2015

String Searching with Ranking Constraints and Uncertainty

Sudip Biswas
Louisiana State University and Agricultural and Mechanical College

STRING SEARCHING WITH RANKING CONSTRAINTS AND UNCERTAINTY

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Sudip Biswas
B.Sc., Bangladesh University of Engineering and Technology, 2009
December 2015

Dedicated to my parents, Purabi Chowdhury and Babul Chandra Biswas.

Acknowledgments

I am grateful to my supervisor Dr. Rahul Shah for giving the opportunity to fulfill my dream of pursuing Ph.D in computer science. His advice, ideas and feedback helped me understand the difficult concepts and get better as a student. Without his continuous support and guidance, I wouldn't have come so far in my studies and research. His algorithm course, seminars and discussions were very effective and motivating.

I had the privilege to collaborate with some strong researchers in this field. My labmates Sharma Thankachan and Manish Patil helped me developing better background in research and played significant role in my publications. My friend and former classmate Debajyoti Mondal has always been a great inspiration for me. He was always there to help me when I had difficult time in my studies and research. In addition to these people, I am thankful to my fellow Ph.D. student Arnab Ganguly for collaboration.

A special thanks to Dr. Sukhamay Kundu. I admired and learned from his devotion and integrity in research.

Finally, I would like to thank my thesis committee members Dr. Jianhua Chen, Dr. Konstantin Busch, and Dr. Charles Monlezun for their many constructive feedback.

Table of Contents

List of Figures

Abstract

Strings play an important role in many areas of computer science. Searching pattern in a string or string collection is one of the most classic problems. Enormous growth of internet, large genomic projects, sensor networks, digital libraries necessitates not just efficient algorithms and data structures for the general string indexing, but indexes for texts with fuzzy information and support for constrained queries. This dissertation addresses some of these problems and proposes indexing solutions.

One such variation is document retrieval query for included and forbidden patterns We propose indexing solution for this problem and conjecture that any significant improvement over these results is highly unlikely. We also consider the scenario when the query consists of more than two patterns. Continuing this path, we introduce document retrieval with forbidden extension query, where the forbidden pattern is an extension of the included pattern. We achieve linear space and optimal query time for this variation. We also propose succinct indexes for both these problems.

Position restricted pattern matching considers the scenario where only part of the text is searched. We propose succinct index for this problem with efficient query time. An important application for this problem stems from searching in partial genomic sequences.

Computing discriminating(resp. generic) words is to report all minimal(resp. maximal) extensions of a query pattern which are contained in at most(resp. at least) a given number of documents. These problems are motivated from applications in computational biology, text mining and automated text classification. We propose succinct indexes for these problems.

Strings with uncertainty and fuzzy information play an important role in increasingly many applications. We propose a general framework for indexing uncertain strings.

We also discuss a constrained variation of orthogonal range searching. Shared constraint range searching is a special four sided range reporting query problem where two constraints has sharing among them. We propose a linear space index that can match the best known bound for three dimensional dominance reporting problem. We extend our data structure in the external memory model.

Chapter 1
Introduction

1.1 Overview and Motivation

Text indexing and searching is a well studied branch in Computer Science. We start by looking at two fundamental problems in this domain, namely pattern matching and document listing.

Let $T[0...n-1]$ be a text of size n over an alphabet set Σ of size σ. The pattern matching problem by text indexing is to preprocess T and maintain an index for reporting all occ occurrences of a query pattern P within T. Linear space data structures such as suffix trees and suffix arrays can answer this query in $O(p + occ)$ and $O(p + \log n + occ)$ time respectively [116, 89, 87].

Most string databases consist of a collection of strings (or documents) rather than just one single string. We shall use $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ for denoting the string collection of D strings of n characters in total. In this case, a natural problem is to preprocess $\mathcal{D}$ and maintain it as a data structure, so that, whenever a pattern $P[1 \ldots p]$ comes as a query, those documents where P occurs at least once can be reported efficiently, instead of reporting all the occurrences. This is known as the *document listing* problem. A more generalized variation is to report k documents most relevant to the query pattern, based on relevance metrics.

However, different applications demand more constrained approach. For example, restricting the search to a subset of dynamically chosen documents in a document database and restricting the search to only parts of a long DNA sequence, retrieving most interesting documents based on multiple query patterns. We revisit the well studied text indexing problem involving different constraints and uncertainty. A closely related problem is orthogonal range reporting. We formulate a new orthogonal range reporting problem which finds motivation in constrained pattern matching applications. We propose indexing solution for uncertain strings, which have become increasingly more prevalent due to unreliability of source, imprecise measurement, data loss, and artificial noise. Below we introduce the problems discussed in this thesis.

1.1.1 Ranked Document Retrieval with Multiple Patterns

We consider the problem of retrieving the top-k documents when the input query consists of multiple patterns, under some standard relevance functions such as *document importance*, *term-frequency*, and *term-proximity*.

Let $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ be a collection of D string documents of n characters in total. Given two patterns P and Q, and an integer $k > 0$, we consider the following queries.

- top-k *forbidden pattern query:* Among all the documents that contain P, but not Q, the task is to report those k documents that are most relevant to P.
- top-k *two pattern query:* Among all the documents that contain both P and Q, the task is to report those k documents that are most relevant to P.

For each of the above two queries, we provide a linear space index with $O(|P| + |Q| + \sqrt{nk})$ query time. The document listing version of the above two problems asks to report all t documents that either contains P, but not Q, or contains both P and Q, depending on the query type. As a corollary of the top-k result, we obtain a linear space and $O(|P| + |Q| + \sqrt{nt})$ query time solution for the document listing problem. We conjecture that any significant improvement over these results is highly unlikely. We also consider the scenario when the query consists of more than two patterns. Finally, we present a space efficient index for these problems.

1.1.2 Document Retrieval with Forbidden Extensions

We introduce the problem of document retrieval with forbidden extensions.

Let $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ be a collection of D string documents of n characters in total, and P^+ and P^- be two query patterns, where P^+ is a proper prefix of P^-. We call P^- as the forbidden extension of the included pattern P^+. A forbidden extension query $\langle P^+, P^- \rangle$ asks to report all occ documents in $\mathcal{D}$ that contains P^+ as a substring, but does not contain P^- as one. A top-k forbidden extension query $\langle P^+, P^-, k \rangle$ asks to report those k documents among the occ documents that are most relevant to P^+. We present a linear index (in words) with an $O(|P^-| + occ)$ query time for the document listing problem. For the top-k version of the problem, we achieve the following results, when the relevance of a document is based on PageRank:

- an $O(n)$ space (in words) index with optimal $O(|P^-| + k)$ query time.
- for any constant $\epsilon > 0$, a $|\mathsf{CSA}| + |\mathsf{CSA}^*| + D \log \frac{n}{D} + O(n)$ bits index with $O(\mathsf{search}(P) + k \cdot \mathsf{t}_{\mathsf{SA}} \cdot \log^{2+\epsilon} n)$ query time, where $\mathsf{search}(P)$ is the time to find the suffix range of a pattern P, t_{SA} is the time to find suffix (or inverse suffix) array value, and $|\mathsf{CSA}^*|$ denotes the maximum of the space needed to store the *compressed suffix array* CSA of the concatenated text of all documents, or the total space needed to store the individual CSA of each document.

1.1.3 Document Retrieval using Shared Constraint Range Reporting

Orthogonal range searching is a classic problem in computational geometry and database. Motivated by document retrieval, searching queries with constraint sharing and several well known text indexing problems, we study a special four sided range reporting query problem, which we call as the *Shared-Constraint Range Reporting* (SCRR) problem. Given a set $\mathcal{P}$ of N three dimensional points, the query input is a triplet (a, b, c), and our task is to report all those points within a region $[a, b] \times (-\infty, a] \times [c, \infty)$. We can report points within any region $[a, b] \times (-\infty, f(a)] \times [c, \infty)$, where $f(\cdot)$ is a pre-defined monotonic function (using a simple transformation). The query is four sided with only three independent constraints. Many applications which model their formulation as 4-sided problems actually have this sharing among the constraints and hence better bounds can be obtained for them using SCRR data structures. Formally, we have the following definition.

> ***SCRR problem:*** A SCRR query $Q_{\mathcal{P}}(a, b, c)$ on a set $\mathcal{P}$ of three dimensional points asks to report all those points within the region $[a, b] \times (-\infty, a] \times [c, \infty)$.

We propose a linear space and optimal time index for this problem. We extend our index into the external memory model.

1.1.4 Minimal discriminating words and maximal Generic Words

Computing minimal discriminating words and maximal generic words stems from computational biology applications. An interesting problem in computational bilogy is to identify words that are exclusive to the genomic sequences of one species or family of species [42]. Such patterns that appear in a small set of biologically related DNA sequences but do not appear in other sequences in the collection, often carries a biological significance. Computing maximal generic word problem finds all the maximal extensions of a query pattern that occurs in atleast a given number of documents, whereas computing minimal discriminating word problem finds all the minimal extensions of a query pattern that occurs in atmost a given number of documents. Below we describe these problems more formally.

Let $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ be a collection of D strings (which we call as documents) of total n characters from an alphabet set Σ of size σ. For simplicity we assume, every document ends with a special character \$ which does not appear any where else in the documents. Our task is to index $\mathcal{D}$ in order to compute all (i) *maximal generic words* and (ii) *minimal discriminating words* corresponding to a query pattern P (of

length p). The document frequency $df(.)$ of a pattern P is defined as the number of distinct documents in $\mathcal{D}$ containing P. Then, a generic word is an extension $\bar{P}$ of P with $df(\bar{P}) \geq \tau$, and is maximal if $df(P') < \tau$ for all extensions P' of $\bar{P}$. Similarly, a discriminating word is an extension $\bar{P}$ of P with $df(\bar{P}) \leq \tau$, and is called a minimal discriminating word if $df(P') > \tau$ for any proper prefix P' of $\bar{P}$ (i.e., $P' \neq \bar{P}$). These problems were introduced by Kucherov et al. [77], and they proposed indexes of size $O(n \log n)$ bits or $O(n)$ words. The query processing time is optimal $O(p + output)$ for reporting all maximal generic words, and is near optimal $O(p + \log\log n + output)$ for reporting all minimal discriminating words. We describe succinct indexes of $n \log \sigma + o(n \log \sigma) + O(n)$ bits space with $O(p + \log\log n + output)$ query times for both these problems.

Identification of genomic markers, or probe design for DNA microarrays are also closely related problems. Discriminating and generic words also find applications in text mining and automated text classification.

1.1.5 Position-restricted Substring Searching

We revisit the well studied *Position-restricted substring searching* (PRSS) problem as defined below:

> ***PRSS problem:*** The query input consists of a pattern P (of length p) and two indices ℓ and r, and the task is to report all $occ_{\ell,r}$ occurrences of P in $T[\ell...r]$.

Many text searching applications, where the objective is to search only a part of the text collection can be modeled as PRSS problem. For example, restricting the search to a subset of dynamically chosen documents in a document database, restricting the search to only parts of a long DNA sequence, etc [85]. The problem also finds applications in the field of information retrieval as well.

We introduce a space efficient index for this problem. Our index takes $O(n \log \sigma)$-words and supports PRSS queries in $O(p + occ_{\ell,r} \log\log n)$ time. For smaller alphabets, the space saving becomes significant.

1.1.6 Uncertain String Searching

String matching becomes a probabilistic event when a string T contains uncertainty and fuzzy information, i.e. each position of T can have different probable characters with associated probability of occurrence. An uncertain string $T = t_1 t_2 ... t_n$ over alphabet Σ is a sequence of sets $t_i, i = 1, 2, ..., n$. Every t_i is a set of pairs $(s_j, pr(s_j))$, where every s_j is a character in Σ and $0 \leq pr(s_j) \leq 1$ is the probability of occurrence of s_j at position i in the text. Uncertain string is also known as probabilistic string or weighted string. Note that, at

each position, summation of probability for all the characters at each position should be 1, i.e. $\sum_j pr(s_j) = 1$. We explore the problem of indexing uncertain strings for efficient searching. Given a collection of uncertain strings $\mathcal{D}$=$D_1, D_2, ..., D_{|\mathcal{D}|}$ of total length n, our goal is to report all the strings containing a certain or deterministic query string p with probability more than or equal to an probability threshold τ. We also discuss the problem of searching a deterministic query string p within a single uncertain string S of length n.

1.2 Model of Computation

Unless explicitly mentioned, the model of computation in this thesis is the RAM model. In RAM model, random access of any memory cell and basic arithmetic operations can be performed in constant time. We measure the run time of an algorithm by counting up the number of steps it takes on a given problem instance. We assume that, the memory is partitioned into continuous blocks of $\Theta(\log n)$ size, where n denotes the input problem size.

In external-memory model, performance of an algorithm is measured by the number of I/O operations used. Since internal memory (RAM) is much faster compared to the slow external disk, operations performed in memory are considered free. The disk is divided into blocks of size B. The space of a structure is the number of blocks occupied. The CPU can only operate on data inside the internal memory. So, we need to transfer data between internal memory and disk through I/O operations, where each I/O may transfer a block from the disk to the memory (or vice versa). An efficient algorithm in the external-memory model aims to minimize the I/O transfer cost.

1.3 Outline

Each chapter of this thesis is self-contained. However the key concepts and techniques used are closely related. In Chapter 1.3, we discuss standard data-structures, and introduce the terminologies used in this paper. We describe the ranked document retrieval with forbidden pattern in Chapter 2.7. A version of this work appeared in [16]. In Chapter 3.8, we discuss the document retrieval with forbidden extension index. Next, we present a succinct index for the forbidden extension problem in Chapter 4.2. Chapter 7.4 is dedicated to the Position-restricted substring searching problem. A version of this work appeared in [18]. Indexes for computing discriminating and generic words are presented in Chapter 6.7.2. A version of this work appeared in [19]. In Chapter 8.2, we introduce uncertain string indexes. Chapter 4.2 presents the index for

shared constraint range reporting problem. A version of this work appeared in [21]. Finally, we conclude in Chapter 9.8.7 with a brief summary and future work direction.

Chapter 2
Preliminaries

In this section we discuss some common data structures, tools and notations used in this thesis. Data structures and definitions that pertain only to a single chapter of this work are defined within that chapter.

2.1 Suffix trees and Generalized Suffix Trees

The best known data structure for text indexing is the suffix tree [89]. A suffix tree considers each position in the text as a suffix, i.e., a string extending from that position to the end of the text. Each suffix is unique and can be identified by its starting position. For a text $S[1...n]$, a substring $S[i...n]$ with $i \in [1, n]$ is called a suffix of T. The suffix tree [116, 89] of S is a lexicographic arrangement of all these n suffixes in a compact trie structure of $O(n)$ words space, where the i-th leftmost leaf represents the i-th lexicographically smallest suffix of S. For a node i (i.e., node with pre-order rank i), $path(i)$ represents the text obtained by concatenating all edge labels on the path from root to node i in a suffix tree. The locus node i_P of a pattern P is the node closest to the root such that the P is a prefix of $path(i_P)$. The suffix range of a pattern P is given by the maximal range $[sp, ep]$ such that for $sp \leq j \leq ep$, P is a prefix of (lexicographically) j-th suffix of S. Therefore, i_P is the lowest common ancestor of sp-th and ep-th leaves. Using suffix tree, the locus node as well as the suffix range of P can be computed in $O(p)$ time, where p denotes the length of P.

The generalized suffix tree works for a collection of text documents. Let $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ be a collection of D strings (which we call as documents) of total n characters. Let $T = \mathsf{T}_1\mathsf{T}_2...\mathsf{T}_D$ be the text obtained by concatenating all documents in $\mathcal{D}$. Recall that each document is assumed to end with a special character \$. The suffix tree of T is called the generalized suffix tree (GST) of $\mathcal{D}$.

2.2 Suffix Array

The *suffix array* $\mathsf{SA}[1..n]$ is an array of length n, where $\mathsf{SA}[i]$ is the starting position (in T) of the ith lexicographically smallest suffix of T [86]. In essence, the suffix array contains the leaf information of GST but without the tree structure. An important property of SA is that the starting positions of all the suffixes with the same prefix are always stored in a contiguous region of SA. Based on this property, we define the *suffix range* of P in SA to be the maximal range $[sp, ep]$ such that for all $i \in [sp, ep]$, $\mathsf{SA}[i]$ is the starting

point of a suffix of T prefixed by P. Therefore, ℓ_{sp} and ℓ_{ep} represents the first and last leaves in the subtree of the locus node of P in GST.

2.3 Document Array

The document array annotates each leaf of GST by a document identifier. The document array $E\,[1..n]$ is defined as $E\,[j] = r$ if the suffix $\mathsf{T}[\mathsf{SA}[j]..n]$ belongs to document T_r. Moreover, the corresponding leaf node ℓ_j is said to be *marked* with document T_r.

2.4 Bit Vectors with Rank/Select Support

Let $\mathsf{B}[1..n]$ be a bit vector with its m bits set to 1. Then, $rank_{\mathsf{B}}(i)$ represents the number of 1's in $\mathsf{B}[1..i]$ and $select_{\mathsf{B}}(j)$ represents the position in B where the jth 1 occurs (if $j > m$, return NIL). The minimum space needed for representing B is given by $\lceil \log \binom{n}{m} \rceil \leq m \log(ne/m) = m \log(n/m) + 1.44m$ [103]. There exists representations of B in $n + o(n)$ bits and $m \log(n/m) + O(m) + o(n)$ bits, which can support both $rank_{\mathsf{B}}(\cdot)$ and $select_{\mathsf{B}}(\cdot)$ operations in constant time. These structures are known as fully indexable dictionaries.

2.5 Succinct Representation of Ordinal Trees

The lower bound on the space needed for representing any n-node ordered rooted tree, where each node is labeled by its preorder rank in the tree, is $2n - O(\log n)$ bits. Using succinct data structure occupying $o(n)$ bits extra space, the following operations can be supported in constant time [108]: (i) $parent(u)$, which returns the parent of node u, (ii) $lca(u, v)$, which returns the lowest common ancestor of two nodes u and v, and (iii) $lmost\_leaf(u)/rmost\_leaf(u)$, which returns the leftmost/rightmost leaf of node u.

2.6 Marking Scheme in GST

We briefly explain the marking scheme introduced by Hon et al. [66] which will be used later in the proposed succinct index. We identify certain nodes in the GST as marked nodes and prime nodes with respect to a parameter g called the *grouping factor*. The procedure starts by combining every g consecutive leaves (from left to right) together as a group, and marking the lowest common ancestor (LCA) of first and last leaf in each group. Further, we mark the LCA of all pairs of marked nodes recursively. We also ensure that the root is always marked. At the end of this procedure, the number of marked nodes in GST will be $O(n/g)$. Hon et al. [66] showed that, given any node u with u^* being its highest marked descendent (if

exists), number of leaves in $GST(u \backslash u^*)$ i.e., the number of leaves in the subtree of u, but not in the subtree of u^* is at most $2g$.

Prime nodes are the children of marked nodes. Corresponding to any marked node u^* (except the root node), there is a *unique prime* node u', which is its closest prime ancestor. In case u^*'s parent is marked then $u' = u^*$. For every prime node u', the corresponding closest marked descendant u^* (if it exists) is unique.

2.7 Range Maximum Query

A Range maximum query (RMQ) solves the problem of finding the maximum value in a sub-array of an array of numbers. Let A be an array containing n numbers, a range maximum query(RMQ) asks for the position of the maximum value between two specified array indices $[i, j]$. i.e., the RMQ should return an index k such that $i \leq k \leq j$ and $A[k] \geq A[x]$ for all $i \leq x \leq j$. We use the result captured in following lemma for our purpose.

Lemma 1. [46, 47] By maintaining a $2n+o(n)$ bits structure, range maximum query(RMQ) can be answered in $O(1)$ time (without accessing the array).

Chapter 3
Ranked Document Retrieval with Multiple Patterns

3.1 Overview

Document retrieval is a fundamental problem in *Information Retrieval*. Given a collection of strings (called documents), the task is to index them so that whenever a pattern comes as query, we can report all the documents that contain the pattern as a substring. Many variations of this problem have been considered, such as the top-k document retrieval problem, document retrieval for multiple patterns and for forbidden patterns. The top-k document retrieval problem returns those k documents which are most relevant to an input pattern. In this work, we consider the problem of retrieving the top-k documents when the input query consists of multiple patterns, under some standard relevance functions such as *document importance*, *term-frequency*, and *term-proximity*.

Let $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ be a collection of D string documents of n characters in total. Given two patterns P and Q, and an integer $k > 0$, we consider the following queries.

- top-k *forbidden pattern query:* Among all the documents that contain P, but not Q, the task is to report those k documents that are most relevant to P.

- top-k *two pattern query:* Among all the documents that contain both P and Q, the task is to report those k documents that are most relevant to P.

For each of the above two queries, we provide a linear space index with $O(|P| + |Q| + \sqrt{nk})$ query time. The document listing version of the above two problems asks to report all t documents that either contains P, but not Q, or contains both P and Q, depending on the query type. As a corollary of the top-k result, we obtain a linear space and $O(|P| + |Q| + \sqrt{nt})$ query time solution for the document listing problem. We conjecture that any significant improvement over these results is highly unlikely. We also consider the scenario when the query consists of more than two patterns. Finally, we present a space efficient index for these problems.

3.2 Problem Formulation and Related Work

In most of the earlier string retrieval problems, the query consists of a single pattern P. Introduced by Matias et al. [88], the most basic problem is *document listing*, which asks to report all unique documents

containing P. Later Muthukrishnan [93] gave a linear space and optimal $O(|P| + t)$ query time solution, where $|P|$ is the length of the pattern P and t is the number of documents containing P. Often all the documents containing P are not desired, but it suffices to find a subset of the t documents that are most relevant to P. This led to the study of the top-k document retrieval problem, which asks to report only those k documents that are most relevant to P. For this problem, and some standard relevance functions like PageRank (which is independent of P), TermFrequency (i.e., the number of times a pattern appears in the document), TermProximity (i.e., the distance between the closest appearance of a pattern in the document), Hon et al. [67] presented a linear space index with $O(|P| + k \log k)$ query time. Navarro and Nekrich [97] improved this to a linear space index with optimal $O(|P| + k)$ query time. Later Hon et al. [65] showed how to find the documents in $O(k)$ time, once the suffix range of P is located; the idea stems from the work of Shah et al. [112], which primarily presents an external memory index for the problem.

Yet another interesting direction of study is obtaining the full functionality of suffix trees/arrays using indexes which occupy space close to the size of the text. Grossi and Vitter [58], and Ferragina and Manzini [44, 45] were pioneers in the field of designing succinct (or, compressed) space indexes. Their full text indexes, namely Compressed Suffix Arrays (CSA) and FM-Index respectively, have led to the establishment of an exciting field of compressed text indexing. See [96] for an excellent survey. Hon et al. [65, 67] extended the use of compressed text indexes for string matching to that for top-k document listing. However, their index could only answer top-k queries when the relevance function was either PageRank or TermFrequency. It remained unknown whether top-k document retrieval based on TermProximity could be answered in a space efficient way, or not. Recently, Munro et al. [91] answered this question positively by presenting an index which requires a CSA and additional $o(n)$ bits, and answers queries in time $O((|P| + k) \operatorname{polylog} n)$.

Often the input queries are not simplistic, in the sense that there may be more than one pattern. For two patterns P and Q, Muthukrishnan [93] showed that by maintaining an $O(n^{3/2} \log^{O(1)} n)$ space index, all t documents containing both P and Q can be reported in time $O(|P| + |Q| + \sqrt{n} + t)$. Cohen and Porat [35] observed that the problem can be reduced to the *set intersection* problem, and presented an $O(n \log n)$ space (in words) index with query time $O(|P| + |Q| + \sqrt{nt} \log^{5/2} n)$. Subsequently, Hon et al. [62] improved this to an $O(n)$ space (in words) index with query time $O(|P| + |Q| + \sqrt{nt} \log^{3/2} n)$. Also see [65, 67] for a succinct solution, and [48] for a lower bound which states that for $O(|P| + |Q| + \log^{O(1)} n + t)$ query time, $\Omega(n(\log n / \log \log n)^3)$ bits are required. A recent result [78] on the hardness of this problem states that any

improvement other than poly-logarithmic factors is highly unlikely. In this work, we revisit this problem, and also consider the following more general top-k variant.

Problem 1 (top-k Documents with Two Patterns)**.** Index a collection $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ of D strings (called documents) of n characters in total such that when two patterns P and Q, and an integer k come as a query, among all documents containing both P and Q, those k documents that are the most relevant to P can be reported efficiently.

The problem of forbidden pattern queries can be seen as a variation of the two-pattern problem. Specifically, given P and Q, the task is to report those documents that contains P, but not Q. Fischer et al. [48] introduced this problem, and presented an $O(n^{3/2})$-bit solution with query time $O(|P| + |Q| + \sqrt{n} + t)$. Hon et al. [63] presented an $O(n)$-word index with query time $O(|P| + |Q| + \sqrt{nt}\log^{5/2} n)$. Larsen et al. [78] presented a hardness result of this problem via a reduction from boolean matrix multiplication and claimed that any significant (i.e., beyond poly-logarithmic factors) improvement over the existing results are highly unlikely. In this work, we revisit this problem, and also consider the following more general top-k variant.

Problem 2 (top-k Documents with Forbidden Pattern)**.** Index a collection $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ of D strings (called documents) of n characters in total such that when two patterns P and Q, and an integer k come as a query, among all documents containing P, but not Q, those k documents that are the most relevant to P can be reported efficiently.

3.2.1 Relevance Function

In both Problem 1 and Problem 3, the relevance of a document T_d is determined by a function $\mathsf{score}(P, Q, d)$. In particular, for Problem 1, $\mathsf{score}(P, Q, d) = -\infty$ if Q does not occur in T_d, and for Problem 3, $\mathsf{score}(P, Q, d) = -\infty$ if Q occurs in T_d. Otherwise, $\mathsf{score}(P, Q, d)$ is a function of the set of occurrences of P in T_d. We assume that the relevance function is monotonic increasing i.e., $\mathsf{score}(P, Q, d) \leq \mathsf{score}(P', Q, d)$, where P' is a prefix of P. Various functions will fall under this category, such as PageRank and TermFrequency. With respect to the patterns P and Q, a document T_d is more relevant than $\mathsf{T}_{d'}$ iff $\mathsf{score}(P, Q, d) > \mathsf{score}(P, Q, d')$. We remark that term-proximity is monotonic decreasing; however, by considering the negation of the proximity function, this can be made to fit the criteria of monotonic increasing; in this case T_d is more relevant than $\mathsf{T}_{d'}$ iff $\mathsf{score}(P, Q, d) < \mathsf{score}(P, Q, d')$. See the bottom-$k$ document retrieval problem in [101] as an example of a relevance function which is not monotonic.

3.2.2 Our Contribution

The main component of our data structure comprises of the *generalized suffix tree* GST on the collection of documents $\mathcal{D}$. We first show how to obtain a solution for Problem 3. The approach is to first identify some nodes as *marked* and *prime* based on a modified form of the marking scheme of Hon et al. [67]. Then, for each pair of marked and prime node, and properly chosen values of k, we pre-compute and store the answers in a space-efficient way so that the relevant documents can be retrieved efficiently. Our solution for Problem 3 is summarized in the following theorem.

Theorem 1. There exists an $O(n)$ words data structure such that for two patterns P and Q, and an integer k, among all documents containing P, but not Q, in $O(|P|+|Q|+\sqrt{nk})$ time, we can report those k documents that are most relevant to P, where the relevance function is monotonic.

Using the above result as a black box, we can easily obtain the following solution for the document listing problem with a forbidden pattern.

Corollary 1. There exists an $O(n)$ words data structure such that for two patterns P and Q, in time $O(|P|+|Q|+\sqrt{nt})$, we can report all t documents that contain P, but not Q.

