):60–70, 2009.

[ISTA04] Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compact directed acyclic word graphs for a sliding window. *J. Discrete Algorithms*, 2(1):33–51, 2004.

[KDLT04] Purushottam Kulkarni, Fred Douglis, Jason D. LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proc. USENIX ATC, General Track 2004*, pages 59–72, 2004.

[KDY+06] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan S. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. SIGCOMM 2006*, pages 339–350, 2006.

[KH15] Lubos Krcál and Jan Holub. Incremental locality and clustering-based compression. In *DCC 2015*, pages 203–212, 2015.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[KK16] Tsvi Kopelowitz and Robert Krauthgamer. Color-distance oracles and snippets. In *Proc. 27th CPM*, pages 24:1–24:10, 2016.

[KK17a] Dominik Kempa and Dmitry Kosolobov. LZ-end parsing in compressed space. In *Proc. DCC 2017*, pages 350–359, 2017.

[KK17b] Dominik Kempa and Dmitry Kosolobov. LZ-end parsing in linear time. In *Proc. 25th ESA*, volume 87, pages 53:1–53:14, 2017.

[KK19] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proc. 51st STOC*, pages 756–767, 2019.

[KK22] Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proc. 54th STOC*, pages 1657–1670, 2022.

[KKL07] Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. Range non-overlapping indexing and successive list indexing. In *Proc. 11th WADS*, pages 625–636, 2007.

[KKP13a] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. 12th SEA*, pages 139–150, 2013.

[KKP13b] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. 24th CPM*, pages 189–200, 2013.

[KKP14] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. DCC 2014*, pages 153–162, 2014.

[KLA+01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th CPM*, volume 2089, pages 181–192, 2001.

[KN10] Sebastian Kreft and Gonzalo Navarro. LZ77-like compression with fast random access. In *Proc. DCC 2010*, pages 239–248, 2010.

[KN13] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013.

[KN17] Gregory Kucherov and Yakov Nekrich. Full-Fledged Real-Time Indexing for Constant Size Alphabets. *Algorithmica*, 79(2):387–400, 2017.

[Koc19] Tomasz Kociumaka. *Efficient data structures for internal queries in texts*. PhD thesis, University of Warsaw, 2019.

[Kop12] Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *53rd FOCS*, pages 283–292, 2012.

[Kos94] S. Rao Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In *Proc. 26th STOC*, pages 310–316, 1994.

[Kos15] Dmitry Kosolobov. Faster lightweight Lempel-Ziv parsing. In *Proc. 40th MFCS*, pages 432–444, 2015.

[KPP16] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proc. 27th SODA*, pages 1272–1287, 2016.

[Kru56] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7(1):48–50, 1956.

[KS16] Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC 2016*, pages 3–12, 2016.

[KS22] Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-end parsing. In *Proc. SODA 2022*, pages 2847–2866, 2022.

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[KSE08] Shijin Kong, Randy Smith, and Cristian Estan. Efficient signature matching with multiple alphabet compression tables. In *Proc. 4th SECURECOMM*, page 1, 2008.

[KTW06] Sailesh Kumar, Jonathan S. Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proc. ANCS 2006*, pages 81–92, 2006.

[Lar99] N. Jesper Larsson. *Structures of String Matching and Data Compression.* PhD thesis, Lund University, Sweden, 1999.

[Lar14] N. Jesper Larsson. Most recent match queries in on-line suffix trees. In *Proc. 25th CPM*, volume 8486, pages 252–261, 2014.

[Lew11] Moshe Lewenstein. Indexing with gaps. In *Proc. 18th SPIRE*, pages 135–143, 2011.

[LMNT15] Kasper Green Larsen, J Ian Munro, Jesper Sindahl Nielsen, and Sharma V Thankachan. On hardness of several string indexing problems. *Theoret. Comput. Sci.*, 582:74–82, 2015.

[LSL+17] Shu Liu, Shaojing Su, Desheng Liu, Zhiping Huang, and Mingyan Xiao. Efficient compression algorithm for ternary content addressable memory-based regular expression matching. *Electronics Letters*, 53(3):152–154, 2017.

[LT14] Alex X. Liu and Eric Torng. An overlay automata approach to regular expression matching. In *Proc. 33rd INFOCOM*, pages 952–960, 2014.

[LZ76] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theory*, 22(1):75–81, 1976.

[MKMK18] Denis Matousek, Juraj Kubis, Jirí Matousek, and Jan Korenek. Regular expression matching with pipelined delayed input dfas for high-speed networks. In *Proc. ANCS 2018*, pages 104–110, 2018.

[MM93a] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[MM93b] Gerhard Mehldau and Gene Myers. A system for pattern matching applications on biosequences. *Bioinformatics*, 9(3):299–314, 1993.

[MMK18] Denis Matousek, Jirí Matousek, and Jan Korenek. High-speed regular expression matching with pipelined memory-based automata. In *Proc. 26th FCCM*, page 214, 2018.

[MNN+17] J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top-k term-proximity in succinct space. *Algorithmica*, 78(2):379–393, 2017. Announced at 25th ISAAC.

[MNST20] J. Ian Munro, Gonzalo Navarro, Rahul Shah, and Sharma V. Thankachan. Ranked document selection. *Theor. Comput. Sci.*, 812:149–159, 2020.

[MPN+10] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In *19th USENIX Security*, pages 111–126, 2010.

[Mye92] Eugene W. Myers. Approximate matching of network expressions with spacers. *J. Comput. Bio.*, 3(1):33–51, 1992.