Proof. In the query time complexity of Theorem 1, the term $O(|P|+|Q|)$ is due to the time for finding the locus nodes of P and Q in a generalized suffix tree of $\mathcal{D}$. To answer document listing queries using Theorem 1, we perform top-k queries for values of k from $1, 2, 4, 8, \ldots$ up to k'. Here, the number of documents returned by the top-k' query is $< k'$. On the other hand, for $k'' < k'$, the number of documents returned by a top-k'' query is k''. This means the answer to top-k' query is the same as that of a document listing query. Also, $k'/2 \leq t < k'$. Therefore, total time spend over all queries (in addition to the time for initial loci search) can be bounded by $O(\sqrt{n}+\sqrt{2n}+\sqrt{4n}+\cdots+\sqrt{nk'})$ i.e., by $O(\sqrt{nt})$. □ □

Using essentially the same techniques as in the solution of Problem 3, we can obtain the following solution (see Theorem 2) to Problem 1. Once, the loci of P and Q are found, as in Corollary 1, we may use the data structure of Theorem 2 to obtain a solution (see Corollary 2) to the document listing problem with two included patterns.

Theorem 2. There exists an $O(n)$ words data structure such that for two patterns P and Q, and an integer k, among all documents containing both P and Q, in $O(|P| + |Q| + \sqrt{nk})$ time, we can report those k documents that are the most relevant to P, where the relevance function is monotonic.

Corollary 2. There exists an $O(n)$ words data structure such that for two patterns P and Q, in time $O(|P| + |Q| + \sqrt{nt})$, we can report all t documents that contain both P and Q.

3.2.2.1 A note on the tightness of our result.

In order to show the hardness of the document listing problems, let us first define a couple of related problems. Let $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_r\}$ be a collection of sets of total cardinality n. The *set intersection* (resp. *set difference*) problem is to preprocess $\mathcal{S}$ into a data structure, such that for any distinct i and j, we can report the elements in $\mathcal{S}_i \cap \mathcal{S}_j$ (resp. $\mathcal{S}_i \setminus \mathcal{S}_j$) efficiently.

For each element e_x in the collection of sets, create document T_x, where the content of T_x is the sequence of identifiers of all sets containing e_x. Clearly, a forbidden pattern document listing query with $P = i$ and $Q = j$ gives the answer to the set difference problem. Likewise, a two pattern document listing query with $P = i$ and $Q = j$ gives the answer to the set intersection problem. We conclude that the problems are at least as hard as the set difference/intersection problem. The best known upper bound for the set intersection problem is by Cohen and Porat [35], where the space is $O(n)$ words and time is $O(\sqrt{n|\mathcal{S}_i \cap \mathcal{S}_j|})$. The framework by Cohen and Porat can be used to obtain the same space and time solution for the set difference problem. It is unclear whether a better solution for the set difference problem exists, or not; however, the existence of such a solution seems unlikely. We remark that based on Corollary 1 and Corollary 2 the existence of better solutions for the top-k variants are also unlikely.

3.2.3 Organization

The rest of the chapter is dedicated for proving Theorem 1 and Theorem 2. We prove Theorem 1 in Section 3.5 and Theorem 2 in Section 3.6. Other additional results such as indexes for multiple included and forbidden patterns, and space efficient indexes are presented in Section 3.7 and Section 3.8 respectively.

3.3 Computing the relevance function

Note that a pattern P occurs in a document T_d iff $d = \mathsf{doc}(i)$ for some leaf ℓ_i which lies in the suffix range of P. We proceed to prove an important lemma.

Lemma 2. Given the locus nodes of pattern P and Q in GST, and the identifier d of a document in $\mathcal{D}$, by using an $O(n)$ word data-structure, in constant time, we can compute $\mathsf{score}(P, Q, d)$.

Proof. Let $\mathcal{I}$ be a set of integers drawn from a set $\mathcal{U} = \{0, 1, 2, \ldots, 2^{\omega} - 1\}$, where $\omega \geq \log n$ is the word size. Alstrup et al. [5] presents a data-structure of size $O(|\mathcal{I}|)$ words which for a given $a, b \in \mathcal{U}$, $b \geq a$, can report in $O(1)$ time whether $\mathcal{I} \cap [a, b]$ is empty, or not. In the case when $\mathcal{I} \cap [a, b]$ is not-empty, the data-structure returns any arbitrary value in $\mathcal{I} \cap [a, b]$. We build the above data-structure for the sets $\mathcal{I}_d = \{i \mid \mathsf{doc}(i) = d\}$ for $d = 1, 2, \ldots, D$. Total space can be bounded by $O(n)$ words. Using this, we can answer whether a pattern P' occurs in T_d, or not, by checking if there exists an element in $\mathcal{I}_d \cap [L', R']$, where $[L', R']$ is the suffix range of P'. In case an element exists, we a get a leaf $\ell_i \in \mathsf{GST}$ such that $\mathsf{doc}(i) = d$.

For Problem 1, we assign $\mathsf{score}(P, Q, d) = -\infty$ iff T_d does not contain Q. Likewise, for Problem 3, we assign $\mathsf{score}(P, Q, d) = -\infty$ iff T_d contains Q. Otherwise, $\mathsf{score}(P, Q, d)$ equals the relevance of document T_d w.r.t P. Denote by ST_d, the suffix tree of document T_d, and by $path_d(u)$, the string formed by the concatenation of the edge-labels from root to a node u in ST_d. For every node u in ST_d, we maintain the relevance of the pattern $path_d(u)$ w.r.t the document T_d. Also, we maintain a pointer from every leaf ℓ_i of GST to that leaf node ℓ_j of $\mathsf{ST}_{\mathsf{doc}(i)}$ for which $\mathsf{path}_{\mathsf{doc}(i)}(\ell_j)$ is same as $\mathsf{path}(\ell_i)$. Figure 3.1 illustrates this. We now use a more recent result of Gawrychowski et al. [54], which can be summarized as follows: given a suffix tree ST having $|\mathsf{ST}|$ nodes, where every node u has an integer weight $\mathsf{weight}(u) \leq |ST|$, and satisfies the min-heap property, there exists an $O(|\mathsf{ST}|)$ words data structure, such that for any leaf $\ell \in \mathsf{ST}$ and an integer W, in constant time, we can find the lowest ancestor v (if any) of ℓ that satisfies $\mathsf{weight}(v) \leq W$. For every node $u \in \mathsf{ST}_d$, we let $\mathsf{weight}(u) = |\mathsf{path}_d(u)|$. Note that this satisfies the min-heap property and $|\mathsf{weight}(u)| \leq |\mathsf{ST}_d|$. Total space required is bounded by $O(n)$ words. Using the data-structure of Gawrychowski et al., in constant time we can locate the lowest ancestor v of ℓ_j such that $|path_d(v)| \leq |P|$. If $|path_d(v)| = |P|$, then v is the locus node of P in ST_d. Otherwise, one of the children of v is the desired locus, which can be found in constant time (assuming perfect hashing) by checking the $(|path_d(v)| + 1)$th character of P. Therefore, we can compute $\mathsf{score}(P, Q, d)$ in constant time. □ □

3.4 Marking Scheme

We identify certain nodes in GST as *marked* and *prime* based on a parameter g called grouping factor [65]. Starting from the leftmost leaf in GST, we combine every g leaves together to form a group. In particular, the

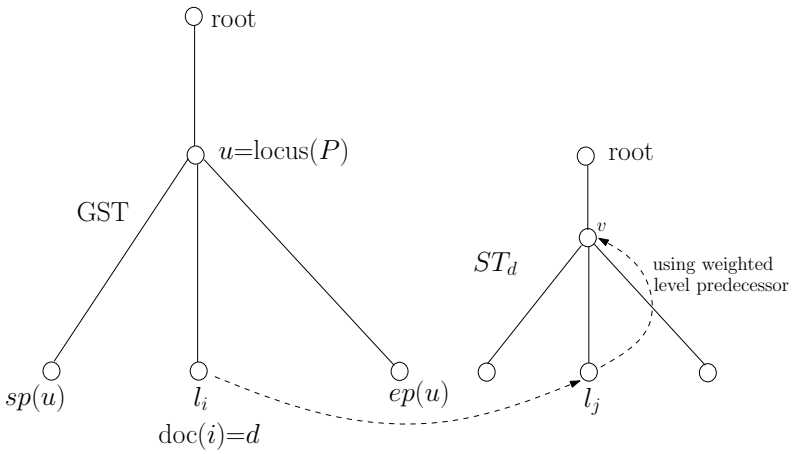

FIGURE 3.1. Illustration of Lemma 2

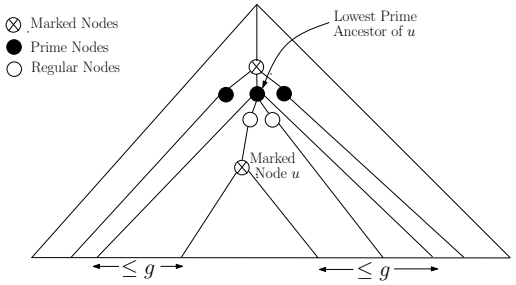

FIGURE 3.2. Marked nodes and Prime nodes

leaves ℓ_1 through ℓ_g form the first group, ℓ_{g+1} through ℓ_{2g} form the second, and so on. We mark the lowest common ancestor (LCA) of the first and last leaves of every group. Moreover, for any two marked nodes, we mark their LCA (and continue this recursively). Figure 3.2 illustrates this. Note that the root node is marked, and the number of marked nodes is at most $2\lceil n/g \rceil$. A marked node is said to be a *lowest marked node* if it has no marked descendant.

Corresponding to each marked node (except the root), we identify a unique node called the prime node [20]. Specifically, the prime node u' corresponding to a marked node u^* is the node on the path from root to u^*, which is a child of the lowest marked ancestor of u^*. We refer to u' as the lowest prime ancestor of any node whose highest marked descendant is u^*. Also, u' is the lowest prime ancestor of u^* itself. Since the root node is marked, there is always such a node. If the parent of u^* is marked, then u' is same as u^*. Observe that for every prime node, the corresponding closest marked descendant is unique. Thus, the number of prime nodes is one less than the number of marked nodes. For any marked node u^* and its lowest prime ancestor u', any subset of the leaves Leaf(u', u^*) is referred to as *fringe leaves*.

Fact 1 ([20, 65])**.** The following are some crucial properties of marked and prime nodes.

1. The number of marked nodes and prime nodes are bounded by $O(n/g)$.

2. Let u^* be a marked node and u' be its lowest prime ancestor. Then, the number of leaves in the sub-tree of u', but not of u^*, is at most $2g$ i.e., $|\text{Leaf}(u', u^*)| \leq 2g$.

3. If a node u (which may be marked) has no marked descendant, then $|\text{Leaf}(u)| \leq 2g$.

4. Given any node u, in constant time, we can detect whether u has a marked descendant, or not.

We now present a useful lemma.

Lemma 3. Let u be a node in GST such that w.r.t a grouping factor g, u^* is its highest marked descendant, u' is its lowest prime ancestor, and u'' is its lowest marked ancestor. By maintaining an $O(n/g)$ space array, we can compute the pre-order ranks of u^*, u', and u'' in $O(n/g)$ time.

Proof. We maintain two arrays containing all marked and prime nodes (along with their pre-order ranks) of GST. The size of these arrays is bounded by $O(n/g)$. Note that in constant time, we can check if a given node is an ancestor/descendant of another by comparing the ranges of the leaves in their respective sub-tree. Therefore, by examining all elements in the array one by one, we can identify the nodes corresponding to u^*, u', and u''. We remark that although it is possible to achieve constant query time using additional structures, we chose to use this simple structure as such an improvement would not affect the overall query complexity of Theorem 1 or Theorem 2. □ □

3.5 Index for Forbidden Pattern

In this section, we prove Theorem 1. We start with the following definitions.

Definition 1. Let u and v be any two nodes in GST. Then

- $\mathsf{list}(u, v) = \{\mathsf{doc}(i) \mid \ell_i \in \text{Leaf}(u)\} \setminus \{\mathsf{doc}(i) \mid \ell_i \in \text{Leaf}(v)\}$.
- $\mathsf{list}_k(u, v)$ is the set of k most relevant document identifiers in $\mathsf{list}(u, v)$.
- $\mathsf{cand}_k(u, v)$, a **k-candidate set**, is any super set of $\mathsf{list}_k(u, v)$.

Moving forward, we use p and q to denote the loci of the included pattern P and the forbidden pattern Q respectively. Our task is then to report $\mathsf{list}_k(p, q)$.

Lemma 4. Given a candidate set $\mathsf{cand}_k(p, q)$, we can find $\mathsf{list}_k(p, q)$ in time $O(|\mathsf{cand}_k(p, q)|)$.

Proof. We first compute the $\mathsf{score}(P, Q, d)$ of each document identifier d in $\mathsf{cand}_k(p, q)$ in constant time (refer to Lemma 2). If $\mathsf{score}(P, Q, d) = -\infty$, then d does not contribute to $\mathsf{list}_k(p, q)$, and can be safely ignored. The reduced set of documents, say $\mathcal{D}'$, can be found in total $O(|\mathsf{cand}_k(u, v)|)$ time. Furthermore, we maintain only the distinct documents in $\mathcal{D}'$; this is achieved in $O(|\mathcal{D}'|)$ time using a bit-array of size D. Among these distinct documents, we first find the document identifier, say d_k, with the kth largest $\mathsf{score}(P, Q, \cdot)$ value in $O(|\mathcal{D}'|)$ time using order statistics [36]. Finally, we report the identifiers d that satisfy $\mathsf{score}(P, Q, d) \geq \mathsf{score}(P, Q, d_k)$. Time required for the entire process can be bounded by $O(|\mathsf{cand}_k(u, v)|)$. □ □

Lemma 5. For any two nodes u and v in GST, let $u^{\downarrow}$ be u or a descendent of u and $v^{\uparrow}$ be v or an ancestor of v, and $\mathcal{L} = \text{Leaf}(u, u^{\downarrow}) \cup \text{Leaf}(v^{\uparrow}, v)$. Then, $\mathsf{list}_k(u, v) \subseteq \mathsf{list}_k(u^{\downarrow}, v^{\uparrow}) \cup \{\mathsf{doc}(i) \mid \ell_i \in \mathcal{L}\}$.

Proof. First observe that $\mathsf{list}_k(u, v) \subseteq \mathsf{list}(u^{\downarrow}, v^{\uparrow}) \cup \{\mathsf{doc}(i) \mid \ell_i \in \mathcal{L}\}$. Let, d be a document identifier in $\mathsf{list}(u^{\downarrow}, v^{\uparrow})$ such that $d \notin \mathsf{list}_k(u^{\downarrow}, v^{\uparrow})$ and $d \in \mathsf{list}_k(u, v)$. This holds when $\mathsf{score}(\mathsf{path}(u), \mathsf{path}(v), d) > \mathsf{score}(\mathsf{path}(u), \mathsf{path}(v), d_k)$, where d_k is the document identifier in $\mathsf{list}_k(u^{\downarrow}, v^{\uparrow})$ with the lowest score. Note that $\mathsf{score}(\mathsf{path}(u^{\downarrow}), \mathsf{path}(v^{\uparrow}), d_k) \geq \mathsf{score}(\mathsf{path}(u^{\downarrow}), \mathsf{path}(v^{\uparrow}), d)$. Therefore, d must appear in $\{\mathsf{doc}(i) \mid \ell_i \in \mathcal{L}\}$, and the lemma follows. □ □

3.5.1 Index Construction

The data-structure in the following lemma is the most intricate component for retrieving $\mathsf{list}_k(p, q)$.

Lemma 6. For grouping factor $g = \sqrt{n\kappa}$, there exists a data-structure requiring $O(n)$ bits of space such that for any marked node u^* and a prime node v' (resp. a lowest marked node v''), we can find $\mathsf{list}_\kappa(u^*, v')$ (resp. $\mathsf{list}_\kappa(u^*, v'')$) in $O(\sqrt{n\kappa})$ time.

We prove the lemma in the following section. Using the lemma, we describe how to obtain $\mathsf{list}_k(p, q)$ in $O(|P| + |Q| + \sqrt{nk})$ time, thereby proving Theorem 1.

Let $\kappa \geq 1$ be a parameter to be fixed later. For grouping factor $\sqrt{n\kappa}$, construct the data-structure DS_κ in Lemma 6 above. Furthermore, we maintain a data-structure DS'_κ, as described in Lemma 3, such that for any node, in $O(\sqrt{n/\kappa})$ time, we can find its highest marked descendant/lowest prime ancestor (if any), and its lowest marked ancestor; this takes $O(\sqrt{n/\kappa} \log n)$ bits. Construct the data-structures DS_κ and DS'_κ for $\kappa = 1, 2, 4, 8, \ldots, D$. Total space is bounded by $O(n)$ words.

3.5.2 Answering top-k Query

For the patterns P and Q, we first locate the loci p and q in $O(|P| + |Q|)$ time. Now for a top-k query, let $k' = \min\{D, 2^{\lceil \log k \rceil}\}$ and $g' = \sqrt{nk'}$. Note that $k \leq k' < 2k$. Depending on whether p has any marked node below it (which can be verified in constant time using Fact 1), we have the following two cases. We arrive at Theorem 1 by showing that in either case $\mathsf{list}_k(p, q)$ can be found in an additional $O(\sqrt{nk})$ time.

case:1 Assume that p contains a descendant marked node, and therefore, the highest descendant marked node, say p^*. If p is itself marked, then $p^* = p$. Let q' be the lowest prime ancestor of q, if exists; otherwise, let q' be the lowest marked ancestor of q. Both p^* and q' are found in $O(n/g')$ (refer to Lemma 3). Now using the data-structure $\mathsf{DS}_{k'}$, we find $\mathsf{list}_{k'}(p^*, q')$ in $O(\sqrt{nk'})$ time. The number of leaves in $\mathcal{L} = \text{Leaf}(p, p^*) \cup \text{Leaf}(q', q)$ is bounded by $O(g')$ (see Fact 1). Finally, we compute $\mathsf{list}_k(p, q)$ from $\mathsf{list}_{k'}(p^*, q')$ and $\mathcal{L}$ (refer to Lemmas 4 and 5). Time required can be bounded by $O(n/g' + \sqrt{nk'} + k' + g')$ i.e., by $O(\sqrt{nk})$.

case:2 If there is no descendant marked node of p, then $|\text{Leaf}(p)| < 2g'$ (see Fact 1). Using the document array, we find $\mathsf{doc}(i)$ corresponding to each leaf $\ell_i \in \text{Leaf}(p)$ in constant time. These documents constitute a k-candidate set, and the required top-k documents are found using Lemma 4. Time required can be bounded by $O(g')$ i.e., by $O(\sqrt{nk})$.

3.5.3 Proof of Lemma 6

We first prove the lemma for a marked node and a prime node. Later, we show how to easily extend this concept to a marked node and a lowest marked node.

A slightly weaker version of the result can be easily obtained as follows: maintain $\mathsf{list}_\kappa(\cdot, \cdot)$ for all pairs of marked and prime nodes explicitly for $g = \sqrt{n\kappa}$. This requires space $O((n/g)^2 \kappa \log D)$ bits i.e., $O(n \log n)$ bits (off by a factor of $\log n$ from our desired space), but offers $O(\kappa)$ query time (better than the desired time). Note that this saving in time will not have any effect on the time complexity of our final result implying that we can afford to spend any time up to $O(\sqrt{n\kappa})$, but the space cannot be more than $O(n)$ bits. Therefore, we seek to encode these lists in a compressed form, such that each list can be decoded back in $O(\sqrt{n\kappa})$ time. The scheme is recursive, and is similar to that used in [40, 100]. Before we begin with the proof of Lemma 6, let us first present the following useful result.

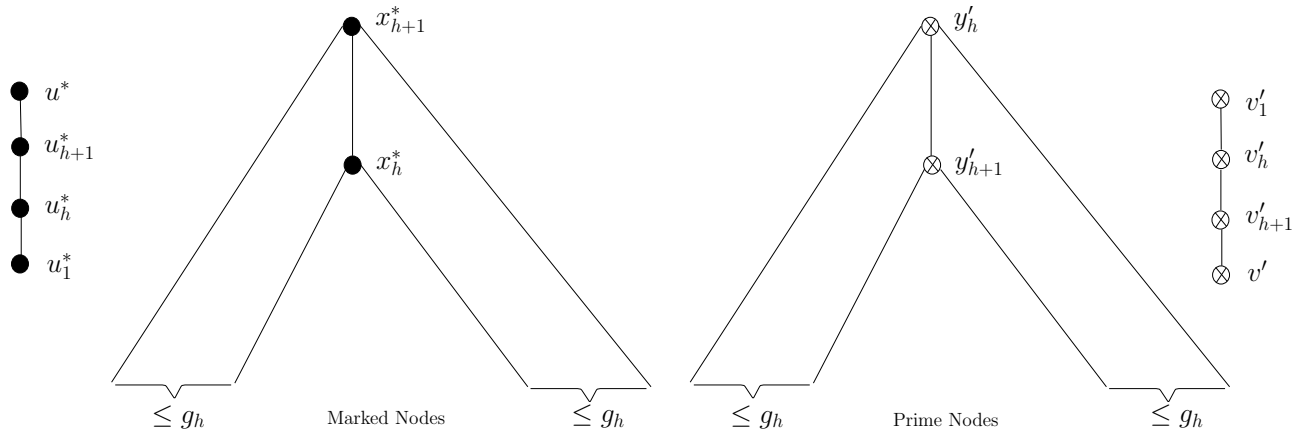

FIGURE 3.3. Recursive encoding scheme.

Fact 2 ([41, 43, 59])**.** A set of m integers from $\{1, 2, \ldots, U\}$ can be encoded in $O(m \log(U/m))$ bits, such that they can be decoded back in $O(1)$ time per integer.

Let $\log^{(h)} n = \log(\log^{(h-1)} n)$ for $h > 1$, and $\log^{(1)} n = \log n$. Furthermore, let $\log^* n$ be the smallest integer α such that $\log^{(\alpha)} n \leq 2$. Let $g_h = \sqrt{n\kappa} \log^{(h)} n$ (rounded to the next highest power of 2). Note that $g = g_{\log^* n}$. Therefore, our task is to report $\mathsf{list}_\kappa(u^*, v')$, whenever a marked node u^* and a prime node u' comes as a query, where both u^* and v' are based on the grouping factor $g_{\log^* n}$. For $1 \leq h \leq \log^* n$, let N^*_h (resp. N'_h) be the set of marked (resp. prime) nodes w.r.t g_h. We maintain the data structure of Lemma 3 over N^*_h and N'_h for $1 \leq h \leq \log^* n$. Using this, for any node u and grouping factor g_h, $1 \leq h \leq \log^* n$, we can compute (i) u's highest marked descendant, or (ii) u's lowest marked/prime ancestor, both in $O(n/g_h)$ time i.e., in $O(\sqrt{n/\kappa}/\log^{(h)} n)$ time (see Lemma 3). The space in bits can be bounded as follows:

$$\sum_{h=1}^{\log^* n} \frac{n \log n}{g_h} = \frac{\sqrt{n} \log n}{\sqrt{\kappa}} \sum_{h=1}^{\log^* n} \frac{1}{\log^{(h)} n} = O(\sqrt{n} \log n)$$

We are now ready to present the recursive encoding scheme. Assume there exists a scheme for encoding $\mathsf{list}_\kappa(\cdot, \cdot)$ of all pairs of marked/prime nodes w.r.t. to g_h in S_h bits of space, such that the list for any such pair can be decoded in T_h time. Then,

$$S_{h+1} = S_h + O\Big(\frac{n}{\log^{(h+1)} n}\Big) \tag{3.1}$$

$$T_{h+1} = T_h + O(\sqrt{n/\kappa}/\log^{(h+1)} n) + O\Big(\frac{\sqrt{n\kappa}}{\log^{(h)} n}\Big) \tag{3.2}$$

By storing the answers explicitly for $h = 1$, the base case is established: $S_1 = O((n/g_1)^2 \kappa \log n)$ i.e, $S_1 = O(n/\log n)$ and $T_1 = O(\kappa)$ plus $O(\sqrt{n/\kappa}/\log n)$ time for finding the highest marked descendant of u^* and the lowest prime ancestor of v', both w.r.t g_1. Solving the above recursions leads to space bound $S_{\log^* n}$ (in bits) and time bound $T_{\log^* n}$ as follows:

$$S_{\log^* n} = S_1 + O\Big(\sum_{h=2}^{\log^* n} \frac{n}{\log^{(h)} n}\Big) \qquad \implies S_{\log^* n} = O(n)$$

$$T_{\log^* n} = T_1 + O\Big(\sum_{h=2}^{\log^* n} \frac{\sqrt{n/\kappa}}{\log^{(h)} n}\Big) + O\Big(\sum_{h=1}^{\log^* n - 1} \frac{\sqrt{n\kappa}}{\log^{(h)} n}\Big) \qquad \implies T_{\log^* n} = O(\sqrt{n\kappa})$$

3.5.3.1 Space Bound.

First we show how to arrive at equation 3.1. Specifically, we show how to encode $\mathsf{list}_\kappa(\cdot,\cdot)$ w.r.t g_{h+1} given that $\mathsf{list}_\kappa(\cdot,\cdot)$ w.r.t g_h is already encoded separately. Let x^*_h (resp. y'_h) be any marked (resp. prime) node w.r.t g_h. Furthermore, let x^*_{h+1} be the lowest marked ancestor (w.r.t g_{h+1}) of x^*_h. The number of leaves in $\text{Leaf}(x^*_{h+1}, x^*_h)$ are bounded above by $2g_h$. Otherwise, due to Fact 1, there is a marked node w.r.t g_h on the path from x^*_{h+1} to x^*_h (excluding both), and hence a marked node w.r.t g_{h+1}; this contradicts the definition of x^*_{h+1}. Likewise, if y'_{h+1} is the highest prime descendant (w.r.t g_{h+1}) of y'_h, then $|\,\text{Leaf}(y'_h, y'_{h+1})| \leq 2g_h$. Figure 3.3 illustrates this. Observe that the difference in $\mathsf{list}_\kappa(x^*_h, y'_h)$ and $\mathsf{list}_\kappa(x^*_{h+1}, y'_{h+1})$ is due to the leaves $\text{Leaf}(x^*_{h+1}, x^*_h)$ and $\text{Leaf}(y'_h, y'_{h+1})$, and at most κ leaves contribute to this difference. The key idea is to encode these leaves in a compressed from, so that they can be decoded later.

Since, explicitly maintaining these contributing leaves is expensive (requiring $O(\kappa \log n)$ bits), we maintain an approximate range in which the leaf lies. Let $f_h = \sqrt{n}/(\sqrt{\kappa}\log^{(h)} n)$ (rounded to the next highest power of 2). We divide the leaves in $\text{Leaf}(x^*_{h+1}, x^*_h)$ (resp. $\text{Leaf}(y'_h, y'_{h+1})$) on either side of the suffix range of x^*_h (resp. y'_{h+1}) into g_h/f_h chunks. Thus, there are $O(\kappa(\log^{(h)} n)^2)$ chunks, and each chunk has $O(f_h)$ leaves. For x^*_{h+1} and y'_{h+1} pair, we maintain those at most κ chunks, leaves of which may contribute to the $(h+1)$-th level list. This can be seen as selecting at most κ integers from an universe of size $O(\kappa(\log^{(h)} n)^2)$, and can be encoded in $O(\kappa \log(\kappa(\log^{(h)} n)^2/\kappa))$ i.e., in $O(\kappa \log^{(h+1)} n)$ bits (refer to Fact 2). Since, the number of marked and prime nodes at the $(h+1)$-th level are both bounded by $O(n/g_{h+1})$, the total size in bits for maintaining these κ chunks for all pairs of marked and prime nodes w.r.t g_{h+1} can be bounded by

$O((n/g_{h+1})^2\kappa \log^{(h+1)} n)$ i.e. by $O(n/\log^{(h+1)} n)$. For g_{h+1}, we also maintain the index w.r.t g_h having size S_h. Thus we have $S_{h+1} = S_h + O(n/\log^{(h+1)} n)$.

3.5.3.2 Time Bound.

Suppose the κ candidate leaves for h-th level marked node u_h^* and prime node v'_h have been retrieved. We show how to obtain the candidate leaves for the $(h+1)$-th level marked node immediately above u_h^* and the $(h+1)$-th level prime node immediately below v'_h. We first locate u_{h+1}^* and v'_{h+1}. This can be achieved in time $O(n/g_{h+1})$ i.e., in $O(\sqrt{n/\kappa}/\log^{(h+1)} n)$ time with the aid of the arrays maintaining N_{h+1}^* and N'_{h+1}. By using the index, having size S_{h+1}, at the $(h+1)$-th level of recursion, we decode the κ chunks for u_{h+1}^* and v'_{h+1}, and for each chunk we find the leaves in it. The number of such leaves is bounded by $O(\kappa f_h)$. By Fact 2, decoding each chunk takes constant time, and therefore finding the set of candidate leaves for u_{h+1}^* and v'_{h+1} is achieved in total time $T_{h+1} = T_h + O(n/g_{h+1}) + O(\kappa f_h)$ i.e., in time $T_h + O(\sqrt{n/\kappa}/\log^{(h+1)} n) + O(\sqrt{n\kappa}/\log^{(h)} n)$.

3.5.3.3 Retrieving top-k Documents.

In the query process described above, at the end of $\log^* n$ levels, we have the candidate leaves for $\mathsf{list}_\kappa(u^*, v')$. The number of such leaves is bounded above by $\kappa f_{\log^* n} = O(\sqrt{n\kappa})$. By using these leaves and the document array, we find the κ most relevant unique document identifiers d in time $O(\sqrt{n\kappa})$.

3.5.3.4 Handling Lowest Marked Nodes.

In the base case, we explicitly maintain the index for a marked node and a lowest marked node (both w.r.t g_1). Suppose the $\mathsf{list}_\kappa(u_h^*, v'_h)$ for an hth level marked node u_h^* and an h-th level lowest marked node v'_h has been encoded. Then, the $(h+1)$-th level lists are encoded by taking the $(h+1)$-th lowest marked ancestor (resp. lowest marked descendant) of u_h^* (resp. v'_h). The number of leaves to be encoded is bounded above by κf_h. Using essentially the same encoding/decoding techniques, space and time complexity can be bounded as before. This concludes the proof of Lemma 6.

3.6 Index for Two Patterns

The extension from the forbidden pattern case to that of two (included) patterns is based on the index for the former. First we present slightly modified definitions of those in Section 3.5 as follows.

Definition 2. Let u and v be any two nodes in GST. Then

- $\mathsf{list}^+(u, v) = \{\mathsf{doc}(i) \mid \ell_i \in \text{Leaf}(u)\} \cap \{\mathsf{doc}(i) \mid \ell_i \in \text{Leaf}(v)\}$.

- $\mathsf{list}^+_k(u, v)$ is the set of k most relevant document identifiers in $\mathsf{list}^+(u, v)$ defined above.

- $\mathsf{cand}^+_k(u, v)$, a k**-candidate set**, is any super set of $\mathsf{list}^+_k(u, v)$.

Moving forward, we use p and q to denote the loci of the two (included) patterns P and Q respectively. Our task is then to report $\mathsf{list}^+_k(p, q)$. Recall that in this scenario, $\mathsf{score}(P, Q, d) = -\infty$ iff T_d does not contain Q; otherwise, $\mathsf{score}(P, Q, d)$ is the relevance of T_d w.r.t P. For notational consistency, we denote this function by $\mathsf{score}^+(P, Q, d)$ The following two lemmas are immediate from the arguments present in Lemma 4 and Lemma 5.

Lemma 7. Given a candidate set $\mathsf{cand}^+_k(p, q)$, we can find $\mathsf{list}^+_k(p, q)$ in time $O(|\mathsf{cand}^+_k(p, q)|)$.

Lemma 8. For any two nodes u and v in GST, let $u^{\downarrow}$ (resp. $v^{\downarrow}$) be u or a descendant of u (resp. v or a descendant of v). Then, $\mathsf{list}^+_k(u, v) \subseteq \mathsf{list}^+_k(u^{\downarrow}, v^{\downarrow}) \cup \{\mathsf{doc}(i) \mid \ell_i \in \text{Leaf}(u, u^{\downarrow}) \cup \text{Leaf}(v, v^{\downarrow})\}$

3.6.1 Index Construction

The data-structure in the following lemma is the most intricate component for retrieving $\mathsf{list}^+_k(p, q)$.

Lemma 9. For grouping factor $g = \sqrt{n\kappa}$, there exists a data-structure requiring $O(n)$ bits of space such that for any two marked nodes u^* and v^*, we can find $\mathsf{list}^+_\kappa(u^*, v^*)$ in $O(\sqrt{n\kappa})$ time.

sketch. The idea is similar to the proof of Lemma 6 in Section 3.5.3 for the case of a lowest marked node. Instead of encoding $\mathsf{list}_\kappa(\cdot, \cdot)$ for every marked node and lowest marked node pair, we encode $\mathsf{list}^+_\kappa(\cdot, \cdot)$ for every pair of marked nodes. Since, the number of marked nodes is bounded by $O(n/g)$, as are the number of lowest marked nodes, space complexity does not change. As we apply the same encoding scheme as in the case of Lemma 6, the decoding mechanism remains the same as well. Therefore, time complexity remains unchanged, and the lemma follows. □ □

For $\kappa = 1, 2, 4, 8, \ldots, D$, we maintain the data structure of the above lemma w.r.t grouping factor $\sqrt{n\kappa}$. We also maintain the data structure of Lemma 3, only for marked nodes, again w.r.t grouping factor $\sqrt{n\kappa}$, $\kappa = 1, 2, 4, 8, \ldots, D$. Total space can be bounded by $O(n)$ words.

3.6.2 Answering top-k Query

For the patterns P and Q, we first locate the loci p and q in $O(|P| + |Q|)$ time. Now for a top-k query, let $k' = \min\{D, 2^{\lceil \log k \rceil}\}$ and $g' = \sqrt{nk'}$. Note that $k \leq k' < 2k$. Depending on whether p and q both have a marked descendant (which can be verified in constant time using Fact 1), we have the following two cases. We arrive at Theorem 1 by showing that in either case $\mathsf{list}_k^+(p, q)$ can be found in an additional $O(\sqrt{nk})$ time.

Case 1: Assume that the subtree of both p and q contains a marked node. Let the respective highest marked descendants be p^* and q^*. If p (resp. q) is itself marked, then $p^* = p$ (resp. $q^* = q$). Both p^* and q^* are found in $O(n/g')$ (refer to Lemma 3). Using the data-structure of Lemma 9 for p^*, q^*, and k', we retrieve $\mathsf{list}_{k'}^+(p^*, q^*)$ in $O(\sqrt{nk'})$ time. The number of leaves in $\mathcal{L} = \mathrm{Leaf}(p, p^*) \cup \mathrm{Leaf}(q, q^*)$ is $O(g')$ (see Fact 1). Finally, we compute $\mathsf{list}_k^+(p, q)$ from $\mathsf{list}_{k'}^+(p^*, q^*)$ and $\mathcal{L}$ (refer to Lemmas 7 and 8). Time required can be bounded by $O(n/g' + \sqrt{nk'} + k' + g')$ i.e., by $O(\sqrt{nk})$.

Case 2: If there is no marked descendant of p, then $|\mathrm{Leaf}(p)| < 2g'$ (see Fact 1). Likewise, if there is no marked descendant of q, then $|\mathrm{Leaf}(q)| < 2g'$. In either case, using the document array, we can find the document identifiers corresponding to each of these leaves in total $O(g')$ time. These documents constitute a k-candidate set, and the required top-k documents are found using Lemma 7. Time required can be bounded by $O(g')$ i.e., by $O(\sqrt{nk})$.

3.7 Generalizing to Multiple Patterns

The basic idea behind generalizing the data structure for the forbidden pattern query to that for m_i included patterns and m_e excluded (i.e., forbidden) patterns, where the relevance is w.r.t a single included pattern, is to pre-compute the answers and store them for all possible marked and prime node pairs. Likewise, we also maintain the answers explicitly for all marked nodes and lowest marked nodes combination.

To limit the space to $O(n)$ words, we first concentrate on how to modify Lemma 6. The key idea is to modify the grouping factor g_h appropriately. The encoding procedure and the decoding procedure remain largely unchanged, just that when we encode the most relevant κ leaves (i.e., the at most κ chunks in which they occur), we have to take into account the leaves falling between all h-th level and $(h+1)$-th level marked nodes. Likewise, for prime nodes and lowest marked nodes. To this end, we choose $g_h = n^{1-1/m}\kappa^{1/m}\log^{(h)} n$, where $m = m_i + m_e$. Also, by choosing $f_h = n^{1-1/m}\kappa^{1/m-1}/\log^{(h)} n$, we maintain the number of chunks at $O(\kappa(\log^{(h)} n)^2)$. Thus, the space for maintaining these chunks at the $(h+1)$-

th level requires $O((n/g_{h+1})^m \kappa \log^{(h+1)} n)$ bits i.e., $O(n/\log^{(h+1)} n)$ bits. The space required by additional structures for maintaining the marked and prime nodes at this level is bounded by $O((n/g_{h+1}) \log n)$ bits. Clearly, total space over $\log^* n$ levels is bounded by $O(n)$ bits.

Although the space is maintained, the query time suffers. Recall that in the querying phase, we have to decode κ chunks at every recursion level, and find the $O(f_h)$ leaves in each chunk. Thus, the time needed to decode the leaves is $O(n^{1-1/m}k^{1/m})$.

We maintain the data structure of Lemma 6 w.r.t grouping factor $\sqrt{n\kappa}$ for $\kappa = 1, 2, 4, \ldots, D$, which causes the space to rise to $O(n)$ words. The idea for multiple patterns is similar, just that the grouping factor has to be changed from $\sqrt{n\kappa}$ to $n^{1-1/m}\kappa^{1/m}$ to accommodate storage for every marked and prime (or lowest marked) nodes. Total space is $O(n)$ words.

In the querying procedure, we begin by locating the desired (lowest) marked node/prime node corresponding to the locus of each of the query patterns. In addition to the initial $O(\sum_{j=1}^{m_i} |P_j| + \sum_{j=1}^{m_e} |Q_j|)$ time for finding the loci of each pattern, this takes time $O(mn^{1/m}/k^{1/m})$. Now, we need to compute the score for all fringe leaves. Note that computing score for a document corresponding to a leaf takes $O(m)$ time as we need to issue m range emptiness queries (refer to Lemma 2). We assume that the relevance is w.r.t to the first included pattern P_1, which can be computed in constant time. Since, the total number of fringe leaves is $O(mn^{1-1/m}k^{1/m})$, computing score for all leaves takes $O(m^2n^{1-1/m}k^{1/m})$ time. Combining these, we obtain the following theorem.

Theorem 3. Given m_i included patterns P_j, $1 \leq j \leq m_i$, and m_e forbidden patterns Q_j, $1 \leq j \leq m_e$, by maintaining an $O(n)$ word index, we can answer top-k forbidden pattern queries in $O(\sum_{j=1}^{m_i} |P_j| + \sum_{j=1}^{m_e} |Q_j| + m^2n^{1-1/m}k^{1/m})$ time, where $m = m_i + m_e$ and the relevance is w.r.t the first included pattern P_1.

We conclude this section by remarking that the same space and time trade-off can be attained for the multiple (included) pattern problem using essentially the same techniques and data structures.

3.8 Space Efficient Index

The linear space (in words) data structure for the forbidden pattern queries comprises of the following components.

1. Suffix tree which requires $O(n)$ words of space and allows to search for the suffix range of a pattern P in $O(|P|)$ time, and find suffix array value or inverse suffix array value in $O(1)$ time.

2. A data structure occupying $O(\sqrt{n}\log n)$ bits of space, such that for any node we can find its highest marked/prime descendant or lowest marked/prime ancestor.

3. An $O(n)$ words data structure which maintains a pre-computed top-k (for some selected values of k) list for pair of marked and prime nodes, or a pair marked and a lowest marked node.

4. Document array occupying $O(n)$ words of space, such that for a leaf ℓ_i, we can find $\mathsf{doc}(i)$ in constant time.

5. A data structure occupying $O(n)$ words of space, such that for any document T_d, we can compute $\mathsf{score}(P, Q, d)$ in constant time.

In order to reduce the space of the index close to the size of the text, our first step is to replace the suffix tree with a compressed suffix array CSA that occupies $|\mathsf{CSA}|$ bits of space, and allows us to find the suffix range of a pattern P' in $\mathsf{search}(P') = \Omega(|P'|)$ time and suffix array/inverse suffix array value[1] in $\mathsf{t_{SA}} = \Omega(1)$ time. Different versions of CSA may be used to obtain different trade offs, such as the CSA of Belazzougi and Navarro [11] offers $\mathsf{search}(P') = O(|P'|)$ and $\mathsf{t_{SA}} = O(\log^{1+\epsilon} n)$. Also, denote by $|\mathsf{CSA}^*|$ the maximum of the space (in bits) needed to store the CSA of the concatenated text of all documents, or the total space needed to store the individual CSA of each document.

To limit the space required for storing the pre-computed top-k lists, $k = 1, 2, 4, \ldots, D$, to $O(n)$ bits, we first concentrate on how to modify Lemma 6. The key idea is to modify the grouping factor g_h appropriately. To this end, we choose $g_h = \sqrt{n\kappa \log D}\log^{(h)} n$. Thus, $g_{\log^* D} = \sqrt{n\kappa \log D}$. Note that the choice of g_h increases the number of leaves between the h-th level and the $(h+1)$-th level marked (or, prime) nodes (refer to Section 3.5.3) to $O(\sqrt{n\kappa \log D}\log^{(h)} n)$. By choosing $f_h = \sqrt{n \log D}/(\sqrt{\kappa}\log^{(h)} n)$, we maintain the number of chunks at $O(\kappa(\log^{(h)} n)^2)$. Thus, the space required to encode these chunks is $O((n/g_{h+1})^2 \kappa \log^{(h+1)} n)$ bits i.e., by $O((n/\log D)\log^{(h+1)} n)$ bits. Clearly, total space over $\log^* n$ levels is bounded by $O(n/\log D)$ bits.

[1]Given the suffix array SA of a text T, $\mathsf{SA}[i] = j$ and $\mathsf{SA}^{-1}[j] = i$ if and only if the ith smallest lexicographically suffix of T starts at position j in T. The values returned by $\mathsf{SA}[\cdot]$ and $\mathsf{SA}^{-1}[\cdot]$ are called the suffix array value and inverse suffix array value respectively.

Recall that, in the decoding phase, we have to decode $O(\kappa f_h)$ leaves at every level, and for each leaf, we need to find the corresponding document. This is achieved using the following lemma.

Lemma 10. Given a CSA, using an additional $n + o(n)$ bit data structure, for any leaf ℓ_i, we can find $\mathsf{doc}(i)$ in $\mathsf{t_{SA}}$ time.

Proof. We use the following data-structure [56, 90]: a bit-array $B[1..m]$ can be encoded in $m + o(m)$ bits, such that $rank_B(q, i) = |\{j \mid B[j] = q \text{ and } 1 \leq j \leq i\}|$ can be found in constant time.

Consider the concatenated text T of all the documents which has length n. Let B be a bit array of length n such that $B[i] = 1$ iff a document starts at the position i in the concatenated text T. We maintain a $rank$ structure on this bit-array. Space required is $n + o(n)$ bits. We find the text position j of ℓ_i in $\mathsf{t_{SA}}$ time. Then $\mathsf{doc}(i) = rank_B(1, j)$, and is retrieved in constant time. Time required can be bounded by $\mathsf{t_{SA}}$. □ □

Thus, the overall query time increases to $O(\mathsf{t_{SA}} \cdot \sqrt{n\kappa \log D})$, resulting in the following modified form of Lemma 6.

Lemma 11. For grouping factor $g = \sqrt{n\kappa \log D}$, there exists a data-structure requiring $O(n/\log D)$ bits of space such that for any marked node u^* and a prime node v' (resp. a lowest marked node v''), we can find $\mathsf{list}_\kappa(u^*, v')$ (resp. $\mathsf{list}_\kappa(u^*, v'')$) in $O(\mathsf{t_{SA}} \cdot \sqrt{n\kappa \log D})$ time.

For $\kappa = 1, 2, 4, \ldots, D$, we maintain the above data structure for grouping factor $\sqrt{n\kappa \log D}$. Total space is bounded by $O(n)$ bits. Note that choosing the grouping factor as $\sqrt{n\kappa \log D}$ increases the number of fringe leaves for any marked and prime pair to $O(\sqrt{n\kappa \log D})$. Also, the number of leaves below a lowest marked node is $O(\sqrt{n\kappa \log D})$.

The total space occupied by the data structure for finding marked/prime nodes (see Lemma 3) reduces to $O(\sqrt{n \log n})$ bits. We maintain the topology of the GST using the succinct tree encoding of Navarro and Sadakane [110] in additional $4n + o(n)$ bits, such that given any node v, in constant time, we can find

- depth of v i.e., the number of nodes on the path from root to v
- the leftmost/rightmost leaf in the sub-tree of v

Using these data structures, for any κ, we can find the desired marked/prime node, or verify their existence in $O(1 + \sqrt{n/(\kappa \log D)})$ time. At this point, all we are required to do is present a space-efficient index using which we can compute $\mathsf{score}(P, Q, d)$ for a document d. Although, in Lemma 2, we can compute the score

for any monotonic function, unfortunately, for the space efficient index, we are only limited to PageRank and TermFrequency. The following lemma demonstrates how to achieve this.

Lemma 12. Given the suffix ranges of pattern P and Q, and a document identifier d, by maintaining CSA and an additional $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bit data structure, in $O(\mathsf{t_{SA}} \log \log n)$ time we can compute $\mathsf{score}(P, Q, d)$.

Proof. Number of occurrences of d in any suffix range $[L, R]$ is given by $rank_{\mathsf{DA}}(d, R) - rank_{\mathsf{DA}}(d, L-1)$, where $rank_{\mathsf{DA}}(q, i) = |\{j \mid \mathsf{DA}[j] = q \text{ and } 1 \leq j \leq i\}|$ Thus, $\mathsf{score}(P, Q, d) = -\infty$ if d occurs in the suffix range of Q. Otherwise, for PageRank $\mathsf{score}(P, Q, d) = d$, and for TermFrequency $\mathsf{score}(P, Q, d) = rank_{\mathsf{DA}}(d, R_p) - rank_{\mathsf{DA}}(d, L_p - 1)$, where $[L_p, R_p]$ is the suffix range of P. Space and time complexity is due to the following result of Hon et al. [65]: the document array DA can be simulated using CSA and an additional $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bit data structure to support $rank_{\mathsf{DA}}$ operation in $O(\mathsf{t_{SA}} \log \log n)$ time. □ □

Thus for each leaf we can find the corresponding document identifier d in $\mathsf{t_{SA}}$ time, and compute $\mathsf{score}(P, Q, d)$ in $O(\mathsf{t_{SA}} \log \log n)$ time. The number of such leaves we need to explictly check are bounded by $O(\sqrt{nk \log D})$. Also, for a marked node and a prime node pair (or a marked node and lowest marked node pair), we can find the corresponding top-k document identifiers in $O(\mathsf{t_{SA}} \sqrt{nk \log D})$ time. Combining these, we arrive at the following theorem.

Theorem 4. By maintaining CSA and additional data structures occupying $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(n)$ bits, we can answer top-k forbidden pattern queries in $O(\mathsf{search}(P) + \mathsf{search}(Q) + \mathsf{t_{SA}} \sqrt{nk \log D} \log \log n)$ time, when the relevance function is PageRank or TermFrequency.

As a simple corollary (refer to Corollary 1 for details), we obtain the following result.

Corollary 3. By maintaining CSA and additional data structures occupying $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(n)$ bits, we can answer forbidden pattern document listing queries in $O(\mathsf{search}(P) + \mathsf{search}(Q) + \mathsf{t_{SA}} \sqrt{nt \log D} \log \log n)$ time, where t is the number of documents reported.

We conclude this section by remarking that the same space and time trade-off can be attained for the two (included) pattern problem using essentially the same techniques and data structures. Also, space efficient

data structures for multiple included and forbidden patterns can be obtained by combining the ideas of this section and of Section 3.7.

Chapter 4
Document Retrieval with Forbidden Extension

In this chapter, we continue our ranked document retrieval with multi-pattern problem by introducing the problem of document retrieval with forbidden extensions.

Let $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ be a collection of D string documents of n characters in total, and P^+ and P^- be two query patterns, where P^+ is a proper prefix of P^-. We call P^- as the forbidden extension of the included pattern P^+. A forbidden extension query $\langle P^+, P^- \rangle$ asks to report all occ documents in $\mathcal{D}$ that contains P^+ as a substring, but does not contain P^- as one. A top-k forbidden extension query $\langle P^+, P^-, k \rangle$ asks to report those k documents among the occ documents that are most relevant to P^+. We present a linear index (in words) with an $O(|P^-| + occ)$ query time for the document listing problem. For the top-k version of the problem, we achieve the following results, when the relevance of a document is based on PageRank:

- an $O(n)$ space (in words) index with $O(|P^-| \log \sigma + k)$ query time, where σ is the size of the alphabet from which characters in $\mathcal{D}$ are chosen. For constant alphabets, this yields an optimal query time of $O(|P^-| + k)$.

- for any constant $\epsilon > 0$, a $|\mathsf{CSA}| + |\mathsf{CSA}^*| + D \log \frac{n}{D} + O(n)$ bits index with $O(\mathsf{search}(P) + k \cdot \mathsf{t_{SA}} \cdot \log^{2+\epsilon} n)$ query time, where $\mathsf{search}(P)$ is the time to find the suffix range of a pattern P, $\mathsf{t_{SA}}$ is the time to find suffix (or inverse suffix) array value, and $|\mathsf{CSA}^*|$ denotes the maximum of the space needed to store the *compressed suffix array* CSA of the concatenated text of all documents, or the total space needed to store the individual CSA of each document.

In the previous chapter, we presented a linear space (in words) and $O(|P|+|Q|+\sqrt{nk})$ query time solution for the top-k version of forbidden pattern problem, which yields a linear space and $O(|P| + |Q| + \sqrt{n \cdot occ})$ solution to the listing problem. We also showed that this is close to optimal via a reduction from the *set intersection/difference* problem. The *document listing with forbidden extension problem* is a stricter version of the forbidden pattern problem, and asks to report all documents containing an included pattern P^+, but not its forbidden extension P^-, where P^+ is a proper prefix of P^-. As shown in the previous chapter, the forbidden pattern problem suffers from the drawback that linear space (in words) solutions are unlikely to yield a solution better than $O(\sqrt{n/occ})$ per document reporting time. Thus, it is of theoretical interest to see

whether this hardness can be alleviated by putting further restrictions on the forbidden pattern. We show that indeed in case when the forbidden pattern is an extension of the included pattern, by maintaining a linear space index, the document listing problem can be answered in optimal time for constant alphabet. For further theoretical interest, we study the following more general top-k variant.

Problem 3 (top-k Document Listing with Forbidden Extension)**.** Let $\mathcal{D} = \mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D$ be D weighted strings (called documents) of n characters in total. Our task is to index $\mathcal{D}$ such that when a pattern P^+, its extension P^-, and an integer k come as a query, among all documents containing P^+, but not P^-, we can report the k most weighted ones.

Results. Our contributions to Problem 3 are summarized in the following theorems.

Theorem 5. The top-k forbidden extension queries can be answered by maintaining an $O(n)$-words index in optimal $O(|P^-| + k)$ time.

4.1 Linear Space Index

In this section, we present our linear space index. We use some well-known range reporting data-structures [22, 98, 105] and the chaining framework of Muthukrishnan [65, 94], which has been extensively used in problems related to document listing.Using these data structures, we first present a solution to the document listing problem. Then, we present a simple linear index for the top-k version of the problem, with a $O(|P^-| \log n + k \log n)$ query time. Using more complicated techniques, based on the heavy path decomposition of a tree, we improve this to arrive at Theorem 5.

Orthogonal Range Reporting Data Structure.

Fact 3 ([98])**.** A set of n weighted points on an $n \times n$ grid can be indexed in $O(n)$ words of space, such that for any $k \geq 1$, $h \leq n$ and $1 \leq a \leq b \leq n$, we can report k most weighted points in the range $[a, b] \times [0, h]$ in decreasing order of their weights in $O(h + k)$ time.

Fact 4 ([105])**.** A set of n 3-dimensional points (x, y, z) can be stored in an $O(n)$-word data structure, such that we can answer a three-dimensional dominance query in $O(\log n + output)$ time, with outputs reported in the sorted order of z-coordinate.

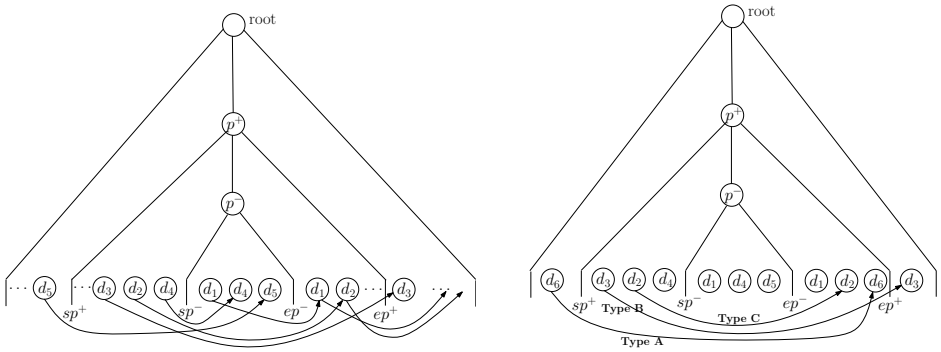

FIGURE 4.1. Chaining framework. Although Leaf(p^+) has documents d_1, d_2, d_3, d_4, and d_5, only d_2 and d_3 qualify as output, since d_1, d_4, and d_5 are present in Leaf(p^-).

Fact 5 ([22])**.** Let A be an array of length n. By maintaining an $O(n)$-words index, given two integers i, j, where $j \geq i$, and a positive integer k, in $O(k)$ time, we can find the k largest (or, smallest) elements in the subarray $A[i..j]$ in sorted order.

Chaining Framework. For every leaf ℓ_i in GST, we define $\mathsf{next}(i)$ as the minimum index $j > i$, such that $\mathsf{doc}(j) = \mathsf{doc}(i)$. We denote i as the source of the chain and $\mathsf{next}(i)$ as the destination of the chain. We denote by $(-\infty, i)$ (resp. (i, ∞)) the chain that ends (resp. starts) at the first (resp. last) occurrence ℓ_i of a document. Figure 4.1(a) illustrates chaining. The integral part of our solution involves categorizing the chains into the following 3 types, and then build separate data structure for each type.

Type A: $i < sp^+$ and $ep^- < \mathsf{next}(i) \leq ep^+$

Type B: $sp^+ \leq i < sp^-$ and $\mathsf{next}(i) > ep^+$

Type C: $sp^+ \leq i < sp^-$ and $ep^- < \mathsf{next}(i) \leq ep^+$

Figure 4.1(b) illustrates different types of chains. It is easy to see that any output of forbidden extension query falls in one of these 3 types. Also observe that the number of chains is n. For a type A chain $(i, \mathsf{next}(i))$, we refer to the leaves ℓ_i and $\ell_{\mathsf{next}(i)}$ as type A leaves; similar remarks hold for type B and type C chains. Also, LCA of a chain (i, j) refers to the LCA of the leaves ℓ_i and ℓ_j. Furthermore, with slight abuse of notation, for any two nodes $u, v \in \mathsf{GST}$, we denote by $\mathsf{depth}(u, v)$, the depth of the LCA of the nodes u and v.

Document Listing Index. Linear space index for the forbidden extension document listing problem is achieved by using Fact 5. We store two arrays as defined below.

A_{src}**:** $A_{src}[i]$=$\mathsf{next}(i)$, for each chain $(i, \mathsf{next}(i))$

A_{dest}: $A_{dest}[\mathsf{next}(i)]$=$i$, for each chain $(i, \mathsf{next}(i))$

Querying in A_{src} within the range $[sp^+, sp^- - 1]$ will give us the chains in descending order of their destination, we stop at ep^- to obtain all the Type B and Type C chains. We query in A_{dest} within the range $[ep^- + 1, ep^+]$ to obtain the chains in ascending order of their source and stop at sp^+ to obtain all the type A chains. Time, in addition to that required for finding the suffix ranges, can be bounded by $O(|P^-| + occ)$.