[NAIP03] Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 304(1-3):87–101, 2003.

[Nao91] Moni Naor. String matching with preprocessing of text and pattern. In *Proc. 18th ICALP*, pages 739–750, 1991.

[Nav14] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):1–47, 2014.

[Nav16] Gonzalo Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016.

[NN12] Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *Proc 13th SWAT*, pages 271–282, 2012.

[NN17] Gonzalo Navarro and Yakov Nekrich. Time-optimal top-k document retrieval. *SIAM J. Comput.*, 46(1):80–113, 2017. Announced at 23rd SODA.

[NR03] Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *J. Comput. Bio.*, 10(6):903–923, 2003.

[NT14] Gonzalo Navarro and Sharma V. Thankachan. New space/time tradeoffs for top-k document retrieval on sequences. *Theor. Comput. Sci.*, 542:83–97, 2014. Announced at 20th SPIRE.

[NT16] Gonzalo Navarro and Sharma V. Thankachan. Reporting consecutive substring occurrences under bounded gap constraints. *Theor. Comput. Sci.*, 638:108–111, 2016. Announced at 26th CPM.

[OCGO96a] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[OCGO96b] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[OG11] Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Proc. 22nd CPM*, pages 15–26, 2011.

[OMST02] Zan Ouyang, Nasir D. Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *Proc. 3rd WISE*, pages 257–268, 2002.

[OS08] Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In *Proc. 16th ESA*, pages 696–707, 2008.

[PLT14] Jignesh Patel, Alex X. Liu, and Eric Torng. Bypassing space explosion in high-speed regular expression matching. *IEEE/ACM Trans. Netw.*, 22(6):1701–1714, 2014.

[Pri57] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[PS17] S Prithi and S Sumathi. A survey on recent dfa compression techniques for deep packet inspection in network intrusion detection system. *Journal of Electrical Engineering*, 17(3):14–14, 2017.

[PWZ11] Andrew Peel, Anthony Wirth, and Justin Zobel. Collection-based compression using discovered long matching strings. In *Proc. 20th CIKM*, pages 2361–2364, 2011.

[QWF+11] Yaxuan Qi, Kai Wang, Jeffrey Fong, Yibo Xue, Jun Li, Weirong Jiang, and Viktor K. Prasanna. FEACAN: front-end acceleration for content-aware network processing. In *Proc. 30th INFOCOM*, pages 2114–2122, 2011.

[RHH+04] Michael Roberts, Wayne B. Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinform.*, 20(18):3363–3369, 2004.

[Rou10] Vassil Roussev. Data fingerprinting with similarity digests. In *IFIP Int. Conf. Digital Forensics 2010*, volume 337, pages 207–226, 2010.

[SD08] Martin Senft and Tomás Dvorák. Sliding CDAWG perfection. In *Proc. 15th SPIRE*, pages 109–120, 2008.

[Sen05] M Senft. Suffix tree for a sliding window: An overview. In *Proc. WDS*, volume 5, pages 41–46, 2005.

[Shu18] Julian Shun. *Parallel Lempel-Ziv Factorization*, chapter 13. Association for Computing Machinery and Morgan & Claypool, 2018.

[SHWH12] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. Wan-optimized replication of backup datasets using stream-informed delta compression. *ACM Trans. Storage*, 8(4):13:1–13:26, 2012.

[SLHW17] Subramanian Shiva Shankar, Pinxing Lin, Andreas Herkersdorf, and Thomas Wild. A divide and conquer state grouping method for bitmap based transition compression. In *Proc. 18th PDCAT*, pages 400–406, 2017.

[SLLM10] Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic extended suffix arrays. *J. Discrete Algorithms*, 8(2):241–257, 2010.

[SS82] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[SSTV13] Rahul Shah, Cheng Sheng, Sharma V. Thankachan, and Jeffrey Scott Vitter. Top-k document retrieval in external memory. In *Proc. 21st ESA*, pages 803–814, 2013.

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[Sta12] Tatiana Starikovskaya. Computing Lempel-Ziv factorization online. In *Proc. 37th MFCS*, pages 789–799, 2012.

[SWA03] Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. SIGMOD 2003*, pages 76–85, 2003.

[SZ13] Julian Shun and Fuyao Zhao. Practical parallel Lempel-Ziv factorization. In *Proc. DCC 2013*, pages 123–132, 2013.

[TJD+17] Qiu Tang, Lei Jiang, Qiong Dai, Majing Su, Hongtao Xie, and Binxing Fang. RICS-DFA: a space and time-efficient signature matching algorithm with reduced input character set. *Concurr. Comput. Pract. Exp.*, 29(20), 2017.

[TSCV04] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proc. 23rd INFOCOM*, pages 2628–2639, 2004.

[Tsu13] Dekel Tsur. Top-k document retrieval in optimal space. *Inf. Process. Lett.*, 113(12):440–443, 2013.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th SWAT*, pages 1–11, 1973.

[XCS+16] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas Chi Kwong Hui. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Commun. Surv. Tutorials*, 18(4):2991–3029, 2016.

[XJFH11] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Silo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *USENIX ATC 2011*, 2011.

[YIB+14] Jun'ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster Compact On-Line Lempel-Ziv Factorization. In *Proc. 31st STACS*, pages 675–686, 2014.

[Zho16] Gelin Zhou. Two-dimensional range successor in optimal time and almost linear space. *Inf. Process. Lett.*, 116(2):171–174, 2016.