4.1.1 A Simple $O(|P^-| \log n + k \log n)$ time Index

We start with a simple indexing scheme for answering top-k forbidden extension query. In this section, we design data structures by processing different types of chains separately and mapping them into range reporting problem.

Processing Type A and Type B Chains. For type A chains, we construct range reporting data structure, as described in Fact 3, with each chain (i, j), $j = \mathsf{next}(i)$, mapped to a weighted two dimensional point $(j, \mathsf{depth}(i, j))$ with weight $\mathsf{doc}(i)$. Likewise, for type B chains, we map chain (i, j) to the point $(i, \mathsf{depth}(i, j))$ with weight $\mathsf{doc}(i)$. Recall that d is the PageRank of the document T_d. For Type A chains, we issue a range reporting query for $[ep^- + 1, ep^+] \times [0, \mathsf{depth}(p^+)]$. For Type B chains, we issue a range reporting query for $[sp^+, sp^- - 1] \times [0, \mathsf{depth}(p^+)]$. In either case, we can obtain the top-k leaves in sorted order of their weights in $O(|P^-| + k)$ time, which gives us the following lemma.

Lemma 13. There exists an $O(n)$ words data structure, such that for a top-k forbidden extension query, we can report the top-k Type A and Type B leaves in time $O(|P^-| + k)$.

Processing Type C Chains. We maintain the 3-dimensional dominance structure of Fact 4 at each node of GST. For a chain (i, j), $j = \mathsf{next}(i)$, we store the point $(i, j, \mathsf{doc}(i))$ in the dominance structure maintained in the node $lca(i, j)$. For query answering, we traverse the path from p^+ to p^-, and query the dominance structure of each node on this path with x-range $[-\infty, sp^- - 1]$ and y-range $[ep^- + 1, \infty]$. Any chain falling completely outside of $\mathsf{GST}(p^+)$ will not be captured by the query, since their LCA lies above p^+. There can be at most $\mathsf{depth}(p^-) - \mathsf{depth}(p^+) + 1 \leq |P^-| = \Theta(n)$ sorted lists containing k elements each. The $\log n$ factor in the query of Fact 4 is due to locating the first element to be extracted; each of the remaining $(k - 1)$ elements can be extracted in constant time per element. Therefore, time required for dominance queries (without extracting the elements) is bounded by $O(|P^-| \log n)$. Using a max-heap of size $O(n)$,

we obtain the top-k points from all the lists as follows: insert the top element from each list into the heap, and extract the maximum element from the heap. Then, the next element from the list corresponding to the extracted element is inserted into the heap. Clearly, after extracting k elements, the desired top-k identifiers are obtained. Time required is $O(k \log n)$, which gives the following lemma.

Lemma 14. There exists a $O(n)$ words space data-structure for answering top-k documents with forbidden extension queries in $O(|P^-| \log n + k \log n)$ time.

4.1.2 $O(|P^-| \log \sigma + k)$ Index

In this section, we prove Theorem 5. Note that type A and type B chains can be processed in $O(|P^-| + k)$ time by maintaining separate range reporting data structures (refer to Section 4.1.1). Therefore, in what follows, the emphasis is to obtain type C outputs. Recall that for processing type C chains in Section 4.1.1, we traversed the path from p^+ to p^-, and query the individual data structure at each node. Our idea for more efficient solution is to group together the data structures of the nodes falling on the same heavy path.

Heavy Path Decomposition. We revisit the heavy path decomposition of a tree T, proposed by Harel et al. [60]. For any internal node u, the heaviest child of u is the one having the maximum number of leaves in its subtree (ties broken arbitrarily). The first heavy path of T is the path starting at T's root, and traversing through every heavy node to a leaf. Each off-path subtree of the first heavy path is further decomposed recursively. Thus, a tree with m leaves has m heavy paths. With slight abuse of notation, let Leaf(hp_i) be the leaf where heavy path hp_i ends. Let v be a node on a heavy path and u be a child of v not on that heavy path. We say that the subtree rooted at u *hangs from node* v.

Problem 1. For a tree having m nodes, the path from the root to any node v traverses at most $\log m$ heavy paths.

Heavy Path Tree. We construct the heavy path tree T_H, in which each node corresponds to a distinct heavy path in GST. The tree T_H has n nodes as there are so many heavy paths in GST. For a heavy path hp_i of GST, the corresponding node in T_H is denoted by h_i. All the heavy paths hanging from hp_i in GST are the children of h_i in T_H. Let the first heavy path in the heavy path decomposition of GST be hp_r, and $T_1, T_2, \ldots,$ be the subtrees hanging from hp_r. The heavy path tree T_H is recursively defined as the tree whose root is h_r, representing hp_r, having children $h_1, h_2, \ldots$ with subtrees in T_H resulting from the heavy path decomposition

of $T_1, T_2, \ldots$ respectively. Figure 4.2 illustrates heavy path decomposition of GST and the heavy path tree T_H. Based on the position of a hanging heavy path w.r.t. hp_i in GST, we divide the children of h_i into two groups: left children h_i^l and right children h_i^r. A child heavy path h_j of h_i belongs to h_i^l (resp. h_i^r) if Leaf(hp_j) falls on the left (resp. right) of Leaf(hp_i) in GST. The nodes in h_i^l and h_i^r are stored contiguously in T_H. We traverse the left attached heavy paths of hp_i in GST in top-to-bottom order, include them as the nodes of h_i^l, and place them in left-to-right order as children of h_i in T_H. The h_i^r nodes are obtained by traversing the right attached heavy paths of hp_i in GST in bottom-to-top order, and place them after the h_i^l nodes in T_H in left-to-right order.

Transformed Heavy Path Tree. We transform the heavy path tree T_H into a binary search tree T_H^t. For each node h_i in T_H, we construct a left (resp. right) binary tree $BT_{h_i^l}$ (resp. $BT_{h_i^r}$) for the left children h_i^l (resp. right children h_i^r). Leaves of $BT_{h_i^l}$ (resp. $BT_{h_i^r}$) are the nodes of h_i^l (resp. h_i^r) preserving the ordering in T_H. The binary tree $BT_{h_i^l}$ (resp. $BT_{h_i^r}$) has a path, named left spine (resp. right spine), denoted by LS_{h_i} (resp. RS_{h_i}) containing $\lfloor \log |h_i^l| \rfloor$ (resp. $\lfloor \log |h_i^r| \rfloor$) nodes, denoted by $dl_1, dl_2, \ldots$ (resp. $dr_1, dr_2, \ldots$) in the top-to-bottom order. The right child of dl_i is dl_{i+1}. Left subtree of dl_i is a height balanced binary search tree containing $h_{2^{i-1}}, \ldots, h_{\lfloor 2^i - 1 \rfloor}$ as the leaves and dummy nodes for binarization. Right spine is constructed in similar way, however left child of dr_i is dr_{i+1} and left subtree contains the leaves of h_i^r in a height balanced binary tree. Clearly, the length of LS_{h_i} (resp. RS_{h_i}) is bounded by $\lfloor \log |h_i^l| \rfloor$ (resp. $\lfloor \log |h_i^r| \rfloor$). Subtrees hanging from the nodes of h_i^l and h_i^r are decomposed recursively. See Figure 4.2(c) for illustration. We have the following important property of T_H^t.

Lemma 15. Let u be an ancestor node of v in GST. The path length from u to v is d_{uv}. The node u (resp. v) falls on the heavy path hp_1 (resp. hp_t) and let h_1 (resp. h_t) be the corresponding node in T_H^t. Then, the h_1 to h_t path in T_H^t has $O(\min(d_{uv} \log \sigma, \log^2 n))$ nodes, where σ is the size of the alphabet from which characters in the documents are chosen.

Proof. We first recall from Property 1 that the height of T_H is $O(\log n)$. Since each node in T_H can have at most n children, each level of T_H can contribute to $O(\log n)$ height in T_H^t. Thus, the height of T_H^t is bounded by $O(\log^2 n)$. Hence, the $\log^2 n$ bound in the lemma is immediate. Let $p_1, p_2, \ldots, p_t$ be the segments of the path from u to v traversing heavy paths $hp_1, hp_2, \ldots, hp_t$, where $p_i \in hp_i, 1 \leq i \leq t$. Let $h_1, h_2, \ldots, h_t$ be the corresponding nodes in T_H^t. We show that the number of nodes traversed to reach from h_i to h_{i+1} in T_H^t is

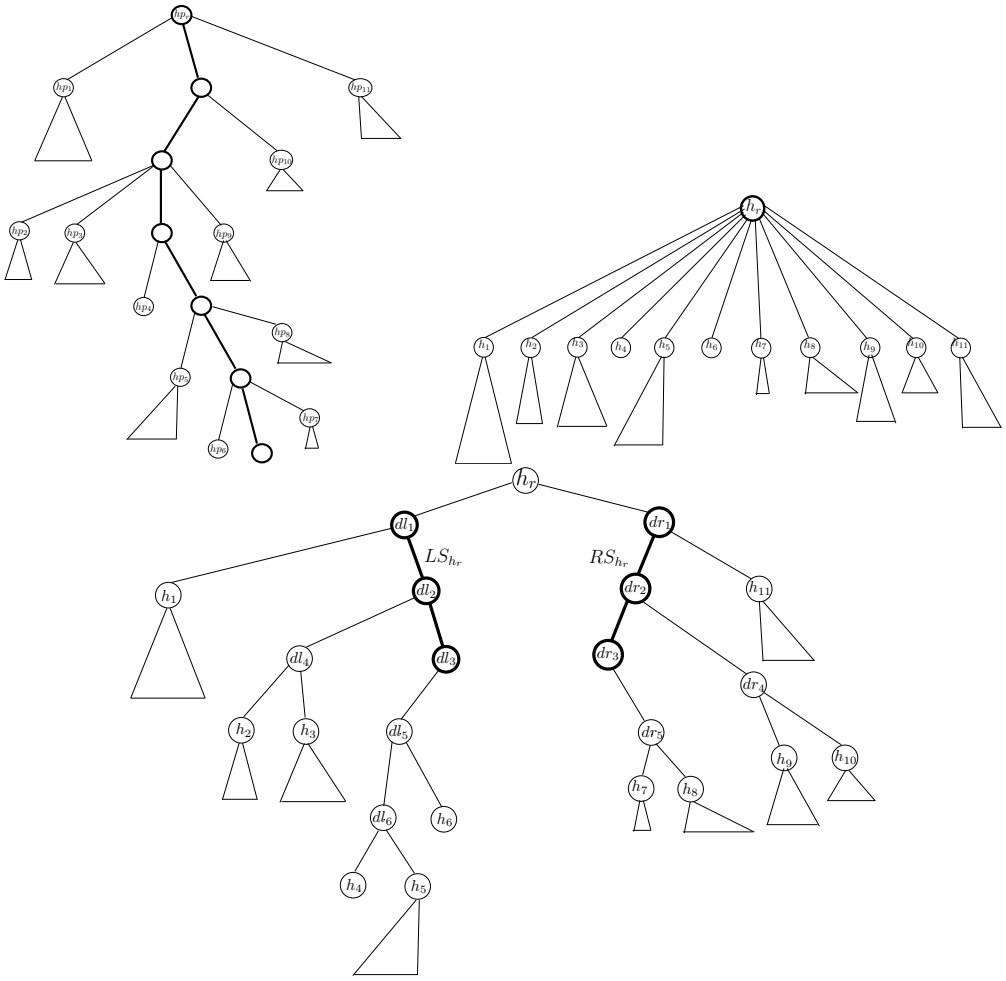

FIGURE 4.2. Heavy path decomposition, heavy path tree T_H, and transformed heavy path tree T_H^t.

$O(|p_i| \log \sigma)$. Without loss of generality, assume h_{i+1} is attached on the left of of h_i and falls in the subtree attached with dl_x on spine LS_{h_i}. We can skip all the subtrees attached to the nodes above dl_x on LS_{h_i}. One node on a heavy path can have at most σ heavy paths as children. Thus, the number of nodes traversed on the spine is $O(|p_i| \log \sigma)$. Within the subtree of the dl_x, we can search the tree to find the desired heavy path node. Since, each node in the GST can have at most σ heavy paths as children, the height of this subtree is bounded by $O(\log \sigma)$. For each p_i, we may need to traverse the entire tree height to locate the desired heavy path, and hence the lemma follows. □

Associating the chains. Let hp_i (resp. hp_j) be the heavy path having i (resp. j) as the leaf node in GST and h_i (resp. h_j) as the corresponding heavy path node in T_H^t. Then, we *associate* chain (i, j) with $lca(h_i, h_j)$ in T_H^t.

Constructing the Index. Our index consists of two components, maximum chain depth structure (MDS) and transformed heavy path structure (THS) defined as follows.

MDS component: Let hp_t be the heavy path in the original heavy path decomposition (i.e., not a dummy heavy path), associated with chain (i, j), $j = \mathsf{next}(i)$. Let, $d_i = \mathsf{depth}(i, \text{Leaf}(hp_t))$ and $d_j = \mathsf{depth}(j, \text{Leaf}(hp_t))$. Define $\mathsf{maxDepth}(i, j) = \max(d_i, d_j)$. Let m_t be the number of chains associated with hp_t. Create two arrays A_t and A'_t, each of length m_t. For each chain (i, j) associated with hp_t, store $\mathsf{doc}(i)$ in the first empty cell of the array A_t, and $\mathsf{maxDepth}(i, j)$ in the corresponding cell of the array A'_t. Sort both the arrays w.r.t the values in A'_t. For each node u lying on hp_t, maintain a pointer to the minimum index x of A such that $A'_t[x] = \mathsf{depth}(u)$. Discard the array A'_t. Finally, build the 1-dimensional sorted range-reporting structure (Fact 5) over A_t. Total space for all t is bounded by $O(n)$ words.

THS component: We construct the transformed heavy path tree T^t_H from GST. Recall that every chain in GST is associated with a node in T^t_H. For each node h_i in T^t_H, we store two arrays, chain source array CS_i and chain destination array CD_i. The arrays CS_i (resp. CD_i) contains the weights (i.e., the document identifier) of all the chains associated with h_i sorted by the start (resp. end) position of the chain in GST. Finally we build the RMQ data structure (Fact 6) RMQ_{CS_i} and RMQ_{CD_i} over CS_i and CD_i respectively. Total space can be bounded by $O(n)$ words.

Fact 6 ([49, 50])**.** By maintaining a $2n + o(n)$ bits structure, range maximum query(RMQ) can be answered in $O(1)$ time (without accessing the array).

Query Answering. Query answering is done by traversing from p^+ to p^- in GST. We start with the following observation.

Observation 1. For every type C chain (i, j), $lca(i, j)$ falls on the p^+ to p^- path in GST.

This observation is crucial to ensure that we do not miss any type C chain in query answering. We consider the following two cases for query answering.

4.1.2.1 p^+ and p^- falls on the same heavy path

In this case, we resort to component MDS for query answering. Assume that p^+ and p^- fall on heavy path hp_t. Note that a chain (i, j) qualifies as an output, iff $\mathsf{maxDepth}(i, j)$ falls within the range

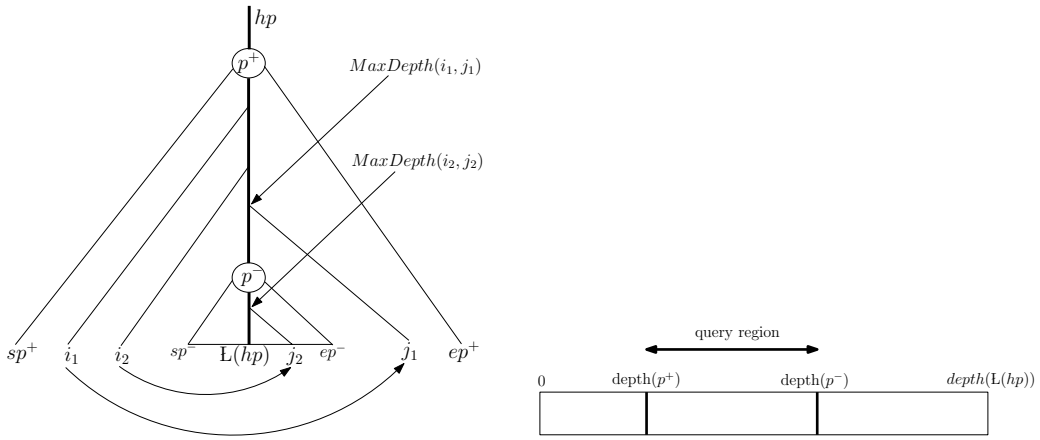

FIGURE 4.3. p^+ and p^- falling on the same heavy path. (a) Chain (i_1, j_1) qualifies since $\mathsf{maxDepth}(i, j) \in [\mathsf{depth}(p^+), \mathsf{depth}(p^-))$. (i_2, j_2) does not qualify. (b) Query range in the 1-dimensional sorted range reporting structure of hp.

$[\mathsf{depth}(p^+), \mathsf{depth}(p^-) - 1]$. See Figure 4.3(a) for illustration. For query answering, follow the pointers from p^+ and p^- to the indexes x and y in the array A_t, and issue the query $\langle x, y-1, k\rangle$ in the corresponding Fact 5 data structure. Note that Type A and Type B outputs can arise. We obtain the following lemma.

Lemma 16. There exists an $O(n)$ words data structure, such that for a top-k forbidden extension query, we can report the top-k Type C leaves in $O(|P^-| + k)$ time when p^+ and p^- falls on the same heavy path.

4.1.2.2 p^+ and p^- falls on different heavy paths

Let $p_1, p_2, \ldots, p_t$ be the path segments of the path from p^+ to p^- traversing heavy paths $hp_1, hp_2, \ldots, hp_t$, where $p_i \in hp_i$, $1 \leq i \leq t$. Let $h_1, h_2, \ldots, h_t$ be the corresponding nodes in T_H^t. In the following subsection, we show how to obtain answers for h_1 through h_{t-1}; we resolve h_t separately. We use the THS component for processing the chains with LCA on $h_1, h_2, \ldots, h_{t-1}$. We start with the following lemma.

Lemma 17. Let (i, j) be a chain associated with a node h_k in T_H^t. If p^- falls on the left (resp. right) subtree of h_k, and $sp^+ \leq i < sp^-$ (resp. $ep^- < j \leq ep^+$), then (i, j) is qualified as an output of the forbidden extension query.

Proof. Recall that chain (i, j) is associated with $h_k = lca(h_i, h_j)$ in T_H^t, where h_i and h_j are the heavy path nodes corresponding to i and j respectively. This implies h_i (resp. h_j) falls on the left (resp. right) subtree of h_k. If p^- falls on the left of hp_k then $j > ep^-$. The added constraint $sp^+ \leq i < sp^-$ ensures that chain (i, j) is either a Type B or a Type C chain, both of which are qualified as an output of the forbidden extension query. The case when p^- falls on the right of h_k is symmetric. □

Lemma 17 allows us to check only the source or destination of a chain based on the position of p^-, and collect the top weighted chains; this is facilitated using the RMQ data structure. We traverse the nodes in T_H^t from p^+ to p^-. At each node h_k, if p^- falls on the left of h_k, we issue a range maximum query within the range $[sp^+, sp^- - 1]$ on RMQ_{CS_k} which gives us the top answer from each node in $O(1)$ time. Note that, $[sp^+, sp^- - 1]$ range needs to be transformed for different RMQ_{CS} structures. We use fractional cascading for the range transformation to save predecessor searching time (refer to Section 4.2 for detailed discussion). Since the height of the tree is $O(\log^2 n)$ (refer to Lemma 15) at any instance, there are at most $O(\log^2 n)$ candidate points. We use the *atomic heap* of Fredman and Willard [52] which allows constant time insertion and delete-max operation when the heap size is $O(\log^2 m)$, where m is the size of the universe. By maintaining each candidate point in the atomic heap, the highest weighted point (among all candidate points) can be obtained in constant time. Also, once the highest weighted point from a heavy path node is obtained, each subsequent candidate point can be obtained and inserted into the the atomic heap in $O(1)$ time. Hence the query time is bounded by the number of nodes traversed in T_H^t. From lemma 15, we obtain that the number of nodes traversed is bounded by $O(\min(|P^-| \log \sigma, \log^2 n))$.

For hp_t, we utilize component MDS. Let r_t be the root of heavy path hp_t. A chain (i, j) qualifies as an output, iff $\mathsf{maxDepth}(i, j)$ falls within the range $[\mathsf{depth}(r_t), \mathsf{depth}(p^-) - 1]$. For query answering, follow the pointers from r_t and p^- to the indexes x and y in the array A_t, and issue the query $\langle x, y - 1, k\rangle$ in the corresponding Fact 5 data structure. Note that Type A and Type B outputs can arise.

From the above discussion, we obtain the following lemma.

Lemma 18. There exists an $O(n)$ words data structure, such that for a top-k forbidden extension query, we can report the top-k Type C leaves in $O(|P^-| \log \sigma + k)$ time when p^+ and p^- falls on different heavy paths.

Combining Lemmas 13, 16, and 18, we obtain the result stated in Theorem 5.

4.2 Range Transformation using Fractional Cascading

We employ the fractional cascading idea of Chazelle et.al [31] for predecessor searching in CS array. Successor searching and CD array are handled in a similar way. The idea is to merge the CS array for siblings and propagate the predecessor information from bottom-to-top. Two arrays are used for this purpose: merged siblings array MS and merged children array MC. Let h_i be an internal node in T_H^t having sibling h_j and two children leaf nodes h_u and h_v. Array MC_u (resp. MC_v) is same as CS_u (resp. CS_v) and stored

in h_u (resp. h_v). The arrays CS_u and CS_v are merged to form a sorted list MS_{uv}. Note that, CS_v values are strictly greater than CS_u; therefore, CS_u and CS_v form two disjoint partitions in MS_{lr} after sorting. We denote the left partition as MS^l_{uv} and the right partition as MS^r_{uv}. We also store a pointer from each value in MS^l_{uv} (MS^r_{uv}) to its corresponding value in MC_u (resp. MC_v). The list MC_i is formed by merging CS_i with every second item from MS_{lr}. With each item x in MC_i, we store three numbers: the predecessor of x in CS_i, the predecessor of x in MS^l_{uv} and the predecessor of x in MS^r_{uv}. Total space required is linear in the number of chains, and is bounded by $O(n)$ words.

Using this data structure, we show how to find predecessor efficiently. Let h_w be an ancestor node of h_z in T^t_H. We want to traverse h_w to h_z path and search for the predecessor of x in CS_i, where h_i is a node on the h_w to h_z path. When we traverse from a parent node h_i to a child node h_j, at first we obtain the predecessor value in parent node using MC_i. If h_j is the left (resp. right) children of h_i, we obtain the predecessor value in MS^l_{jk} (resp. MS^r_{jk}), where h_k is the sibling of h_j. Following the pointer stored at MS^l_{jk} or MS^r_{jk}, we can get the predecessor value at MC_j, and proceed the search to the next level. This way we can obtain the transformed range at each level in $O(1)$ time.

Chapter 5
Succinct Index for Document Retrieval with Forbidden Extension

In this chapter, we propose a succinct space index for the document retrieval with forbidden extension problem. Our result is summarized in the following theorem.

Theorem 6. Let CSA be a *compressed suffix array* on $\mathcal{D}$ of size $|\mathsf{CSA}|$ bits using which we can find the suffix range of a pattern P in $\mathsf{search}(P)$ time, and suffix (or inverse suffix) array value in $\mathsf{t_{SA}}$ time. Also, denote by $|\mathsf{CSA}^*|$ the maximum of the space needed to store the *compressed suffix array* CSA of the concatenated text of all documents, or the total space needed to store the individual CSA of each document. By maintaining CSA and additional $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(n)$ bits structure, we can answer top-k forbidden extension queries in $O(\mathsf{search}(P^-) + k \cdot \mathsf{t_{SA}} \cdot \log^{2+\epsilon} n)$ time

The key idea is to identify some special nodes in the GST, pre-compute the answers for a special node and its descendant special node, and maintain these answers in a data structure. By appropriately choosing the special nodes, the space can be bounded by $O(n)$ bits. Using other additional compressed data structures for document listing [65], we arrive at our claimed result.

We begin by identifying certain nodes in GST as *marked nodes* and *prime nodes* based on a parameter g called *grouping factor* [67]. First, starting from the leftmost leaf in GST, we combine every g leaves together to form a group. In particular, the leaves ℓ_1 through ℓ_g forms the first group, ℓ_{g+1} through ℓ_{2g} forms the second, and so on. We mark the LCA of the first and last leaves of every group. Moreover, for any two marked nodes, we mark their LCA (and continue this recursively). Note that the root node is marked, and the number of marked nodes is at most $2\lceil n/g \rceil$. See Figure 5.1 for an illustration.

Corresponding to each marked node (except the root), we identify a unique node called the prime node. Specifically, the prime node u' corresponding to a marked node u^* is the node on the path from root to u^*, which is a child of the lowest marked ancestor of u^*; we refer to u' as the lowest prime ancestor of u^*. Since the root node is marked, there is always such a node. If the parent of u^* is marked, then u^* is same as u'. Also, for every prime node, the corresponding closest marked descendant (and ancestor) is unique. Therefore number of prime nodes is one less than the number of marked nodes. The following lemma highlights some important properties of marked and prime nodes.

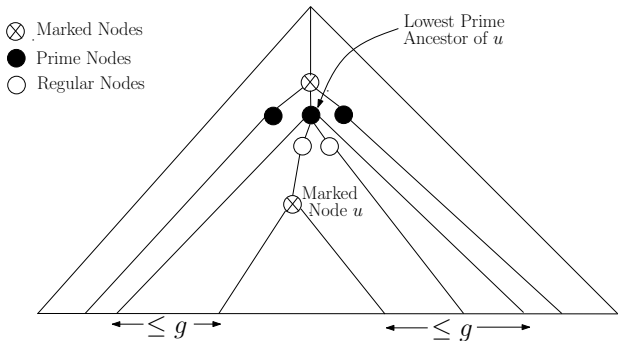

FIGURE 5.1. Marked nodes and Prime nodes with respect to grouping factor g.

Fact 7 ([17, 65])**.** (i) In constant time we can verify whether any node has a marked descendant or not. (ii) If a node u has no marked descendant, then $|\,\mathrm{Leaf}(u)| < 2g$. (iii) If u^* is the highest marked descendant of u, and u is not marked, then $|\,\mathrm{Leaf}(u, u^*)| \leq 2g$. (iv) If u' is the lowest prime ancestor of u^*. Then $|\,\mathrm{Leaf}(u', u^*)| \leq 2g$.

We now present a framework for proving the following lemma.

Lemma 19. Assume the following.

a. The highest marked node u^* and the sequence of prime nodes (if any) on the path from p^+ to p^- can be found in $\mathsf{t}_{\mathsf{prime}}$ time.

b. For any leaf ℓ_i, we can find the corresponding document in t_{DA} time.

c. For any document identifier d and a range of leaves $[sp, ep]$, we can check in $\mathsf{t}_{\in}$ time, whether d belongs in $\{\mathsf{doc}(i) \mid sp \leq i \leq ep\}$, or not.

For any function $f(n)$, such that $f(n) = \Omega(1)$ and $f(n) = o(n)$, by maintaining CSA and additional $O((n/f(n)) \log^2 n)$ bits structures, we can answer top-k forbidden extension queries in $O(\mathsf{search}(P^-) + \mathsf{t}_{\mathsf{prime}} + k \cdot f(n) \cdot (\mathsf{t}_{\mathsf{DA}} + \mathsf{t}_{\in}))$ time.

Creating the Index. First we maintain a full-text index CSA on the document collection $\mathcal{D}$. Let $g_\kappa = \lceil \kappa \cdot f(n) \rceil$, where κ is a parameter to be defined later. We begin by marking nodes in the GST as marked and prime nodes, as defined previously, based on g_κ. Consider any prime node u, and let $u^\uparrow$ and $u^\downarrow$ be its nearest marked ancestor and descendant (both of which are unique) respectively. We compute the arrays $\mathsf{list}_\kappa(u^\uparrow, u)$ and $\mathsf{list}_\kappa(u, u^\downarrow)$, each sorted by increasing importance (i.e., document identifier). The arrays are maintained in the node u w.r.t grouping factor g_κ. Note that explicitly maintaining each array requires $O(\kappa \log n)$ bits. Space required in bits for all prime nodes w.r.t g_κ can be bounded by $O((n/g_\kappa)\kappa \log n)$ i.e.,

by $O((n/f(n)) \log n)$ bits. We maintain this data-structure for $\kappa = 1, 2, 4 \ldots, D$. Total space is bounded by $O((n/f(n)) \log^2 n)$ bits.

Querying Answering. For a top-k forbidden extension query $\langle P^+, P^-, k \rangle$, we begin by locating the suffix ranges $[sp^+, ep^+]$ and $[sp^-, ep^-]$ of the patterns P^+ and P^- respectively; this can be achieved in time bounded by $\mathsf{search}(P^-)$ using the CSA. If the suffix ranges are the same, then clearly every document containing P^+ also contains P^-, and the top-k list is empty. So, moving forward, we assume otherwise. Note that it suffices to obtain a k-candidate set of size $O(k \cdot f(n))$ in the time of Lemma 19.

Let $k' = \min\{D, 2^{\lceil \log k \rceil}\}$. Note that $k \leq k' < 2k$. Moving forwards, we talk of prime and marked nodes w.r.t grouping factor $g' = \lceil k' f(n) \rceil$. We can detect the presence of marked nodes below p^+ and p^- in constant time using Fact 7. Let the prime nodes on the path from p^+ to p^- be $u_1, u_2, \ldots, u_t$ in order of depth. Possibly, $t = 0$. For each prime node $u_{t'}$, $1 \leq t' \leq t$, we denote by $u_{t'}^{\uparrow}$ and $u_{t'}^{\downarrow}$, the lowest marked ancestor (resp. highest marked descendant) of the $u_{t'}$. We have the following cases.

Case 1. We consider the following two scenarios: (i) $\mathsf{GST}(p^+)$ does not contain any marked node, and (ii) $\mathsf{GST}(p^-)$ contains a marked node, but the path from p^+ to p^- does not contain any prime node. In either case, $|\,\mathrm{Leaf}(p^+, p^-)| \leq 2g'$ (refer to Fact 7). The documents corresponding to these leaves constitute a k-candidate set, and can be found in $O(g' \cdot \mathsf{t_{DA}})$ time i.e., in $O(k \cdot f(n) \cdot \mathsf{t_{DA}})$ time. Now, for each document d, we check whether $d \in \{\mathsf{doc}(i) \mid i \in [sp^-, ep^-]\}$, which requires additional $O(g' \cdot \mathsf{t}_{\in})$ time. Total time can be bounded by $O(g' \cdot (\mathsf{t_{DA}} + \mathsf{t}_{\in}))$ i.e., by $O(k \cdot f(n) \cdot (\mathsf{t_{DA}} + \mathsf{t}_{\in}))$.

Case 2. If the path from p^+ to p^- contains a prime node, then let u^* be the highest marked node. Possibly, $u^* = p^+$. Note that $u_1^{\uparrow}$ is same as u^*, and that $u_t^{\downarrow}$ is either p^- or a node below it. For any t', clearly $\mathsf{list}_{k'}(u_{t'}, u_{t'}^{\downarrow})$ and $\mathsf{list}_{k'}(u_{t'}^{\uparrow}, u_{t'})$ are mutually disjoint. Similar remarks hold for the lists stored at two different prime nodes t' and t'', $1 \leq t', t'' \leq t$. Furthermore, let d be an identifier in one of the lists corresponding to $u_{t'}$. Clearly there is no leaf $\ell_j \in \mathsf{GST}(p^-)$, such that $\mathsf{doc}(j) = d$. We select the top-k' document identifiers from the stored lists (arrays) in the prime nodes u_1 through u_t. Time, according to the following fact, can be bounded by $O(t + k)$.

Fact 8 ([22, 51])**.** Given m sorted integer arrays, we can find the k largest values from all these arrays in $O(m + k)$ time.

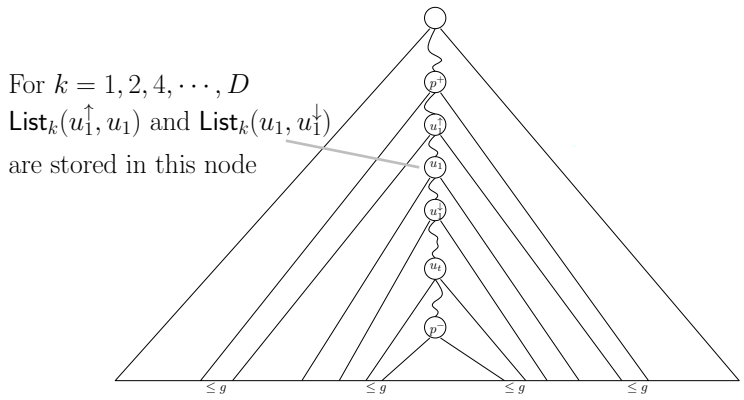

FIGURE 5.2. Illustration of storage scheme and retrieval at every prime node w.r.t grouping factor g. Left and right fringes in Leaf$(p^+ \setminus u_1^{\uparrow})$ and Leaf$(u_t^{\downarrow} \setminus p^-)$ are bounded above by g'.

Now, we consider the *fringe leaves* Leaf(p^+, u^*) and Leaf(u_t, p^-), both of which are bounded above by $2g'$ (refer to Fact 7). The ranges of the these leaves are found in constant time using the following result of Sadakane and Navarro [110].

Lemma 20 ([110])**.** An m node tree can be maintained in $O(m)$ bits such that given a node u, we can find $[sp(u), ep(u)]$ in constant time.

The relevant documents corresponding to these fringe leaves can be retrieved as in Case 1. Clearly, these fringe documents along with the k documents obtained from the stored lists constitute our k-candidate set. Time required can be bounded by $O(t + k + g' \cdot (\mathsf{t_{DA}} + \mathsf{t_{\in}}))$ i.e, by $O(t + k \cdot f(n) \cdot (\mathsf{t_{DA}} + \mathsf{t_{\in}}))$.

Note that $t \leq \mathsf{depth}(p^-) \leq |P^-| = O(\mathsf{search}(P^-))$, and Lemma 19 follows. □

We are now equipped to prove Theorem 6. First, the highest marked node and the t prime nodes from p^+ to p^- are obtained using Lemma 21 in $O(\log n + t)$ time. Maintain the data-structure of this lemma for with $\kappa = 1, 2, 4, \ldots, D$. Space can be bounded by $O(\frac{n}{f(n)} \log n)$ bits. Computing $\mathsf{doc}(i)$ is achieved in $\mathsf{t_{SA}}$ time, according to Lemma 22. Checking whether a document d belongs in a contiguous range of leaves is achieved in $O(\mathsf{t_{SA}} \cdot \log \log n)$ using Lemma 23. Theorem 6 is now immediate by choosing $f(n) = \log^2 n$.

Lemma 21. By maintaining $O((n/g_\kappa) \log n)$ bits in total, we can retrieve the highest marked node, and all t prime nodes, both w.r.t grouping factor $g_\kappa = \lceil \kappa \cdot f(n) \rceil$, that lie on the path from p^+ to p^- in time bounded by $O(\log n + t)$.

Proof. We use the following result of Patil et al. [105]: a set of n three-dimensional points (x, y, z) can be stored in an $O(n \log n)$ bits data structure, such that for a three-dimensional dominance query $\langle a, b, c \rangle$, in

$O(\log n + t)$ time, we can report all t points (x, y, z) that satisfies $x \leq a$, $y \geq b$, and $z \geq c$ with outputs reported in the sorted order of z coordinate.

For each prime node w, we maintain the point $(L_w, R_w, |path(w)|)$ in the data structure above, where L_w and R_w are the leftmost and the rightmost leaves in $\mathsf{GST}(w)$. Total space in bits can be bounded by $O((n/g_\kappa) \log n)$ bits. The t prime nodes that lie on the path from p^+ to p^- are retrieved by querying with $\langle sp^- - 1, ep^- + 1, |P^+| \rangle$. Time can be bounded by $O(\log n + t)$. Likewise, we maintain a structure for marked nodes. Using this, we can obtain the highest marked node in $O(\log n)$ time. □

Lemma 22. Given a CSA, the document array can be maintained in additional $n + o(n)$ bits such that for any leaf ℓ_i, we can find $\mathsf{doc}(i)$ in $\mathsf{t_{SA}}$ time i.e., $\mathsf{t_{DA}} = \mathsf{t_{SA}}$.

Proof. We use the following data-structure [56, 90]: a bit-array $\mathsf{B}[1 \ldots m]$ can be encoded in $m + o(m)$ bits, such that $rank_{\mathsf{B}}(q, i) = |\{j \in [1..i] \mid \mathsf{B}[j] = q\}|$ can be found in $O(1)$ time.

Consider the concatenated text T of all the documents which has length n. Let B be a bit array of length n such that $\mathsf{B}[i] = 1$ if a document starts at the position i in the text T. We maintain a $rank$ structure on this bit-array. Space required is $n + o(n)$ bits. We find the text position j of ℓ_i in $\mathsf{t_{SA}}$ time. Then $\mathsf{doc}(i) = rank_{\mathsf{B}}(1, j)$, and is retrieved in constant time. Time required can be bounded by $\mathsf{t_{SA}}$. □

Lemma 23. Given the suffix range $[sp, ep]$ of a pattern P and a document identifier d, by maintaining CSA and additional $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits structures, in $O(\mathsf{t_{SA}} \log \log n)$ time we can verify whether $d \in \{\mathsf{doc}(i) \mid i \in [sp, ep]\}$, or not.

Proof. Number of occurrences of d in a suffix range $[sp, ep]$ is given by $rank_{\mathsf{DA}}(d, ep) - rank_{\mathsf{DA}}(d, sp - 1)$. Space and time complexity is due to the following result of Hon et al. [65]: the document array DA can be simulated using CSA and additional $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits structures to support $rank_{\mathsf{DA}}$ operation in $O(\mathsf{t_{SA}} \log \log n)$ time. □

Chapter 6
Document Retrieval Using Shared Constraint Range Reporting

6.1 Problem Formulation and Motivation

In this chapter, we transform the document retrieval problem into a new variation of orthogonal range reporting problem, namely **Shared Constraint Range Reporting**. We also show that several well known text indexing problems can be solved using our proposed shared constraint range reporting data structure. At first we present the document retrieval framework proposed by Hon et. al. [66].

In the ranked document retrieval problem, we have a collection $\mathcal{D} = \{\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_D\}$ of documents (strings) of total length N. Define $score(P, \mathsf{T}_d)$, the *score* of a document T_d with respect to a pattern P, which is a function of the locations of all occurrences of P in T_d. Then our goal is to preprocess $\mathcal{D}$ and maintain a structure such that, given a query pattern P and a threshold c, all those documents T_i with $score(P, \mathsf{T}_i) \geq c$ can be retrieved efficiently in sorted order of their score. We construct the GST for $\mathcal{D}$. For each node u in the GST of $\mathcal{D}$, let $depth(u)$ be the number of nodes on the root to u path, and prefix(u) be the string obtained by concatenating all the edge labels of the path. Let u_P be the locus node of pattern P. We traverse the GST in pre-order and annotate each node with its pre-order rank. Then the subtree of u_P is represented by a contiguous pre-order range. Furthermore, we mark each node with document identifiers. A leaf node is marked with a document $d \in D$ if the suffix represented by the leaf belongs to T_d. An internal node u is marked with d if it is the lowest common ancestor of two leaves marked with d. A node can be marked with multiple documents. For each node u and each of its marked documents d, define a link to be a quadruple $(origin, target, doc, Score)$, where origin=$u$, target is the lowest proper ancestor of u marked with d, $doc = d$ and $Score = score(prefix(u), \mathsf{T}_d)$. Total number of such links are bounded by $O(N)$. For each document d that contains a pattern P, there is a unique link whose origin is in the subtree of u_P and whose target is a proper ancestor of u_P . The score of the link is exactly the score of d with respect to P.

Based on the above linking framework, the ranked document retrieval problem can be reduced to the problem of finding the highest scored links (according to its score) stabbed by u_P, In other words, we have to find all the links for which (i) pre-order rank of origin fall in the range of the subtree of u_P , (ii) pre-order rank of target is less than the pre-order rank of u_P, and (iii) score is greater than threshold c. Which is

precisely the interval stabbing problem: given a collection of N intervals (y_i, x_i) with weights w_i and a query (a, b, c), output all the intervals such that $y_i \leq a \leq x_i \leq b$ and $w_i \geq c$. In this chapter, we show how to solve this problem using shared constraint range reporting data structure. We start with a brief introduction of orthogonal range searching.

Orthogonal range searching is one of the central data structure problems which arises in various fields. Many database applications benefit from the structures which answer range queries in two or more dimensions. Goal of orthogonal range searching is to design a data structure to represent a given set of N points in d-dimensional space, such that given an axis-aligned query rectangle, one can efficiently list all points contained in the rectangle. One simple example of orthogonal range searching data structure represents a set of N points in 1-dimensional space, such that given a query interval, it can report all the points falling within the interval. A balanced binary tree taking linear space can support such queries in optimal $O(\log N + K)$ time. Orthogonal range searching gets harder in higher dimensions and with more constraints. The hardest range reporting, yet having a linear-space and optimal time (or query I/Os in external memory) solution is the three dimensional dominance reporting query, also known as $(1, 1, 1)$ query [1] with one-sided constraint on each dimension. Here the points are in three-dimensions and the query asks to report all those points within an input region $[q_1, \infty) \times [q_2, \infty) \times [q_3, \infty)$. A query of the form $[q_1, q_2] \times [q_3, \infty)$ is known as $(2, 1)$ query, which can be seen as a particular case of $(1, 1, 1)$ query. However, four (and higher) sided queries are known to be much harder and no linear-space solution exists even for the simplest two dimensional case which matches the optimal query time of three dimensional dominance reporting. In Word-RAM model, the best result (with optimal query time) is $O(N \log^{\epsilon} N)$ words [27], where N is the number of points and $\epsilon > 0$ is an arbitrary small positive constant. In external memory, there exists an $\Omega(N \log N / \log \log_B N)$-space lower bound (and a matching upper bound) for any two-dimensional four-sided range reporting structure with optimal query I/Os [9]. Therefore, we cannot hope for a linear-space or almost-linear space structure with $O(\log_B N + K/B)$ I/Os for orthogonal range reporting queries with four or more constraints. The model of computation we assume is a unit-cost RAM with word size logarithmic in n. In RAM model, random access of any memory cell and basic arithmetic operations can be performed in constant time.

Motivated by database queries with constraint sharing and several well known problems (More details in Section 6.2), we study a special four sided range reporting query problem, which we call as the *Shared-Constraint Range Reporting* (SCRR) problem. Given a set $\mathcal{P}$ of N three dimensional points, the query input

is a triplet (a, b, c), and our task is to report all those points within a region $[a, b] \times (-\infty, a] \times [c, \infty)$. We can report points within any region $[a, b] \times (-\infty, f(a)] \times [c, \infty)$, where $f(\cdot)$ is a pre-defined monotonic function (using a simple transformation). The query is four sided with only three independent constraints. Many applications which model their formulation as 4-sided problems actually have this sharing among the constraints and hence better bounds can be obtained for them using SCRR data structures. Formally, we have the following definition.

Problem 4. A SCRR query $Q_{\mathcal{P}}(a, b, c)$ on a set $\mathcal{P}$ of three dimensional points asks to report all those points within the region $[a, b] \times (-\infty, a] \times [c, \infty)$.

The following theorems summarize our main results.

Theorem 7 (SCRR in Ram Model)**.** There exists a linear space RAM model data structure for answering SCRR queries on the set $\mathcal{P}$ in $O(\log N + K)$ time, where $N = |\mathcal{P}|$ and K is the output size.

Theorem 8 (Linear space SCRR in External Memory)**.** SCRR queries on the set $\mathcal{P}$ can be answered in $O(\log_B N + \log\log N + K/B)$ I/Os using an $O(N)$-word structure, where $N = |\mathcal{P}|$, K is the output size and B is the block size.

Theorem 9 (Optimal Time SCRR in External Memory)**.** SCRR queries on the set $\mathcal{P}$ can be answered in optimal $O(\log_B N + K/B)$ I/Os using an $O(N \log\log N)$-word structure, where $N = |\mathcal{P}|$, K is the output size and B is the block size.

Our Approach: Most geometric range searching data structures use point partitioning scheme with appropriate properties, and recursively using the data structure for each partition. Our chapter uses a novel approach of partitioning the points which seem to fit SCRR problem very well. Our data structure uses rank-space reduction on the given point-set, divide the SCRR query data structure based on small and large output size, takes advantage of some existent range reporting data structure to obtain efficient solution and then bootstrap the data structure for smaller ranges.

Related Work: The importance of two-dimensional three-sided range reporting is mirrored in the number of publications on the problem. The general two-dimensional orthogonal range searching has been extensively studied in internal memory [2, 3, 4, 28, 29, 30, 25, 12]. The best I/O model solution to the three-sided range reporting problem in two-dimensions is due to Arge et al. [9], which occupies linear space and answers

queries in $O(\log_B N + K/B)$ I/Os. Vengroff and Vitter [115] addressed the problem of dominance reporting in three dimensions in external memory model and proposed $O(N \log N)$ space data structure that can answer queries in optimal $O(\log_B N + K/B)$ I/Os. Recently, Afshani [1] improved the space requirement to linear space while achieving same optimal I/O bound. For the general two-dimensional orthogonal range reporting queries in external memory settings Arge et al. [9] gave $O((N/B)\log_2 N/\log_2 \log_B N)$ blocks of space solution achieving optimal $O(\log_B N + K/B)$ I/Os. Another external memory data structure is by Arge et al. [8] where the query I/Os is $O(\sqrt{N/B} + k/B)$ and the index space is linear. In the case when all points lie on a $U \times U$ grid, the data structure of Nekrich [102] answers range reporting queries in $O(\log \log_B U + K/B)$ I/Os. In [102] the author also described data structures for three-sided queries that use $O(N/B)$ blocks of space and answer queries in $O(\log \log_B U + K/B)$ I/Os on a $U \times U$ grid and $O(\log_B^{(h)} N)$ I/Os on an $N \times N$ grid for any constant $h > 0$. Very recently, Larsen and Pagh [79] showed that three-sided point reporting queries can be answered in $O(1 + K/B)$ I/Os using $O(N/B)$ blocks of space.

Outline: In section 6.2, we show how SCRR arises in database queries and relate SCRR problem to well known problems of colored range reporting, ranked document retrieval, range skyline queries and two-dimensional range reporting. In section 6.3 we discuss rank-space reduction of the input point-set to make sure no two points share the same x-coordinate. In section 6.4 we introduce a novel way to partition the point-set for answering SCRR queries which works efficiently for larger output size. Section 6.5 explains how to answer SCRR queries for smaller output size. Using these two data structures, section 6.6 obtains linear space and $O(\log N + K)$ time data structure for SCRR queries in RAM model thus proving theorem 7. Section 6.7 discusses SCRR queries in external memory, which includes a linear space but sub-optimal I/O and an optimal I/O but sub-optimal space data structures.

6.2 Applications

In this section, we show application of SCRR in database queries and list some of the well known problems, which could be directly reduced to SCRR. We start with two simple examples to illustrate shared constraint queries in database:

1. National Climatic Data Center contains data for various geographic locations. Sustained wind speed and gust wind speed are related to the mean wind speed for a particular time. Suppose we want to retrieve the stations having $(sustained\_wind\_speed, gust\_wind\_speed)$ satisfying criteria 1: $mean\_wind\_speed <$

$sustained\_wind\_speed < max\_wind\_speed$ and criteria 2: $gust\_wind\_speed < mean\_wind\_speed$. Here $mean\_wind\_speed$ and max_wind_speed comes as query parameters. Note that both these criteria have one constraint shared, thus effectively reducing number of independent constraints by one. By representing each station as the 2-dimensional point ($sustained\_wind\_speed$, $gust\_wind\_speed$), this query translates into the orthogonal range query specified by the (unbounded) axis-aligned rectangle $[mean\_wind\_speed : max\_wind\_speed]$ $(-\infty : mean\_wind\_speed]$.

2. Consider the world data bank which contains data for Gross domestic product (gdp), and we are interested in those countries that have gdp within the range of minimum and maximum gdp among all countries and gdp growth is greater than certain proportion of the minimum gdp. Our query might look like: $min\_gdp < gdp < max\_gdp$ and $c \times min\_gdp < gdp\_growth$, where c is a constant. Here min_gdp and max_gdp comes as query parameters. The constraint on gdp_growth is proportional to the lower constraint of gdp, which means the number of independent constraint is only two. This query can be similarly converted to orthogonal range reporting problem by representing each country as the point (gdp, gdp_growth), and asking to report all the points contained in the (unbounded) axis-aligned rectangle $[min\_gdp : max\_gdp]$ $[c \times min\_gdp : \infty)$.

We can take advantage of such sharing among constraints to construct more effective data structure for query answering. This serves as a motivation for SCRR data structures. Below we show the relation between SCRR and some well known problems.

Colored Range Reporting: In colored range reporting, we are given an array A, where each element is assigned a color, and each color has a priority. For a query $[a, b]$ and a threshold c (or a parameter K) we have to report all distinct colors with priority $\geq c$ (or K colors with highest priority) within $A[a, b]$ [73]. We use the chaining idea by muthukrishnan [92] to reduce the colored range reporting to SCRR problem.

We map each element $A[i]$ to a weighted point (x_i, y_i) such that (1) $x_i = i$, (2) y_i is the highest $j < i$ such that both $A[i]$ and $A[j]$ have the same color (if such a y_i does not exist then $y_i = -\infty$) and (3) its weight w_i is same as the priority of color associated with $A[i]$. Then, the colored range reporting problem is equivalent to the following SCRR problem: report all points in $[a, b] \times (-\infty, a)$ with weight $\geq c$. By maintaining a additional linear space structure, for any given a, b and K, a threshold c can be computed in constant time such that number of colors reported is at least K and at most $\Omega(K)$ (we defer details to the full version).

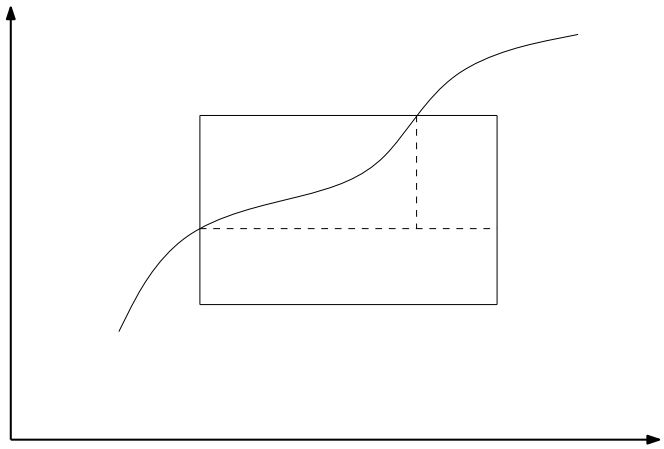

FIGURE 6.1. Special Two-dimensional Range Reporting Query.

Then, by finding the Kth color with highest color among this (using selection algorithm) and filtering out colors with lesser priority, we shall obtain the top-K colors in additional $O(K/B)$ I/Os or $O(K)$ time.

Range Skyline Queries: Given a set S of N points in two-dimensions, a point (x_i, y_i) is said to be dominated by a point (x_j, y_j) if $x_i < x_j$ and $y_i < y_j$. Skyline of S is subset of S which consists of all the points in S which are not dominated by any other point in S. In Range-Skyline problem, the emphasis is to quickly generate those points within a query region R, which are not dominated by any other point in R. There exists optimal solutions in internal as well as external memory models for the case where R is a three-sided region of the form $[a, b] \times [c, +\infty)$ [74, 23].

We can reduce the range skyline query to SCRR by mapping each two-dimensional input point $p_i = (x_i, y_i)$ to a three-dimensional point x'_i, y'_i, z'_i as follows: (1) $x'_i = x_i$, (2) y'_i is the the x-coordinate of the leftmost point dominating p_i and (3) $z'_i = y_i$. Then range skyline query with three-sided region $[a, b] \times [c, +\infty)$ as input can be answered by reporting the output of SCRR query $[a, b] \times (-\infty, a] \times [c, +\infty)$.

Two-dimensional Range Reporting: Even though general four-sided queries are known to be hard as noted earlier, we can efficiently answer "special" four-sided queries efficiently. Any four-sided query with query rectangle R with one of its corners on the line $x = y$ can be viewed as a SCRR query. In fact any query rectangle R which intersect with $x = y$ line (or a predefined monotonic curve) can be reduced to SCRR (Figure 6.1).

6.3 Rank-Space Reduction of Points

We use rank-space reduction on the given point-set. Although rank-space reduction does not save any space for our data structure, it helps to avoid predecessor/successor search while querying and facilitate our partitioning technique. Without loss of generality, we assume that the points $p_i = (x_i, y_i, z_i) \in \mathcal{P}$ satisfy the following conditions: $x_i \leq x_{i+1}$ for all $i \in [1, N-1]$ and also $y_i \leq x_i$ for all $i \in [1, N]$. Note that $x_i \leq x_{i+1}$ can be ensured by sorting the point-set with respect to their x-coordinates and any point not satisfying $y_i \leq x_i$ can not be answer of our SCRR query, so we can remove them from consideration. In this section, we describe how to transform each point $p_i = (x_i, y_i, z_i) \in \mathcal{P}$ to a point $p'_i = (x'_i, y'_i, z'_i) \in \mathcal{P}'$ with the following additional properties guaranteed:

- Points in $\mathcal{P}'$ are on an $[1, N] \times [1, N] \times [1, N]$ grid (i.e., $x_i, y_i, z_i \in [1, N]$)
- $x'_i < x'_{i+1}$ for all $i \in [1, N-1]$. If $y_i \leq y_j$ (resp., $z_i \leq z_j$), then $y'_i \leq y'_j$ (resp., $z'_i \leq z'_j$) for all $i, j \in [1, N-1]$.

Such a mapping is given below: (1) The x-coordinate of the transformed point is same as the rank of that point itself. i.e., $x'_i = i$ (ties are broken arbitrarily), (2) Let $y_i \in (x_{k-1}, x_k]$, then $y'_i = k$, (3) Replace each z_i by the size of the set. i.e., $z'_i = \{j | z_j \leq z_i, j \in [1, N]\}$. We now prove the following lemma.

Lemma 24. If there exists an $S(N)$-space structure for answering SCRR queries on $\mathcal{P}'$ in optimal time in RAM model (or I/Os in external memory), then there exists an $S(N) + O(N)$-space structure for answering SCRR queries on $\mathcal{P}$ in optimal time (or I/Os).

Proof. Assume we have an $S(N)$ space structure for SCRR queries on $\mathcal{P}'$. Now, whenever a query $Q_{\mathcal{P}}(a, b, c)$ comes, our first task is to identify the parameters a', b' and c' such that a point p_j is an output of $Q_{\mathcal{P}}(a, b, c)$ if and only if p'_j is an output of $Q_{\mathcal{P}}(a', b', c')$ and vice versa. Therefore, if point p'_j is an output for $Q_{\mathcal{P}}(a', b', c')$, we can simply output p_j as an answer to our original query. Based on our rank-space reduction, a', b' and c' are given as follows: (1) $x_{a'-1} < a \leq x_{a'}$ (assume $x'_0 = 0$), (2) $x_{b'} \leq b < x_{b'+1}$ (assume $x'_{N+1} = N + 1$), (3) Let z_j be the successor of c, then $c' = z'_j$.

By maintaining a list of all points in $\mathcal{P}$ in the sorted order of their x-coordinate values (along with a B-tree or binary search over it), we can compute a' and b' in $O(\log N)$ time(or $O(\log_B N)$ I/Os). Similarly, c' can also be computed using another list, where the points in $\mathcal{P}$ are arranged in the sorted order of z-coordinate

value. The space occupancy of this additional structure is $O(N)$. Notice that this extra $O(\log N)$ time or $O(\log_B N)$ I/Os is optimal if we do not assume any thing about the coordinate values of points in $\mathcal{P}$. $\square$

6.4 The Framework

In this section we introduce a new point partitioning scheme which will allow us to reduce the SCRR query into a logarithmic number of disjoint planar 3-sided or three dimensional dominance queries. From now onwards, we assume points in $\mathcal{P}$ to be in rank-space (Section 6.3). We begin by proving the result summarized in following theorem.

Lemma 25. By maintaining an $O(|\mathcal{P}|)$-word structure, any SCRR query $Q_{\mathcal{P}}(\cdot,\cdot,\cdot)$ can be answered in $O(\log^2 N + K)$ time in the RAM model, where K is the output size.

For simplicity, we treat each point $p_i \in \mathcal{P}$ as a weighted point (x_i, y_i) in an $[1, N] \times [1, N]$ grid with z_i as its weight. The proposed framework utilizes divide-and-conquer technique based on the following partitioning schemes:

- *Oblique Slabs*: We partition the $[1, N] \times [1, N]$ grid into multi-slabs $OS_0, OS_1, ..., OS_{\lceil \log N \rceil}$ induced by lines $x = y + 2^i$ for $i = 0, 1, ...\lceil \log N \rceil$ as shown in figure 6.2(a). To be precise, OS_0 is the region between the lines $x = y$ and $x = y + 1$ and OS_i for $i = 1, 2, 3, ..., \lceil \log N \rceil$ be the region between lines $x = y + 2^{i-1}$ and $x = y + 2^i$.
- *Step Partitions:* Each slab OS_i for $i = 1, 2, ...$ is further divided into regions with right-angled triangle shape (which we call as *tiles*) using axis parallel lines $x = (2^{(i-1)} * (1 + j))$ and $y = 2^{(i-1)} * j$ for $j = 1, 2, ...$ as depicted in Figure 6.2(b). OS_0 is divided using axis parallel lines $x = j$ and $y = j$ for $j = 1, 2, ...$ Notice that the (axis parallel) boundaries of these triangles within any particular oblique slab looks like a step function.

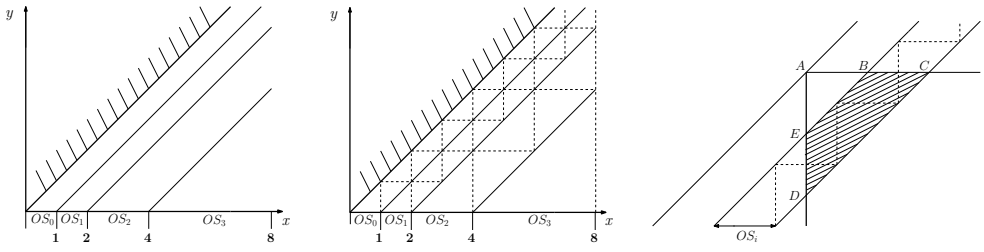

FIGURE 6.2. Point partitioning schemes: (a) Oblique slabs (b) Step partitions.

Our partitioning scheme ensures property summarized by following lemma.

Lemma 26. Any region $[a, b] \times [1, a]$ intersects with at most $O(\log N)$ tiles.

Proof. Let ϕ_i be the area of a tile in the oblique slab OS_i. Note that $\phi_0 = \frac{1}{2}$ and $\phi_i = \frac{1}{2}(2^{i-1})^2$ for $i \in [1, \lceil \log N \rceil]$. And let A_i be the area of the overlapping region between OS_i and the query region $[a, b] \times [1, a]$. Now our task is to simply show A_i/ϕ_i is a constant for all values of i. Assume $b = n$ in the extreme case. Then the overlapping region between OS_i and $[a, n] \times [1, a]$ will be trapezoid in shape and its area is given by $\phi_{i+1} - \phi_i$ (See Figure 6.2(c) for a pictorial proof). Therefore number of tiles needed for covering this trapezoidal region is $A_i/\phi_i = O(1)$. Which means the entires region can be covered by $O(\log N)$ tiles ($O(1)$ per oblique slab). □

In the light of the above lemma, a given SCRR query $Q_{\mathcal{P}}(a, b, c)$ can be decomposed into $O(\log N)$ subqueries of the type $Q_{\mathcal{P}_t}(a, b, c)$. Here $\mathcal{P}_t$ be the set of points within the region covered by a tile t. In the next lemma, we show that each of the $Q_{\mathcal{P}_t}(a, b, c)$ can be answered in optimal time (i.e., $O(\log |\mathcal{P}_t|)$ plus $O(1)$ time per output). Therefore, in total $O(N)$-space, we can maintain such structures for every tile t with at least one point within it. Then by combining with the result in lemma 26, the query $Q_{\mathcal{P}}(a, b, c)$ can be answered in $O(\log N * \log N + K) = O(\log^2 N + K)$ time, and lemma 25 follows.

Lemma 27. Let $\mathcal{P}_t$ be the set of points within the region covered by a tile t. Then a SCRR query $Q_{\mathcal{P}_t}(a, b, c)$ can be answered in $O(\log |\mathcal{P}_t| + k)$ time using a linear-space (i.e., $O(|\mathcal{P}_t|)$ words) structure, where k is the output size.

Proof. The first step is to maintain necessary structure for answering all possible axis aligned three-dimensional dominance queries over the points in $\mathcal{P}_t$, which takes linear-space (i.e., $O(|\mathcal{P}_t|)$ words or $O(|\mathcal{P}_t| \log |\mathcal{P}_t|)$ bits). Let α and β be the starting and ending position of the interval obtained by projecting tile t to x-axis (see Figure 6.3). Then if the tile t intersects with the query region $[a, b] \times [1, a]$, then we have the following cases (see Figure 6.3):

1. $\alpha \leq a \leq \beta \leq b$: In this case, all points in $p_i \in \mathcal{P}_t$ implicitly satisfy the condition $x_i \leq b$. Therefore $Q_{\mathcal{P}_t}(a, b, c)$ can be obtained by a three sided query with $[a, N] \times [1, a] \times [c, N]$ as the input region or a two dimensional dominance query with $[a, N] \times [1, N] \times [c, N]$ as the input region (Figure 6.3(a)).
2. $a \leq \alpha \leq \beta \leq b$: In this case, all points in $p_i \in \mathcal{P}_t$ implicitly satisfy the condition $x_i \in [a, b]$. Therefore, $Q_{\mathcal{P}_t}(a, b, c)$ can be obtained by a two dimensional dominance query with $[1, N] \times [1, a] \times [c, N]$ as the input region (Figure 6.3(b)).

3. $a \leq \alpha \leq b \leq \beta$: In this case, all points in $p_i \in \mathcal{P}_t$ implicitly satisfy the condition $x_i \geq a$. Therefore $Q_{\mathcal{P}_t}(a, b, c)$ can be obtained by a three dimensional dominance query with $[1, b] \times [1, a] \times [c, N]$ as the input region (Figure 6.3(c)).
4. $\alpha \leq a \leq b \leq \beta$: Notice that the line between the points (a, a) and (b, a) are completely outside (and above) the tile t. Therefore, all points in $p_i \in \mathcal{P}_t$ implicitly satisfy the condition $y_i \leq a$. Therefore, $Q_{\mathcal{P}_t}(a, b, c)$ can be obtained by a three sided query with $[a, b] \times [1, N] \times [c, N]$ as the input region (Figure 6.3(d)).

Note that tiles can have two orientations. We have discussed four cases for one of the tile orientations. Cases for other orientation is mirror of the above four cases and can be handled easily. □

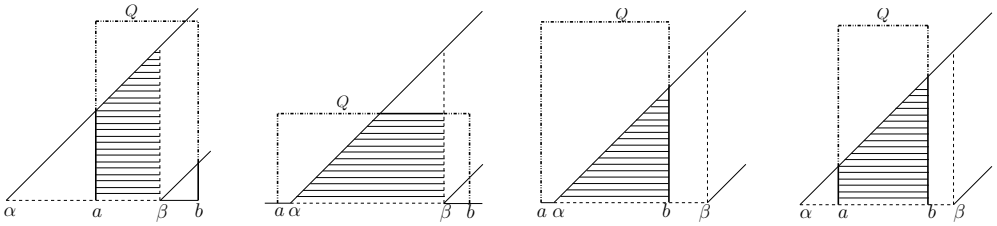

FIGURE 6.3. $Q_{\mathcal{P}}(a, b, c)$ and tile t intersections.

6.5 Towards $O(\log N + K)$ Time Solution

Our result in lemma 25 is optimal for $K \geq \log^2 N$. In this section, we take a step forward to achieve more efficient time solution for smaller values of K using multi-slab ideas. Using a parameter Δ (to be fixed later), we partition the $[1, N] \times [1, N]$ grid into $L = \lceil N/\Delta \rceil$ vertical slabs (Figure 6.4(a)). Multi-slabs $VS_0, VS_1,$ ..., VS_L are the slabs induced by lines $x = i\Delta$ for $i = 0, 1, ..., L$. Denote by δ_i ($i \in [0, L]$) the minimum x-coordinate in VS_i. For notational convenience, we define $\delta_{L+1} = \infty$. By slight abuse of notation, we use VS_i to represent the set of points in the corresponding slab.

A query $Q_{\mathcal{P}}$ with (a, b, c) as an input is called *inter-slab* query if it overlaps two or more vertical slabs, otherwise if it is entirely contained within a single vertical slab we call it an *intra-slab* query. In this section, we propose a data structure that can answer inter-slab queries optimally.

Lemma 28. Inter-slab SCRR queries can be answered in $O(\log N + K)$ time in RAM model using a data structure occupying $O(N)$ words space, where K represents the number of output.

Proof. Given a query $Q_{\mathcal{P}}(a, b, c)$ such that $a \leq b$, let α, β be integers that satisfy $\delta_\alpha \leq a < \delta_{\alpha+1}$ and $\delta_\beta \leq b < \delta_{\beta+1}$. The x interval of an inter-slab query i.e. $[a, b]$ spreads across at least two vertical slabs. Therefore, $Q_{\mathcal{P}}$ can be decomposed into five subqueries $Q^1_{\mathcal{P}}$, $Q^2_{\mathcal{P}}$, $Q^3_{\mathcal{P}}$, $Q^4_{\mathcal{P}}$ and $Q^5_{\mathcal{P}}$ as illustrated in Figure 6.4(b). These subqueries are defined as follows.

- $Q^1_{\mathcal{P}}$ is the part of $Q_{\mathcal{P}}$ which is in $[\delta_\alpha, \delta_{\alpha+1}) \times [1, \delta_\alpha) \times [c, N]$.
- $Q^2_{\mathcal{P}}$ is the part of $Q_{\mathcal{P}}$ which is in $[\delta_\beta, \delta_{\beta+1}) \times [1, \delta_{\alpha+1}) \times [c, N]$.
- $Q^3_{\mathcal{P}}$ is the part of $Q_{\mathcal{P}}$ which is in $[\delta_{\alpha+1}, \delta_\beta) \times [\delta_\alpha, \delta_{\alpha+1}) \times [c, N]$.
- $Q^4_{\mathcal{P}}$ is the part of $Q_{\mathcal{P}}$ which is in $[\delta_{\alpha+1}, \delta_\beta) \times [1, \delta_\alpha) \times [c, N]$.
- $Q^5_{\mathcal{P}}$ is the part of $Q_{\mathcal{P}}$ which is in $[\delta_\alpha, \delta_{\alpha+1}) \times (\delta_\alpha, \delta_{\alpha+1}) \times [c, N]$.

If $\alpha + 1 = \beta$ then we only need to consider subqueries $Q^1_{\mathcal{P}}$, $Q^2_{\mathcal{P}}$ and $Q^5_{\mathcal{P}}$. Each of these subqueries can now be answered as follows.

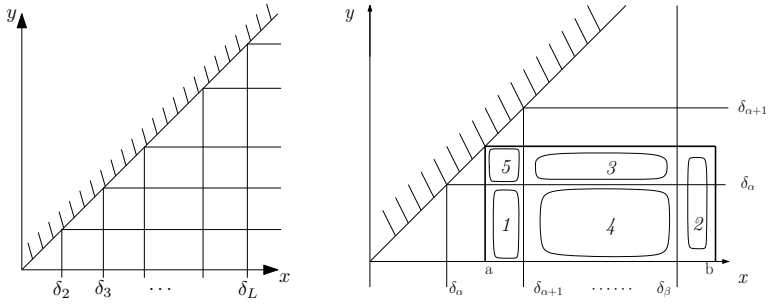

FIGURE 6.4. Divide-and-conquer scheme using Δ

Answering $Q^1_{\mathcal{P}}$***:*** The subquery $Q^1_{\mathcal{P}}$ can be answered by retrieving all points in $VS_\alpha \cap [a, N] \times [1, N]$ with weight $\geq c$. This is a two-dimensional dominance query in VS_α. This can be achieved by maintaining a three-dimensional dominance query structure for RAM model [1] for the points in VS_i for $i = 1, \ldots, L$ separately. The query time will be $O(\log |VS_\alpha| + K_1) = O(\log N + K_1)$, where K_1 is the output size and index space is $O(\sum_{i=1}^{L} |VS_i|) = O(N)$ words.

Answering $Q^2_{\mathcal{P}}$***:*** To answer subquery $Q^2_{\mathcal{P}}$ we will retrieve all points in $VS_\beta \cap [1, b) \times [1, a)$ with weight $\geq c$. By maintaining a collection of three-dimensional dominance query structures [1] occupying linear space overall, $Q^2_{\mathcal{P}}$ can be answered in $O(\log N + K_2)$ time, where K_2 is the output size.

Answering $Q^3_{\mathcal{P}}$***:*** To answer subquery $Q^3_{\mathcal{P}}$, we begin by partitioning the set of points $\mathcal{P}$ into L horizontal slabs $HS_1, HS_2, \ldots, HS_L$ induced by lines $y = i\Delta$, such that $HS_i = \mathcal{P} \cap [\delta_{i+1}, N] \times [\delta_i, \delta_{i+1})$. The subquery $Q^3_{\mathcal{P}}$

can now be answered by retrieving all points in $HS_\alpha \cap [1, \delta_\beta) \times [1, a)$ with weight $\geq c$. This can be achieved by maintaining a three-dimensional dominance query structure [1] for the points in HS_i for $i = 1, ..., L$ separately. Since each point in S belongs to at most one HS_i the overall space can be bounded by $O(N)$ words and the query time can be bounded by $O(\log |HS_\alpha|) + K_3) = O(\log N + K_3)$ time, with K_3 being the number of output.

Answering $Q^5_{\mathcal{P}}$***:*** To answer subquery $Q^5_{\mathcal{P}}$ we will retrieve all points in $VS_\alpha \cap (a, N] \times [1, a)$ with weight $\geq c$. By maintaining a collection of three-dimensional dominance query structures occupying linear space overall as described in earlier subsections, $Q^5_{\mathcal{P}}$ can be answered in $O(\log |VS_\alpha| + K_5) = O(\log N + K_5)$ time, where K_5 is the output size.

Answering $Q^4_{\mathcal{P}}$: We begin by describing a naive way of answering $Q^4_{\mathcal{P}}$ by using a collection of three-dimensional dominance query structures built for answering $Q^1_{\mathcal{P}}$. We query VS_i to retrieve all the points in $VS_i \cap [1, N] \times [1, \delta_\alpha)$ with weight $\geq c$ for $i = \alpha + 1, ..., \beta - 1$. Such a query execution requires $O((\beta - \alpha + 1) \log N + K_4)$ time, where K_4 is the output size. We are required to spend $O(\log N)$ time for each vertical slab even if the query on a particular VS_i does not produce any output. To answer subquery $Q^4_{\mathcal{P}}$ in $O(\log N + K_4)$ time, we make following crucial observations: (1) All three boundaries of $Q^4_{\mathcal{P}}$ are on the partition lines, (2) The left boundary of $Q^4_{\mathcal{P}}$ (i.e., line $x = \delta_{\alpha+1}$) is always the successor of the top boundary (i.e., line $y = \delta_\alpha$), (3) The output size is bounded by $O(\log^2 N)$.

We use these observations to construct following data structure: Since the top left and bottom right corner of $Q^4_{\mathcal{P}}$ falls on the partition lines, there are at most $(N/\Delta)^2$ possible different rectangles for $Q^4_{\mathcal{P}}$. For each of these we store at most top-$O(\log^2 N)$ points in sorted order of their weight. Space requirement of this data structure is $O((N/\Delta)^2 \log^2 N)$ words. Query algorithm first identifies the rectangle that matches with $Q^4_{\mathcal{P}}$ among $(N/\Delta)^2$ rectangles and then simply reports the points with weight greater than c in optimal time. Finally to achieve linear space, we choose $\Delta = \sqrt{N} \log N$.

Thus, we can obtain $K = K_1 + K_2 + K_3 + K_4 + K_5$ output in $O(K)$ time. Also, in the divide and conquer scheme, the point sets used for answering subqueries $Q^1_{\mathcal{P}}, ..., Q^5_{\mathcal{P}}$ are disjoint, hence all reported answers are unique. Now given a query $Q_{\mathcal{P}}(a, b, c)$, if the subquery $Q^4_{\mathcal{P}}$ in the structure just described returns $K_4 = \log^2 N$ output, it suggest that output size $K > \log^2 N$. Therefore, we can query the structure in lemma 25 and still retrieve all output in optimal time. This completes the proof of lemma 28.

□

6.6 Linear Space and $O(\log N + K)$ Time Data Structure in RAM Model

In this section, we show how to obtain our $O(\log N + K)$ time result stated in Theorem 7 via bootstrapping our previous data structure.

We construct $\psi_1, \psi_2, ..., \psi_{\lceil \log \log N \rceil}$ levels of data structures, where $\psi_1 = \sqrt{N} \log N$ and $\psi_i = \sqrt{\psi_{i-1}} \log \psi_{i-1}$, for $i = 2, 3, ..., \lceil \log \log N \rceil$. At each level, we use multi-slabs to partition the points. More formally, at each level ψ_i, the $[1, N] \times [1, N]$ grid is partitioned into $g = \lceil N/\psi_i \rceil$ vertical slabs or multi-slabs. At level ψ_i, multi-slabs $BS_0, BS_1, ..., BS_g$ are the slabs induced by lines $x = j\psi_i$ for $j = 0, 1, ..., g$. Each multi-slab BS_j is further partitioned into disjoint upper partition US_j and lower partition LS_j (Figure 6.5). Below we describe the data structure and query answering in details.

For a multi-slab BS_j and a $SCRR$ query, US_j can have more constraints than LS_j, making it more difficult to answer query on US_j. Our idea is to exclude the points of US_j from each level, and build subsequent levels based on only these points. Query answering for LS_j can be done using the inter-slab query data structure and three-sided range reporting data structure.

At each level ψ_i, we store the data structure described in Lemma 28 capable of answering inter-slab queries by taking slab width $\Delta = \sqrt{\psi_i} \log \psi_i$. Also we store separate three-sided range reporting data structures for each LS_j, $j = 0, 1, ..., g$. The points in US_j at level ψ_i (for $i = 1, \cdots, \lceil \log \log N \rceil - 1$) are removed from consideration. These removed points are considered in subsequent levels. Note that the inter-slab query data structure stored here is slightly different from the data structure described in lemma 28, since we removed the points of US_j (region 5 of figure 6.4b). $\psi_{\lceil \log \log N \rceil}$ is the bottom level and no point is removed from here. Level ψ_{i+1} contains only the points of all the US_j partitions from previous level ψ_i, and again the upper partition points are removed at level ψ_{i+1}. More specifically, level ψ_1 contains an inter-slab query data structure and $\sqrt{N} \log N$ number of separate two-dimensional three-sided query data structures over each of the lower partitions LS_j. Level ψ_2 contains $\sqrt{N} \log N$ number of data structures similar to level ψ_1 corresponding to each of the upper partitions US_j of level ψ_1. Subsequent levels are constructed in a similar way. No point is repeated at any level, two-dimensional three-sided query data structures and inter-slab query data structures take linear space, giving linear space bound for the entire data structure.

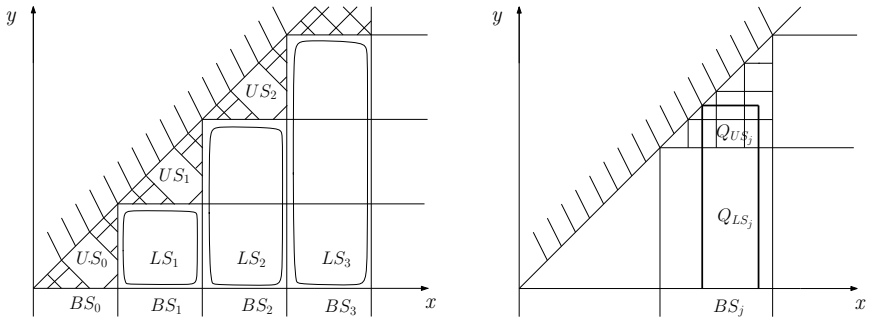

FIGURE 6.5. Optimal time SCRR query data structure.

A SCRR query $Q_{\mathcal{P}}$ can be either an inter-slab or an intra-slab query at level ψ_i (illustrated in figure 6.6). An intra-slab SCRR query $Q_{\mathcal{P}}$ can be divided into Q_{US_i} and Q_{LS_i}. Q_{LS_i} is three-sided query in LS_i, which can be answered by the three-sided range reporting structure stored at LS_i. Q_{US_i} is issued as a SCRR query for level ψ_{i+1} and can be answered by traversing at most $\lceil \log \log N \rceil$ levels. An inter-slab SCRR query $Q_{\mathcal{P}}$ can be decomposed into 5 sub-queries: Q_1, Q_2, Q_3, Q_4 and Q_5. Q_1, Q_2, Q_3 and Q_4 can be answered using the optimal inter-slab query data structure of lemma 28 in similar way described in details in section 6.5. Again Q_5 is issued as a SCRR query for level ψ_{i+1} and can be answered in subsequent levels. At each level $O(\log \psi_i + K_i)$ time is needed where K_i is the output size obtained at level ψ_i. Since no point is repeated at any level, all reported answers are unique. At most $\log \log N$ levels need to be accessed for answering $Q_{\mathcal{P}}$. Total time is bounded by $O(\sum_{i=1}^{\log \log N}(\log \psi_i) + \log \log N + \sum_{i=1}^{\log \log N}(K_i))$ = $O(\log N + K)$, thus proving theorem 7.

6.7 SCRR Query in External Memory

In this section we discuss two external memory data structures for SCRR query, one achieving optimal I/O and another achieving linear space. Both these data structures are obtained by modifying our RAM model data structure. We use the multi-slab ideas similar to section 6.5. We assume points in $\mathcal{P}$ to be in rank-space. We begin by stating external memory variants of lemma 25 and lemma 28.

Lemma 29. By maintaining an $O(N)$-word structure, any SCRR query $Q_{\mathcal{P}}(\cdot, \cdot, \cdot)$ can be answered in $O(\log^2(N/B) + K/B)$ I/Os, where K is the output size.

Proof. We use $\lceil \log(N/B) \rceil$ number of oblique slabs induced by lines $x = y + 2^i B$ for $i = 0, 1, ... \lceil \log(N/B) \rceil$. Each oblique slab is partitioned into tiles using axis parallel lines $x = (2^{(i-1)} * (1 + j))B$ and $y = 2^{(i-1)} * jB$ for $j = 1, 2, ...$ It can be easily shown that any SCRR query $Q_{\mathcal{P}}(a, b, d)$ intersects with at most

$O(\log(N/B))$ tiles, each of which can be resolved in linear space and optimal I/Os using three-dimensional query structure [1] in each tile, achieving $O(\log^2(N/B) + K/B)$ total I/Os. □

Lemma 30. Inter-slab SCRR queries can be answered in $O(\log_B N + K/B)$ I/Os using a data structure occupying $O(N)$ words space, where K represents the number of outputs.

Proof. This can be achieved by using a data structure similar to the one described in lemma 28 with $\Delta = \sqrt{NB}\log_B N$. We use external memory counterparts for three sided and three dimensional dominance reporting and for answering query Q_p^4 we maintain top $O(log_B^2 N)$ points from each of the rectangle. □

6.7.1 Linear Space Data Structure

The linear space data structure is similar to the RAM model linear space and optimal time structure described in section 6.6. Major difference is we use $\psi_i = \sqrt{\psi_{i-1}B}\log_B \psi_{i-1}$ for bootstrapping and use the external memory counterparts of the data structures.

We construct $\psi_1, \psi_2, ..., \psi_{\lceil \log\log N\rceil}$ levels of data structures, where $\psi_1 = \sqrt{NB}\log_B N$ and any $\psi_i = \sqrt{\psi_{i-1}B}\log_B \psi_{i-1}$. At each level ψ_i, the $[1, N] \times [1, N]$ grid is partitioned into $g = \lceil N/\psi_i \rceil$ vertical slabs. Multi-slabs BS_j, upper partition US_j and lower partition LS_j for $j = 0, 1, ..., g$ are defined similar to section 6.6.

At each level ψ_i, we store the data structure described in Lemma 30 capable of answering inter-slab queries in optimal I/Os by taking slab width $\Delta = \sqrt{\psi_i B}\log_B \psi_i$. Also we store separate three-sided external memory range reporting data structures for each LS_j, $j = 0, 1, ..., g$. In order to maintain linear space, the points in US_j at level ψ_i (for $i = 1, \cdots, \lceil \log\log N\rceil - 1$) are removed from consideration. These removed points are considered in subsequent levels. $\psi_{\lceil \log\log N\rceil}$ is the bottom level and no point is removed from here. Level ψ_{i+1} contains only the points of all the US_j partitions from previous level ψ_i, and again the upper partition points are removed at level ψ_{i+1}. External memory two-dimensional three-sided query data structures and optimal inter-slab query data structures for external memory take linear space, and we avoided repetition of points in different levels, thus the overall data structure takes linear space.

Query answering is similar to section 6.6. If the SCRR query $Q_{\mathcal{P}}$ is intra-slab, then $Q_{\mathcal{P}}$ can be divided into Q_{US_i} and Q_{LS_i}. Q_{LS_i} is three-sided query in LS_i, which can be answered by the three-sided range reporting structure stored at LS_i. Q_{US_i} is issued as a SCRR query for level ψ_{i+1} and can be answered by traversing at most $\lceil \log\log N\rceil$ levels. An inter-slab SCRR query $Q_{\mathcal{P}}$ can be decomposed into 5 sub-

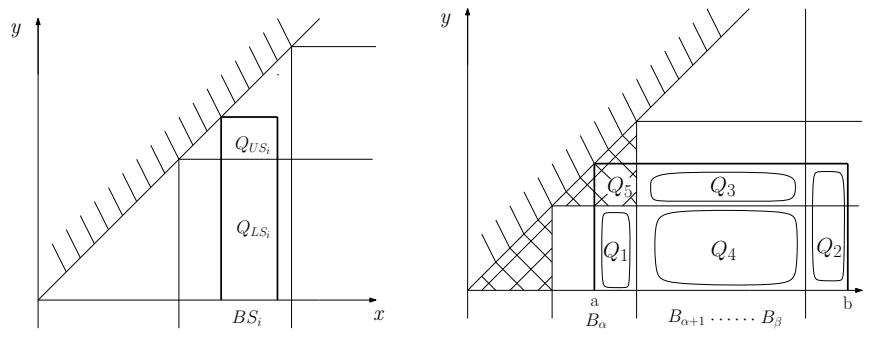

FIGURE 6.6. Intra-slab and Inter-slab query for linear space data structure.

queries: Q_1, Q_2, Q_3, Q_4 and Q_5. Q_1, Q_2, Q_3 and Q_4 can be answered using the optimal inter-slab query data structure of lemma 30 in similar way described in details in section 6.5. Again Q_5 is issued as a SCRR query for level ψ_{i+1} and can be answered in subsequent levels. At each level $O(\log_B \psi_i + K_i/B)$ I/Os are needed where K_i is the output size obtained at level ψ_i. Since no point is repeated at any level, all reported answers are unique. At most $\log\log N$ levels need to be accessed for answering $Q_{\mathcal{P}}$. Total I/O is bounded by $O(\sum_{i=1}^{\log\log N}(\log_B \psi_i) + \log\log N + \sum_{i=1}^{\log\log N}(K_i/B))$ = $O(\log_B N + \log\log N + K/B)$ thus proving theorem 8.

6.7.2 I/O Optimal Data Structure

I/O optimal data structure is quite similar to the linear space data structure. The major difference is the points in US_j at level ψ_i (for $i = 1, \cdots, \lceil\log\log N\rceil - 1$) are not removed from consideration. Instead at each US_j we store external memory data structure capable of answering inter-slab query optimally(Lemma 30). All the points of US_j ($j = 0, 1, ..., g$) of level ψ_i (for $i = 1, \cdots, \lceil\log\log N\rceil - 1$) are repeated in the next level ψ_{i+1}. This will ensure that we will have to use only one level to answer the query. For LS_j ($j = 0, 1, ..., g$), we store three-sided query structures. Since there are $\log\log N$ levels, and at each level data structure uses $O(N)$ space, total space is bounded by $O(N\log\log N)$. To answer a query $Q_{\mathcal{P}}$, we query the structures associated with ψ_i such that $Q_{\mathcal{P}}$ is an intra-slab query for ψ_i and is inter-slab for ψ_{i+1}. We can decompose the query $Q_{\mathcal{P}}$ into Q_{US} and Q_{LS}, where Q_{US}(Q_{LS}) falls completely within US_j(LS_j). Since, Q_{US} is an inter-slab query for the inter-slab query data structure stored at US_j, it can be answered optimally. Also Q_{LS} is a simple three-sided query for which output can be retrieved in optimal time using the three-sided structure of LS_j. This completes the proof for Theorem 9.

Chapter 7
Discriminating and Generic Words

7.1 Problem Formulation and Motivation

Let $\mathcal{D}$=$\{d_1, d_2, d_3, ..., d_D\}$ be a collection of D strings (which we call as documents) of total n characters from an alphabet set Σ of size σ. For simplicity we assume, every document ends with a special character \$ which does not appear any where else in the documents. Our task is to index $\mathcal{D}$ in order to compute all (i) *maximal generic words* and (ii) *minimal discriminating words* corresponding to a query pattern P (of length p). The document frequency $df(.)$ of a pattern P is defined as the number of distinct documents in $\mathcal{D}$ containing P. Then, a generic word is an extension $\bar{P}$ of P with $df(\bar{P}) \geq \tau$, and is maximal if $df(P') < \tau$ for all extensions P' of $\bar{P}$. Similarly, a discriminating word is an extension $\bar{P}$ of P with $df(\bar{P}) \leq \tau$, and is called a minimal discriminating word if $df(P') > \tau$ for any proper prefix P' of $\bar{P}$ (i.e., $P' \neq \bar{P}$). These problems were introduced by Kucherov et al. [77], and they proposed indexes of size $O(n \log n)$ bits or $O(n)$ words. The query processing time is optimal $O(p + output)$ for reporting all maximal generic words, and is near optimal $O(p + \log\log n + output)$ for reporting all minimal discriminating words. We describe succinct indexes of $n \log \sigma + o(n \log \sigma) + O(n)$ bits space with $O(p + \log\log n + output)$ query times for both these problems.

These problems are motivated from applications in computational biology. For example, it is an interesting problem to identify words that are exclusive to the genomic sequences of one species or family of species [42]. Such patterns that appear in a small set of biologically related DNA sequences but do not appear in other sequences in the collection, often carries a biological significance. Identification of genomic markers, or probe design for DNA microarrays are also closely related problems. Discriminating and generic words also find applications in text mining and automated text classification.

7.2 Data Structure and Tools

In this section, we describe some data structure used for our index.

7.2.1 Encoding of generalized suffix tree

Encoding of GST with the goal of supporting navigation and other tree operations has been extensively studied in the literature. We use the data structure by Sadakane and Navarro [111] with focus on following operations:

- $lca(i, j)$: the lowest common ancestor of two nodes i, j
- $child(i, k)$: k-th child of node i
- $level\text{-}ancestor(i, d)$: ancestor j of i such that depth(j)=depth(i)-d
- $subtree\text{-}size(i)$: number of nodes in the subtree of node i

Lemma 31. [111] An ordinal tree with n nodes can be encoded by $2n + O(\frac{n}{polylog(n)})$ bits supporting lca, k-th child, level-ancestor, and subtree-size queries in constant time.

Define $count(i) = df(path(i))$. Using the data structure by Sadakane [109] we can answer $count(i)$ query efficiently for any input node i. Following lemma summarizes the result in [109].

Lemma 32. [109] Generalized suffix tree (GST) with n leaves can be encoded by $2n + o(n)$ bits, supporting $count(i)$ query in constant time.

7.2.2 Segment Intersection Problem

In [77] authors showed how the problem of identifying minimal discriminating words can be reduced to orthogonal segment intersection problem. In this article, we rely on this key insight for both the problems under consideration and use the result summarized in lemma below for segment intersection.

Lemma 33. [24] A given set $\mathcal{I}$ of n vertical segments of the form $(x_i, [y_i, y'_i])$, where $x_i, y_i, y'_i \in [1, n]$ can be indexed in $O(n)$-word space (in Word RAM model), such that whenever a horizontal segment $s_q = ([x_q, x'_q], y_q)$ comes as a query, all those vertical segments in $\mathcal{I}$ that intersect with s_q can be reported in $O(\log \log n + output)$ time.

7.3 Computing Maximal Generic Words

We first review a linear space index, which is based on the ideas from the previous results [77]. Later we show how to employ sampling techniques to achieve a space efficient solution.

7.3.1 Linear Space Index

Let i_P be the locus node of the query pattern P. Then, our task is to return all those nodes j in the subtree of i_P such that $count(j) \geq \tau$ and $count(.)$ of every child node of j is less than τ. Note that corresponding to each such output j, $path(j)$ represents a maximal generic word with respect to the query (P, τ). The mentioned task can be performed efficiently by reducing the original problem to a segment intersection problem as follows. Each node i in GST is mapped to a vertical segment $(i, [count(i_{max}) + 1, count(i)])$, where i_{max} is the child of i with the highest $count(.)$ value. If i is a leaf node, then we set $count(i_{max}) = 0$. Moreover, if $count(i) = count(i_{max})$ we do not maintain such a segment as it can not possibly lead to a generic word. The set $\mathcal{I}$ of these segments is then indexed using a linear space structure as described in Section 7.2.2. Additionally we maintain the GST of $\mathcal{D}$ as well.

The maximal generic words corresponding to a query (P, τ) can be computed by issuing an orthogonal segment intersection query on $\mathcal{I}$ with $s_q = ([i_P, i'_P], \tau)$ as the input, where i_P is the locus node of pattern P and i'_P represents the right most leaf in the subtree of i_P. It can be easily verified than $path(j)$ corresponding to each retrieved interval $(j, [count(j_{max}) + 1, count(j)])$ is a maximal generic word. In conclusion, we have a linear space index of $O(n)$ words with $O(\log \log n + output)$ query time. By combining with the space for GST, and the initial $O(p)$ time for pattern search, we have the following lemma.

Lemma 34. There exists an $O(n)$ word data structure for reporting maximal generic word queries in $O(p + \log \log n + output)$ time. □

7.3.2 Succinct Space Index

In this section, we begin by extending the marking scheme of Hon et al. [66] described earlier in Section 2.6 and then discuss our succinct space index. We introduce the notion of *orphan* and *pivot* nodes in GST based on the marking scheme of Hon et al. as follows:

1. *Orphan node* is a node with no marked node in its subtree. Note that the number of leaves in the subtree of an orphan node is at most g.

2. *Pivot* is an orphan node with *non-orphan* parent. Therefore, every orphan node has a unique pivot ancestor. The number of leaves in the subtree of any pivot node is at most g and the number of pivot nodes can be $\omega(n/g)$.

We now describe a structure to solve a variant of computing maximal generic words summarized in lemma below that forms the basis of our final space-efficient index. We choose $g = \lfloor \frac{1}{8} \log n \rfloor$ as a grouping parameter in the marking scheme of Hon et al. and nodes in GST are identified by their pre-order rank.

Lemma 35. The collection $\mathcal{D}=\{d_1, d_2, d_3, ..., d_D\}$ of strings of total n characters can be indexed in $(n \log \sigma + O(n))$ bits, such that given a query (P, τ), we can identify all marked nodes j^* satisfying either of the following condition in $O(p + \log \log n + output)$ time.

- $path(j^*)$ being a maximal generic word for input (P, τ)
- there is at least one node in $N(j^*)$ that corresponds to desired maximal generic word, where $N(j^*)$ is a set of all those nodes in GST with j' (unique prime ancestor of j^*) as the first prime node on the path from that node to the root
- j^* is a highest marked descendant of i_P

Proof. It can be noted that, $N(j^*)$ is essentially a set of all nodes in the subtree of j' but not in the subtree of j^* ($N(j^*)$ does not include node j^*). To be able to retrieve the required marked nodes in succinct space, we maintain index consisting of following components:

- Instead of GST, we use its space efficient version i.e., a compressed suffix array (CSA) of T. There are many versions of CSA's available in literature, however for our purpose we use the one in [10] that occupies $n \log \sigma + o(n \log \sigma) + O(n)$ bits space and retrieves the suffix range of pattern P in $O(p)$ time.
- A $4n + o(n)$ bits encoding of GST structure (Lemma 31) to support the (required) tree navigational operations in constant time.
- We keep a bit vector $B_{mark}[1...2n]$, where $B_{mark}[i] = 1$ iff node i is a marked node, with constant time rank-select supporting structures over it in $O(n)$ bits space [106]. This bit vector enables us to retrieve the unique highest marked descendant of any given node in GST in constant time.
- We map each marked node i^* to a vertical segment $(i^*, [count(i^*_{max}) + 1, count(i^*)])$, where i^*_{max} is the child of i^* with the highest $count(.)$ value. The number of such segments can be bounded by $O(n/g)$.

The set $\mathcal{I}_\infty$ of these segments is then indexed using a linear space structure as before in $O(n/g) = O(n/\log n)$ words or $O(n)$ bits space. It is to be noted that segments with $count(i^*_{max})+1 = count(i^*)$ are not included in set $\mathcal{I}_\infty$ as those segments do not correspond to generic words.

- We also maintain $O(n)$ bit structure for set $\mathcal{I}_\in$ of segments of the form $(i', [count(i^*) + 1, count(i')])$, where i^* is a marked node and i' is the unique lowest prime ancestor of i^*. Again the segments with $count(i^*) + 1 = count(i')$ are not maintained.

Given an input (P, τ), we first retrieve the suffix range in $O(p)$ time using CSA and the locus node i_P in another $O(1)$ time using GST structure encoding. Then we issue an orthogonal segment intersection query on $\mathcal{I}_\infty$ and $\mathcal{I}_\in$ with $s_q = ([i_P, i'_P], \tau)$ as the input, i'_P being the right most leaf in the subtree of i_P. Any marked node that corresponds to a maximal generic word for query (P, τ) is thus retrieved by querying set $\mathcal{I}_\infty$. Instead of retrieving non-marked nodes corresponding to the maximal generic word, the simplified structure returns their representative marked nodes instead.

For a segment in $\mathcal{I}_\in$ that is reported as an answer, corresponding to a marked node j^* with j' being its lowest prime ancestor, we have $count(j') \geq \tau$ and $count(j^*) < \tau$. Therefore, there must exist a node pair (u, v) sharing parent-child relationship on the path from j' to j^* such that $count(u) \geq \tau$ and $count(v) < \tau$. As before, let u_{max} be the child of node u with the highest $count(.)$ value. It can be easily seen that if $(v = u_{max})$ or $(v \neq u_{max}$ with $count(u_{max}) < \tau)$, then $path(u)$ is a maximal generic word with respect to (P, τ). Otherwise, consider the subtree rooted at node u_{max}. For this subtree $count(u_{max}) \geq \tau$ and $count(.) = 1$ for all the leaves and hence it is guaranteed to contain at least one maximal generic word for query (P, τ).

We highlight that the segment intersection query on set $\mathcal{I}_\in$ will be able to capture the node pairs (j^*, j') where both j^* and j' are in the subtree of locus node i_P as the segments in $\mathcal{I}_\in$ use (pre-order rank of) prime node as their x coordinate. The case when both j^*, j' are outside the subtree of i_P can be ignored as in this case none of the nodes in $N(j^*)$ will be in the subtree of i_P and hence can not lead to a desired output. Further we observe that for the remaining scenario when locus node i_P is on the path from j' to j^* (both exclusive), there can be at-most one such pair (j^*, j'). This is true due to the way nodes are marked in GST and moreover j^* will be the highest marked descendant of i_P (if exists). Such j^* can be obtained in constant time by first obtaining the marked node using query $select_1(rank_1(i_P) + 1)$ on B_{mark} and then evaluating if

it is in the subtree of i_P. We note that in this case $N(j^*)$ (j^* being the highest marked descendant of i_P) may or may not result in a maximal generic word for query (P, τ), however we can afford to verify it irrespective of the output due to its uniqueness. □

□

If a marked node j^* reported by the data structure just described is retrieved from set $\mathcal{I}_\infty$ then $path(j^*)$ can be returned as an maximal generic word for query (P, τ) directly. However, every other marked node retrieved needs to be decoded to obtain the actual maximal generic word corresponding to one or more non-marked nodes it represents. Before we describe the additional data structures that enable such decoding, we classify the non-marked answers as orphan and non-orphan outputs based on whether or not it has any marked node in its subtree as defined earlier. Let j^* be a marked node and j' be its lowest prime ancestor. A non-marked node that is an output is termed as orphan if it is not on the path from j' to j^*. Due to Lemma 35, every marked node retrieved from $\mathcal{I}_\in$ leads to either a orphan output or non-orphan outputs or both. Below, we describe how to report all orphan and non-orphan outputs for a given query (P, τ) and a marked node j^*. We first append the index in Lemma 35 by data structure by Sadakane (Lemma 32) without affecting its space complexity to answer $count(.)$ query in constant time for any GST node.

Retrieving non-orphan outputs: To be able to report a non-orphan output of query (P, τ) for a given marked node (it it exists), we maintain a collection of bit vectors as follows:

- We associate a bit vector to each marked node i^* that encodes $count(.)$ information of the nodes on the (top-down) path from i' to i^*, i' being the (unique) lowest prime ancestor of i^*. Let $x_1, x_2, ..., x_r$ be the nodes on this path inclusive of both i' and i^*. Note that $r \leq g$, $\delta_i = count(x_i - 1) - count(x_i) \geq 0$, and $x_r - x_i \leq 2g$ base on the properties of the marking scheme. Now we maintain a bit vector $B_{i^*} = 10^{\delta_1}10^{\delta_2}...10^{\delta_r}$ along with constant time rank-select supporting structures at marked node i^*. As length of the bit vector is bounded by $O(2g)$ and number of marked nodes is bounded by $O(n/g)$, total space required for these structures is $O(n)$ bits.

- Given a node i we would like to retrieve the node i_{max} i.e., child of node i with the highest $count(.)$ value in constant time. To enable such a lookup we maintain a bit vector $B = 10^{\delta_1}10^{\delta_2}...10^{\delta_n}$, where $child(i, \delta_i) = i_{max}$. If i is leaf then we assume $\delta_i = 0$. As each node contributes exactly one bit

with value 1 and at most one bit with value 0, length of the bit vector B is also bounded by $2g$ and subsequently occupies $O(n)$ bits.

Given a marked node j^* and a query (P, τ), we need to retrieve a parent-child node pair (u, v) on the path from j' to j^* such that $count(u) \geq \tau$ and $count(v) < \tau$. We can obtain the lowest prime ancestor j' of j^* in constant time to begin with [66]. Then to obtain a node u, we probe bit vector B_{j^*} by issuing a query $rank_1(select_0(count(j') - \tau))$. We note that these rank-select operations only returns the distance of the node u from j^* which can be then used along with level-ancestor query on j^* to obtain u. To verify if $path(u)$ indeed corresponds to a maximal generic word, we need to check if $count(.) \leq \tau$ for all the child nodes of u. To achieve this, we retrieve the $j_{max} = child(u, select_1(u+1) - select_1(u) - 1)$ and obtain its count value $count(j_{max})$ using a data structure by Sadakane (Lemma 32) in constant time. Finally, if $count(j_{max}) \leq \tau$ then u can be reported as an maximal generic word for (P, τ) query. If the input node j^* is highest marked descendant of locus node i_P then we need to verify if node u is within the subtree of i_P before it can be reported as an maximal generic word. Thus overall time spent per marked node to output the associated maximal generic word (if any) is $O(1)$. We note that unsuccessfully querying the marked node for non-orphan output does not hurt the overall objective of optimal query time since, such a marked node is guaranteed to generate orphan outputs (possibly except the highest marked ancestor of locus node i_P).

Retrieving orphan outputs: Instead of retrieving orphan outputs of the query based on the marking nodes as we did for non-orphan outputs, we retrieve them based on pivot nodes by following two step query algorithm: (i) identify all pivot nodes i in the subtree of the locus node i_P of P, with $count(i) \geq \tau$ and (ii) explore the subtree of each such i to find out the actual (orphan) outputs. If $count(i) \geq \tau$, then there exists at least on output in the subtree of i, otherwise there will not be any output in the subtree of i.

Since an exhaustive search in the subtree of a pivot node is prohibitively expensive, we rely on the following insight to achieve the optimal query time. For a node i in the GST, let $subtree\text{-}size(i)$, $leaves\text{-}size(i)$ represents the number of nodes and number of leaves in the subtree rooted at node i respectively. The subtree of i can be then encoded (simple balanced parenthesis encoding) in $2subtree\text{-}size(i)$ bits. Also the $count(.)$ values of all nodes in the subtree of i in GST can be encoded in $2leaves\text{-}size(i)$ bits using the encoding scheme by Sadakane [109]. Therefore $2subtree\text{-}size(i) + 2leaves\text{-}size(i)$ bits are sufficient to encode the subtree of any node i along with the $count(.)$ information. Since $subtree\text{-}size(i) < 2leaves\text{-}size(i)$ and

there are at most g leaves in the subtree of a pivot node, for a pivot node i we have $2subtree\text{-}size(i)+2leaves\text{-}size(i) < 6g = \frac{3}{4}\log n$. This implies the number of distinct subtrees rooted at pivot nodes possible with respect to the above encoding is bounded by $\sum_{k=1}^{\frac{3}{4}\log n} 2^k = \Theta(n^{3/4})$.

To be able to efficiently execute the two step algorithm to retrieve all orphan outputs we maintain following components:

- A bit vector $B_{piv}[1...2n]$, where $B_{piv}[i] = 0$ iff node i is a pivot node, along with constant time rank-select supporting structures over it occupying $O(n)$ bits space.

- Define an array $E[1...]$, where $E[i] = count(select_1(i))$ with $select$ operation applied to a bit vector B_{piv} (i.e., the count of i-th pivot node). Array E is not maintained explicitly, instead a $2n + o(n)$ bits RMQ structure over it is maintained.

- For each distinct encoding of a pivot node out of total $\Theta(n^{3/4})$ of them, we shall maintain the list of top-g answers for $\tau = 1, 2, 3, ..., g$ in overall $O(n^{3/4}g^2) = o(n)$ bits space.

- The total $\Theta(n^{3/4})$ distinct encodings of pivot nodes can be thought to be categorized into groups of size 2^k for $k = 1, ..., \frac{3}{4}\log n$. Encodings for all possible distinct pivot nodes i having $k = 2subtree\text{-}size(i) + 2leaves\text{-}size(i)$ are grouped together and let L_k be this set. Then for a given pivot node i in GST with $k = 2subtree\text{-}size(i) + count(i)$, we maintain a pointer so as to enable the lookup of top-g answers corresponding to the encoding of subtree of i among all the encoding in set L_k. With number of bits required to represent such a pointer being proportional to the number leaves in the subtree of a pivot node i i.e. $2subtree\text{-}size(i) + count(i)$, overall space can be bounded by $O(n)$ bits.

Query processing can now be handled as follows. Begin by identifying the x-th and y-th pivot nodes, which are the first and last pivot nodes in the subtree of the locus node i_P in $O(1)$ time as $x = 1 + rank_1(i_P - 1)$ and $y = rank_1(i'_P)$ using bit vector B_{piv}, where i'_P is the rightmost leaf of the subtree rooted at i_P. Then, all those $z \in [x, y]$ where $E[z] \geq \tau$ can be obtained in constant time per z using recursive range maximum queries on E as follows: obtain $z = RMQ_E(x, y)$, and if $E[z] < \tau$, then stop recursion, else recurse the queries on intervals $[x, z - 1]$ and $[z + 1, y]$. Recall that even if $E[z]$ is not maintained explicitly, it can be obtained in constant time using B_{piv} as $E[z] = count(select_1(z))$. Further, the pivot node corresponding to each z can be obtained in constant time as $select_1(z)$. In conclusion, step (i) of query algorithm can be

performed in optimal time. Finally, for each of these pivot nodes we can find the list of pre-computed answers based on given τ and report them in optimal time. It can be noted that, for a given pivot node i, we first obtain $subtree\text{-}size(i)$ and $count(i)$ in constant time using Lemma 31, 32 respectively and than use the pointer stored as in index in the set L_k, with $k = subtree\text{-}size(i) + count(i)$.

Combining all pieces together we achieve the result summarized in following theorem.

Theorem 10. The collection $\mathcal{D}=\{d_1, d_2, d_3, ..., d_D\}$ of strings of total n characters can be indexed in $(n \log \sigma + O(n))$ bits, such that given a query (P, τ), all maximal generic words can be reported in $O(p + \log \log n + output)$ time.

7.4 Computing Minimal Discriminating Words

In the case of minimal discriminating words, given a query pattern P and a threshold τ, the objective is to find all nodes i in the subtree of locus node i_P such that $count(i) \leq \tau$ and $count(i_{parent}) > \tau$, where i_{parent} is the parent of a node i. Then each of these nodes represent a minimal discriminating word given by $path(i_{parent})$ concatenated with the first leading character on the edge connecting nodes i_{parent} and i. A linear space index with same query bounds as summarized in Lemma 34 can be obtained for minimal discriminating word queries by following the description in Section 7.3.1, except in this scenario, we map each node i in GST to a vertical segment $(i, [count(i), count(i_{parent}) + 1])$. Similarly, the succinct space solution can be obtained by following the same index framework as that of maximal generic words. Below we briefly describe the changes required in the index and query algorithm described in Section 7.3.2 so as to retrieve minimal discriminating words instead of maximal generic words.

We need to maintain all the components of index listed in the proof for Lemma 35 with a single modification. The set $\mathcal{I}_\infty$ consists of segments obtained by mapping each marked node i^* to a vertical segment $(i^*, [count(i^*), count(i^*_{parent})+1])$. By following the same arguments as before, we can rewrite the Lemma 35 as follows:

Lemma 36. The collection $\mathcal{D}=\{d_1, d_2, d_3, ..., d_D\}$ of strings of total n characters can be indexed in $(n \log \sigma + O(n))$ bits, such that given a query (P, τ), we can identify all marked nodes j^* satisfying either of the following condition in $O(p + \log \log n + output)$ time.

- $path(j^*_{parent})$ appended with leading character on edge j^*_{parent}-j^* is a minimal discriminating word for input (P, τ)
- there is at least one node in $N(j^*)$ that corresponds to desired minimal discriminating word, where $N(j^*)$ is a set of all those nodes in GST with j' (unique prime ancestor of j^*) as the first prime node on the path from that node to the root
- j^* is a highest marked descendant of locus node for pattern P

We append the components required in the above lemma by data structure by Sadakane (Lemma 32) to answer $count(.)$ query in constant time for any GST node and retrieve the non-orphan, orphan outputs separately as before.

Though we maintain same collection of bit vectors as required for maximal generic words to retrieve non-orphan outputs, query processing differs slightly in this case. Let i^* be the input marked node and i' be its lowest prime ancestor. Then we can obtain parent-child node pair (u, v) on the path from i' to i^* such that $count(u) > \tau$ and $count(v) \leq \tau$ in constant time. Node v can now be returned as an answer since concatenation of $path(u)$ with first character on edge u-v will correspond to a minimal discriminating word. Thus, every marked node obtained by segment intersection query on set $\mathcal{I}_{\in}$ produces a non-orphan output in this case as opposed to the case of maximal generating words where it may or may not produce a non-orphan output. Also if the input node i^* is highest marked descendant of locus node i_P then we need to verify if node v is within the subtree of i_P before it can be reported as an output.

Data structures and query algorithm to retrieve the orphan outputs remain the same described earlier in Section 7.3.2. It is to be noted that top-g answers to be stored for $\tau = 1, 2, 3, ..., g$ corresponding to each of the distinct pivot node encoding now corresponds to the minimal discriminating word.

Based on the above description, following theorem can be easily obtained.

Theorem 11. The collection $\mathcal{D}$=$\{d_1, d_2, d_3, ..., d_D\}$ of strings of total n characters can be indexed in $(n \log \sigma + O(n))$ bits, such that given a query (P, τ), all minimal discriminating words can be reported in $O(p + \log \log n + output)$ time.

Chapter 8
Pattern Matching with Position Restriction

8.0.1 Problem Formulation and Motivation

Let $T[0...n-1]$ be a text of size n over an alphabet set Σ of size σ. We revisit the well studied *Position-restricted substring searching* (PRSS) problem as defined below:

> The query input consists of a pattern P (of length p) and two indices ℓ and r, and the task is to report all $occ_{\ell,r}$ occurrences of P in $T[\ell...r]$.

Many text searching applications, where the objective is to search only a part of the text collection can be modeled as PRSS problem. For example, restricting the search to a subset of dynamically chosen documents in a document database, restricting the search to only parts of a long DNA sequence, etc [85]. The problem also finds applications in the field of information retrieval as well. The PRSS problem was introduced by Mäkinen and Navarro [85], where they show an elegant reduction of PRSS problem into a two dimensional orthogonal range reporting problem. Their data structure consists of a suffix tree (for initial pattern matching) and an orthogonal range reporting structure in two dimension (RR2D). Their structure takes $O(n)$-word space and the queries can be answered in $O(p + \log n + occ_{\ell,r} \log n)$ query time. By using the most recent two-dimensional range reporting structure by Navarro and Nekrich, this query time can be improved to $O(p + \log^{\epsilon} n + occ_{\ell,r} \log^{\epsilon} n)$, where ϵ is any positive constant. Another trade-off given by Mäkinen and Navarro [85] is $O(n \log^{\epsilon} n)$-word space and near optimal $O(p + \log\log n + occ_{\ell,r})$ query time. This query time is further improved to $O(p+\log\log\sigma+occ_{\ell,r})$ by Kopelowitz et al. [76] and then to optimal $O(p+occ_{\ell,r})$ by Billie and Gortz [14] without changing the space bounds. Crochemore et al. [37] also have proposed an optimal time solution with a much higher space requirement of $O(n^{1+\epsilon})$ bits. Recently, Hon et al. [64] have studied the possibility of indexing the text in succinct space and answering PRSS queries efficiently. They proved that designing a succinct index with poly-logarithmic query bounds is at least as hard as designing a three dimensional range reporting structure in linear space with poly-logarithmic query bounds. However, the provided optimal time and succinct space solutions for a special case where the pattern sufficiently long.

The counting version of $PRSS$ is also an interesting problem. For this, the linear space index by Mäkinen and Navarro [85] takes $O(p + \log n)$ time. A solution by Billie and Gortz [14] can perform counting in faster

$O(p + \log\log n)$ time, which is slightly improved to $O(p + \log\log\sigma)$ by Kopelowitz et al. [76]. However these indexes consumes $O(n \log n/\log^2 \log n)$ words of space. Finally Gagie and Gawrychowski proposed a space efficient solution of $O(n)$ words, where counting queries can be answered in $(p + \log\log n)$ time for general alphabets and in optimal $O(p)$ time, when $\sigma = \log^{O(1)} n$ [53]. We revisit the reporting version of PRSS problem and obtain the result summarized in the following theorem:

Theorem 12. There exist an $O(n \log\sigma)$-word index supporting PRSS queries in $O(p + occ_{\ell,r} \log\log n)$ time.

Using the existing techniques, one can easily design an $O(n \log\log n)$-word space index with $O(p + occ_{\ell,r} \log\log n)$ query time. Therefore, our result is interesting when the alphabet size is small (i.e., when $\sigma = 2^{o(\log\log n)}$). Note that when $\sigma = O(1)$, we achieve exponential time improvement over the previously best-known linear space index.

8.1 The Index

Based on the alphabet size σ and the pattern length p, we consider 3 cases. When alphabet size is greater than $\log n$, we can use the PRSS index by Billie and Gortz [14] which achieves $O(p + occ_{\ell,r} \log\log n)$ query time using $O(n \log\log n)$ words space. Since we only use this index for alphabet size greater than $\log n$, this matches our desired space bound of $O(n \log\sigma)$-words.

For bigger alphabet size, we use two different data structure depending on the pattern length. For bigger patterns, we use the suffix sampling technique to convert PRSS problem into Orthogonal Range Reporting problem in two dimension. For smaller patterns, we encode the text for each possible patterns and use rank/select data structure.

8.1.1 Index for $\sigma = \log^{\Omega(1)} n$

The index consists of a suffix tree ST of the text T. Then for each suffix $T[x...n-1]$, we define a two dimensional point (x, y) such that y be the lexicographic rank of $T[x...n-1]$ among all suffixes of T, and maintain an RR2D structure over these n two-dimensional points using the data structure described in preliminaries. The index space can be bounded by $O(n \log\log n) = O(n \log\sigma)$ words. In order to answer a PRSS query, we first obtain the suffix range $[L, R]$ of P in $O(p)$ time via navigating in the suffix tree and report all those points within the box $[\ell, r-p] \times [L, R]$ by querying on the RR2D structure. The y coordinate of each output is an answer to the original PRSS problem. The query time can be bounded by

$O(p + (occ_{\ell,r} + 1) \log \log n)$. This can be improved to $O(p + occ_{\ell,r} \log \log n)$ by combining the result by Billie and Gortz [14].

Lemma 37. There exist an $O(n \log \sigma)$ space index supporting PRSS queries in $O(p + occ_{\ell,r} \log \log n)$ time for $\sigma = \log^{\Omega(1)} n$.

8.1.2 Index for $\sigma = \log^{O(1)} n$ and $p \geq \sqrt{\log n}$

In this section, we introduce suffix sampling techniques to achieve the desired space bound. Based on a sampling factor $\delta = \lfloor \frac{1}{3} \log_\sigma \log n \rfloor$, we define the followings:

- *δ-sampled suffix:* $T[x...n-1]$ is an δ-sampled suffix if $x \ mod(\delta) = 0$.
- *δ-sampled block (or simply block):* any substring $T[x, x+\delta-1]$ of T of length δ with $x \ mod(\delta) = 0$ is called as a block.

The number of blocks in T is $\Theta(n/\delta)$, where the number of all possible distinct blocks is at most the number of distinct strings of length δ, and is bounded by $\sigma^\delta = O(\log^{1/3} n)$. Let B_i represent the lexicographically ith smallest string of length δ. We categorize the δ-sampled suffixes into σ^δ categories $C_1, C_2, ..., C_{\sigma^\delta}$ such that C_i contains the set of all δ-sampled suffixes whose previous block is B_i. For each category C_i, we maintain a two-dimensional range reporting structures $RR2D_i$'s on the set of points (x, y), where $T[x...n-1]$ is a δ-sampled suffix in C_i and y is its lexicographic rank among all suffixes of T (i.e., $SA[y] = x$). Note that each δ-sampled suffix belongs to exactly one category and the number of δ-sampled suffixes is $\Theta(n/\delta)$. Therefore, the overall space for all $RR2D_i$ structures can be bounded by $\Theta((n/\delta) \log \log n) = \Theta(n \log \sigma)$ words.

Query Answering: Since $p \geq \sqrt{\log n} > \delta$, the starting and ending positions of an occurrence of P will not be in the same block of T. Based on the block in which a match starts, we shall categorize the occurrences into $\sigma^\delta \delta$ types as follows:

> We call an occurrence as a type-(i, j) occurrence, if the prefix of P and the suffix of block B_i matches for j characters.

Here $1 \leq i \leq \sigma^\delta$ and $0 \leq j \leq \delta - 1$. Then type-(i, j) occurrences for a fixed i and j can be retrieved as follows: firstly check if the prefix of P and the suffix of block B_i matches for j characters. This takes

only $O(j) = O(\delta) = O(\log\log n)$ time. If it is not matching, $occ_{i,j} = 0$. Otherwise, corresponding to each type-(i,j) occurrence, there exist a δ-sampled suffix $T[x...n-1]$ of T which is (i) prefixed by $P[j...p-1]$, (ii) occurring after a block B_i, and (iii) $x - j \in [\ell, r]$. By issuing a query on the $RR2D_i$ structure with $[\ell + j, r - p + j] \times [L_j, R_j]$ as the input range, all such suffixes can be retrieved. Here $[L_j, R_j]$ represents the suffix range of $P[j, ..., p-1]$. Note that $[L_j, R_j]$ for $j = 0, 1, ...p-1$ can be computed in total $O(p)$ time. From every reported point (x, y), we shall output $x - j$ as an answer to the original PRSS query. This way, all type-(i,j) occurrences can be reported in $O(\log\log n)$ time plus $O(\log\log n)$ time per output. Hence the total time for reporting all possible type-$(.,.)$ occurrences is bounded by $O(p + \sigma^\delta \delta \log\log n + occ_{\ell,r} \log\log n) = O(p + \sqrt{\log n} + occ_{\ell,r} \log\log n)$.

Lemma 38. There exist an $O(n \log\sigma)$ space index supporting PRSS queries in $O(p + occ_{\ell,r} \log\log n)$ time, given $p \geq \sqrt{\log n}$.

8.1.3 Index for $\sigma = \log^{O(1)} n$ and $p \leq \sqrt{\log n}$

Here we maintain $\sqrt{\log n}$ separate structure for answering PRSS queries for patterns of length $1, 2, 3...., \sqrt{\log n}$. Structure for a fixed pattern length (say $\alpha \leq \sqrt{\log n}$) is described as follows: the number of distinct patterns of length α is σ^α. Each such distinct pattern can be encoded using an integer from $\Sigma_\alpha = \{1, 2, ..., \sigma^\alpha\}$ in $\alpha \log\sigma$ bits. Next we transform the original text $T[0...n-1]$ into $T_\alpha[0...n-\alpha]$, such that

- Each character in T_α is drawn from an alphabet set $\Sigma_\alpha = \{1, 2, ..., \sigma^\alpha\}$ and can be represented in $\alpha \log\sigma$ bits.
- $T_\alpha[i]$: the ith character of T_α is the integer in Σ_α corresponding to the encoding of the string $T[i...i + \alpha - 1]$.

Hence, T_α can be seen as a sequence of $(n - \alpha + 1)$ integers drawn from an alphabet set Σ_α. We shall maintain T_α in $O(|T_\alpha| \log|\Sigma_\alpha|) = O(n\alpha \log\sigma)$ bits using the data structure described in [56], such that rank/select queries on any character within T_α can be supported in $O(\log\log|\Sigma_\alpha|) = O(\log\log n)$ time. Since we are maintaining T_α for $\alpha = 1, 2, 3, ..., \sqrt{\log n}$, the total space can be bounded by $O(n \log\sigma \sum_{\alpha=1}^{\sqrt{\log n}} \alpha)$ $= O(n \log\sigma \log n)$ bits or $O(n \log\sigma)$ words.

Query Answering: A PRSS query for a pattern P of length $p \leq \sqrt{\log n}$ can be answered as follows: first we find the integer β in Σ_p corresponding to P in $O(p)$ time. An occurrence of β at position i in T_α

corresponds to an occurrence of P at position i in T. Therefore, all occurrences of P in $T[\ell...r]$ can be answered by reporting all those occurrences of β in $T_\alpha[\ell...r]$. Using rank/select queries on T_α, this can be easily handled as follows: find the number of occurrences (say a) of β in T_α before the position ℓ and the number of occurrences (say b) of β until the position r in T_α. Using two rank queries on T_α, the values of a and b can be obtained in $O(\log\log n)$ time. Next we output the kth occurrence of β in T_α for $k = a+1, a+2, ..., b$ using $(b-a)$ select queries in total $O((b-a)\log\log n)$ time, and each output corresponds to an answer to the original PRSS problem. Hence the total query time can be bounded by $O(p + \log\log n + occ_{\ell,r}\log\log n)$. In order to remove the additive $\log\log n$ factor, we also maintain the linear space structure by Gagie and Gawrychowski [53] for counting the number of outputs in $O(p)$ time (for poly-logarithmic alphabet). And while querying, we first count the output size using this structure, and then we query on our structure only if the output size is non zero.

Lemma 39. There exist an $O(n\log\sigma)$ space index supporting PRSS queries in $O(p + occ_{\ell,r}\log\log n)$ time, for $p \leq \sqrt{\log n}$.

By combining Lemma 37, Lemma 38 and Lemma 39, we obtain Theorem 1.

8.2 Semi Dynamic index for Property Matching

In property matching problem, in addition to the text T, a set $\pi = \{[s_1, e_1], [s_2, e_2], ...\}$ of intervals is also given. Our task is to preprocess T and π and maintain an index, such that when ever a pattern P (of length p) comes as a query, return those occurrences of P in T which are within (at least) one of the interval in π. Efficient linear space [6, 71] and compressed space [61] indexes are known for this problem. In [75], Kopelowitz have studied the dynamic case of this problem, where π can be updated (i.e., intervals can be inserted to or deleted from π), and provide a linear space index with $O(e - s + \log\log n)$ update time, where (s, e) be the interval inserted/deleted. In semi-dynamic case, (i.e., only insertions or deletions) the update time is $O(e-s)$. Note that $e-s$ can be even $\Theta(n)$ in the worst case. We describe a semi-dynamic index (only insertions) with the result summarized in the following theorem. We use position-restricted substring queries as one of the main technique to achieve this result.

Theorem 13. There exists an $O(n\log^\epsilon n)$ space index for semi-dynamic (only insertions) property matching with query time $O(p + \sqrt{n}\log\log n + occ_\pi)$ and update time $O(\sqrt{n})$, where occ_π represents the output size.

We use the following two existing structures as the building blocks of our index.

Lemma 40. [75] There exists a linear space index (which we call as I_{SDPM}) for semi-dynamic (only insertions) property matching with query time $O(p + occ_{\pi})$ and update time $O(e - s)$, where (e, s) is the inserted interval.

Lemma 41. [15] There exists an $O(n \log^{\epsilon} n)$ space index (which we call as I_{PRSS}) supporting PRSS queries in optimal $O(p + occ_{\ell,r})$ time, where $\epsilon > 0$ is any constant.

Our index consists of I_{SDPM}, and I_{PRSS}. An interval (e, s) be called as *small* if $e - s \leq \sqrt{n}$, otherwise it is *large*. Note that a small interval can be inserted into π, and can update I_{SDPM} in $O(\sqrt{n})$ time. However, if the interval comes is $large$, we do not update I_{SDPM}. Actually we insert it into a set S of intervals. When ever the size of S grows up to $\sqrt{n}$, we do a batched insertions of all the intervals in S into I_{SDPM} using the following lemma, and then initialize S to a tree with zero nodes.

Lemma 42. [75] A set I of intervals can be inserted into π, and can update I_{SDPM} in time $O(|coversize(I)|)$, where $coversize(I)$= $\{i \in [1, n] : \exists [s, e] \in I$, such that $i \in [s, e]\}$.

Here $|cover.size(I)|$ can be at most size of the text, n. Batch insertion will be required after $\sqrt{n}$ number of insertions. Therefore, from Lemma 42, the time for batch insertion can be bounded by $O(n)$, and the time for insertion of a single interval can be bounded by $O(\sqrt{n})$ in amortized sense.

Query Answering: Our algorithm has the following 3 phases:

1. Issue a query on I_{SDPM} and retrieve the corresponding occurrences. However, we cannot obtain the complete outputs just by this step alone, as we do not (always) update I_{SDPM} immediately after an insertion.

2. The task of finding those missing occurrences, which are from the region corresponding to those $large$ intervals in S are obtained in this Step. Firstly we sort all intervals in S in $O(\sqrt{n})$ time as follows: if $|S| < \sqrt{n}/\log n$, then the time for sorting is $O(|S| \log |S|) = O(\sqrt{n})$, otherwise use radix sort and the required time is $O(|S|) = O(\sqrt{n})$. After initializing a variable h as 1 (this is an invariant storing the last occurrence retrieved so far), we perform the following steps for each interval $[s_i, e_i] \in S$ (where $s_i \leq s_{i+1}$) in the ascending order of i: if $h \in [s_i, e_i]$, then issue a PRSS query with $[h + 1, e_i - p + 1]$ as the input range, and if $h < s_i$, then issue another PRSS query with $[s_i, e_i - p + 1]$ as the input range, where as if $h > e_i$, skip this step. Based on the retrieved occurrences, we update h with the

last occurrence. The over all time required for this step can be bounded by $O(|S| \log\log n + occ_\pi)$ = $O(\sqrt{n} \log\log n + occ_\pi)$.

3. There can be occurrences which are retrieved by both Step 1 and 2. In order to eliminate these duplicated, we sort the occurrences retrieved from Step 1 and Step 2 separately (in $O(\sqrt{n} + occ_\pi)$ time as described before), and then merge these results without duplicates by scanning both lists simultaneously.

By combining all, the query time can be bounded by $O(p + \sqrt{n} \log\log n + occ_\pi)$. This completes the proof of Theorem 2. □

Those occurrences which are within large intervals can be retrieved as follows: start scanning the intervals within S in the ascending order of the starting point. Let (s'_1, e'_1) be the first interval, then we perform a PRSS query on I_{PRSS} with $[s'_1, e'_1 - p + 1]$ as the query interval and retrieve all the outputs.

When a query pattern P comes, we first query *PST* on P and then we query I_{PRSS} for each interval in L. The union of all the occurrences obtained is our solution for property matching. In order to avoid redundancy, we keep the list L according the increasing order of the left boundary of each interval. That means that suppose $[i_1, j_1]$, $[i_2, j_2]$,...,$[i_{|L|}, j_{|L|}]$ are the intervals; then $i_1 \leq i_2 \leq ... \leq i_{|L|}$. Then, when we query I_{PRSS} on P with each interval one by one, we use a variable h to record the largest position which is an occurrence that we found to the current point. When we query P with the next interval $[i_k, j_k]$, we first check whether h is in $[i_k, j_k]$, where $1 < k \leq |L|$. If h is in $[i_k, j_k]$, instead of querying P with $[i_k, j_k]$, we query P with $[h + 1, j_k]$. Else, if $h < i_k$, we query P with $[i_k, j_k]$. Otherwise, $h \geq j_k$, we skip the interval $[i_k, j_k]$ because this means $[i_k, j_k]$ is covered by the previous interval. Thus we continue to check the next interval with h by the same scheme.

Chapter 9
Uncertain String Indexing

9.1 Overview

In this chapter, we explore the problem of indexing uncertain strings for efficient searching. We begin by describing uncertainty string model, possible world semantics and challenges of uncertain string searching.

Existing research has focused mainly on the study of regular or deterministic string indexing. In this chapter we explore the problem of indexing uncertain strings. We begin by describing the uncertain string model, possible world semantics and challenges of searching in uncertain strings.

Current literature models uncertain strings in two different ways: the string level model and the character level model. In string level model, we look at the probabilities and enumerate at whole string level, whereas character level model represents each position as a set of characters with associated probabilities. We focus on the character level model which arises more frequently in applications. Let S be an uncertain string of length n. Each character c at position i of S has an associated probability $pr(c^i)$. Probabilities at different positions may or may not contain correlation among them. Figure 1(a) shows an uncertain string S of length 5. Note that, the length of an uncertain string is the total number of positions in the string, which can be less than the total number of possible characters in the string. For example, in Figure 1(a), total number of characters in string s with nonzero probability is 9, but the total number of positions or string length is only 5.

"Possible world semantics" is a way to enumerate all the possible deterministic strings from an uncertain string. Based on possible world semantics, an uncertain string S of length n can generate a deterministic string w by choosing one possible character from each position and concatenating them in order. We call w as one of the possible world for S. Probability of occurrence of $w = w_1 w_2 \dots w_n$ is the partial product $pr(w_1^1) \times pr(w_2^2) \times \cdots \times pr(w_n^n)$. The number of possible worlds for S increases exponentially with n. Figure 1(b) illustrates all the possible worlds for the uncertain string S with their associated probability of occurrence.

A meaningful way of considering only a fraction of the possible worlds is based on a probability threshold value τ. We consider a generated deterministic string $w = w_1 w_2 \dots w_n$ as a valid occurrence with respect to τ, only if it has probability of occurrence more than τ. The probability threshold τ effectively removes lower

probability strings from consideration. Thus τ plays an important role to avoid the exponential blowup of the number of generated deterministic strings under consideration.

Character	S[1]	S[2]	S[3]	S[4]	S[5]
a	.3	.6	0	.5	1
b	.4	0	0	0	0
c	0	.4	0	.5	0
d	.3	0	1	0	0

(a) Uncertain string S

w	Prob(w)	w	Prob(w)	w	Prob(w)
w[1] aadaa	.09	w[5] badaa	.12	w[9] dadaa	.09
w[2] aadca	.09	w[6] badca	.12	w[10] dadca	.09
w[3] acdaa	.06	w[7] badca	.08	w[11] dcdaa	.06
w[4] acdca	.06	w[8] badca	.08	w[12] dcdca	.06

(b) Possible worlds of S

FIGURE 9.1. An uncertain string S of length 5 and its all possible worlds with probabilities.

Given an uncertain string S and a deterministic query substring $p = p_1 \dots p_m$, we say that p matched at position i of S with respect to threshold τ if $pr(p_1^i) \times \cdots \times pr(p_m^{i+m-1}) \geq \tau$. Note that, $O(m + occ)$ is the theoretical lower bound for substring searching where m is the substring length and occ is the number of occurrence reported.

9.1.1 Formal Problem Definition

Our goal is to develop efficient indexing solution for searching in uncertain strings. In this chapter, we discuss two basic uncertain string searching problems which are formally defined below.

Problem 2 (Substring Searching)**.** Given an uncertain string S of length n, our task is to index S so that when a deterministic substring p and a probability threshold τ come as a query, report all the starting positions of S where p is matched with probability of occurrence greater than τ.

Problem 3 (Uncertain String Listing)**.** Let $\mathcal{D}$=$\{d_1,\dots,d_D\}$ be a collection of D uncertain strings of n positions in total. Our task is to index $\mathcal{D}$ so that when a deterministic substring p and a probability threshold τ come as a query, report all the strings d_j such that d_j contains atleast one occurrence of p with probability of occurrence greater than τ.

Note that the string listing problem can be naively solved by running substring searching query in each of the uncertain string from the collection. However, this naive approach will take $O(\sum_{d_i \in \mathcal{D}}$ search time on $d_i)$ time which can be very inefficient if the actual number of documents containing the substring is small. Figure 9.2 illustrates an example for string listing. In this example, only the string d_1 contains query substring "BF" with probability of occurrence greater than query threshold 0.1. Ideally, the query time should be

proportionate to the actual number of documents reported as output. Uncertain strings considered in both these problems can contain correlation among string positions.

String collection $\mathcal{D} = \{d_1, d_2, d_3\}$:

$d_1[1]$	$d_1[2]$	$d_1[3]$	$d_2[1]$	$d_2[2]$	$d_2[3]$	$d_3[1]$	$d_3[2]$	$d_3[3]$
A .4	B .3	F .5	A .6	B .5	B .4	A .4	I .3	A 1
B .3	L .3	J .5	C .4	F .3	C .3	F .4	L .3	
F .3	F .3			J .2	E .2	P .2	P .3	
	J .1				F .1		T .3	

Output of string listing query $(\text{"}BF\text{"}, 0.1)$ on $\mathcal{D}$ is: d_1

FIGURE 9.2. String listing from an uncertain string collection $\mathcal{D} = \{d_1, d_2, d_3\}$.

9.1.2 Challenges in Uncertain String Searching

We summarize some challenges of searching in uncertain strings.

- An uncertain string of length n can have multiple characters at each position. As the length of an uncertain string increases, the number of possible worlds grows exponentially. This makes a naive technique that exhaustively enumerates all possible worlds infeasible.

- Since multiple substrings can be enumerated from the same starting position, care should be taken in substring searching to avoid possible duplication of reported positions.

- Enumerating all the possible sequences for arbitrary probability threshold τ and indexing them requires massive space for large strings. Also note that, for a specific starting position in the string, the probability of occurrence of a substring can change arbitrarily (non-decreasing order) with increasing length, depending on the probability of the concatenated character This makes it difficult to construct index that can support arbitrary probability threshold τ.

- Correlated uncertainty among the string positions is not uncommon in applications. An index that handles correlation appeals to a wider range of applications. However, handling the correlation can be a bottleneck on space and time.

9.1.3 Related Work

Although, searching over clean data has been widely researched, indexing uncertain data is relatively new. Below we briefly mention some of the previous works related to uncertain strings.

Algorithmic Approach Li et al. [83] tackled the substring searching problem where both the query substring and uncertain sequence comes as online query. They proposed a dynamic programming approach to

calculate the probability that a substring is contained in the uncertain string. Their algorithm takes linear time and linear space.

Approximate substring Matching Given as input a string p, a set of strings $\{x_i|1 \leq i \leq r\}$, and an edit distance threshold k, the substring matching problem is to find all substrings s of x_i such that $d(p, s) \leq k$, where $d(p, s)$ is the edit distance between p and s. This problem has been well studied on clean texts (see [95] for a survey). Most of the ideas to solve this problem is based on partitioning p. Tiangjian et al. [55] extended this problem for uncertain strings. Their index can handle strings of arbitrary lengths.

Frequent itemset mining Some articles discuss the problem of frequent itemset mining in uncertain databases [33, 34, 80, 13], where an itemset is called frequent if the probability of occurrence of the itemset is above a given threshold.

Probabilistic Database Several works ([32, 114, 113]) have developed indexing techniques for probabilistic databases, based on R-trees and inverted indices, for efficient execution of nearest neighbor queries and probabilistic threshold queries. Dalvi et al. [38] proposed efficient evaluation method for arbitrary complex SQL queries in probabilistic database. Later on efficient index for ranked top-k SQL query answering on a probabilistic database was proposed([107, 82]). Kanagal et al. [72] developed efficient data structures and indexes for supporting inference and decision support queries over probabilistic databases containing correlation. They use a tree data structure named junction tree to represent the correlations in the probabilistic database over the tuple-existence or attribute-value random variables.

Similarity joins A string similarity join finds all similar string pairs between two input string collections. Given two collections of uncertain strings R, S, and input (k, τ), the task is to find string pairs (r, s) between these collections such that $Pr(ed(R, S) \leq k) > \tau$ i.e., probability of edit distance between R and S being at most k is more than probability threshold τ. There are some works on string joins, e.g., ([26, 57, 81]), involving approximation, data cleaning, and noisy keyword search, which has been discussed in the probabilistic setting([70]). Patil et al. [104] introduced filtering techniques to give upper and (or) lower bound on $Pr(ed(R, S) \leq k)$ and incorporate such techniques into an indexing scheme with reduced filtering overhead.

9.1.4 Our Approach

Since uncertain string indexing is more complex than deterministic string indexing, a general solution for substring searching is challenging. However efficiency can be achieved by tailoring the data structure based on some key parameters, and use the data structure best suited for the purposed application. We consider the following parameters for our index design.

Threshold parameter τ_{min}. The task of substring matching in uncertain string is to find all the probable occurrences, where the probable occurrence is determined by a query threshold parameter τ. Although τ can have any value between 0 to 1 at query time, real life applications usually prohibits arbitrary small value of τ. For example, a monitoring system does not consider a sequence of events as a real threat if the associated probability is too low. We consider a threshold parameter τ_{min}, which is a constant known at construction time, such that query τ does not fall below τ_{min}. Our index can be tailored based on τ_{min} at construction time to suit specific application needs.

Query substring length. The query substring searched in the uncertain string can be of arbitrary length ranging from 1 to n. However, most often the query substrings are smaller than the indexed string. An example is a sensor system, collecting and indexing big amount of data to facilitate searching for interesting query patterns, which are smaller compared to the data stream. We show that more efficient indexing solution can be achieved based on query substring length.

Correlation among string positions. Probabilities at different positions in the uncertain string can possibly contain correlation among them. In this chapter we consider character level uncertainly model, where a probability of occurrence of a character at a position can be correlated with occurrence of a character at a different position. We formally define the correlation model and show how correlation is handled in our indexes.

Our approach involves the use of suffix trees, suffix arrays and range maximum query data structure, which to the best of our knowledge, is the first use for uncertain string indexing. Succinct and compressed versions of these data structures are well known to have good practical performance. Previous efforts to index uncertain strings mostly involved dynamic programming and lacked theoretical bound on query time. We also formulate the uncertain string listing problem. Practical motivation for this problem is given in Section 9.6. As mentioned before, for a specific starting position of an uncertain string, the probability of occurrence of a substring can change arbitrarily with increasing length depending on the probability of the

concatenated character. We propose an approximate solution by discretizing the arbitrary probability changes with conjunction of a linking structure in the suffix tree.

9.1.5 Our Contribution:

In this chapter, we propose indexing solutions for substring searching in a single uncertain string, searching in a uncertain string collection, and approximate index for searching in uncertain strings. More specifically, we make the following contributions:

1. For the substring searching problem, we propose a linear space solution for indexing a given uncertain string S of length n, such that all the occurrences of a deterministic query string p with probability of occurrence greater than a query threshold τ can be reported. We show that for frequent cases our index achieves optimal query time proportional to the substring length and output size. Our index can be designed to support arbitrary probability threshold $\tau \geq \tau_{min}$, where τ_{min} is a constant known at index construction time.

2. For the uncertain string listing problem, given a collection of uncertain strings $\mathcal{D} = \{d_1, \ldots, d_D\}$ of total size n, we propose a linear space and near optimal time index for retrieving all the uncertain strings that contain a deterministic query string p with probability of occurrence greater than a query threshold τ. Our index supports queries for arbitrary $\tau \geq \tau_{min}$, where τ_{min} is a constant known at construction time.

3. We propose an index for approximate substring searching, which can answer substring searching queries in uncertain strings for arbitrary $\tau \geq \tau_{min}$ in optimal $O(m + occ)$ time, where τ_{min} is a constant known at construction time and ϵ is the bound on desired additive error in the probability of a matched string, i.e. outputs can have probability of occurrence $\geq \tau - \epsilon$.

9.1.6 Outline

The rest of the chapter is organized as follows. In section 9.2 we show some practical motivations for our indexes. In section 9.3 we give a formal definition of the problem, discuss some definitions related to uncertain strings and supporting tools used in our index. In section 9.4 we build a linear space index for answering a special form of uncertain strings where each position of the string has only one probabilistic character. In section 9.5 we introduce a linear space index to answer substring matching queries in general

uncertain strings for variable threshold. Section 9.6 discusses searching in an uncertain string collection. In section 9.7, we discuss approximate string matching in uncertain strings. In section 9.8, we show the experimental evaluation of our indexes.

9.2 Motivation

Various domains such as bioinformatics, knowledge discovery for moving object database trajectories, web log analysis, text mining, sensor networks, data integration and activity recognition generates large amount of uncertain data. Below we show some practical motivation for our indexes.

Biological sequence data Sequence data in bioinformatics is often uncertain and probabilistic. For instance, reads in shotgun sequencing are annotated with quality scores for each base. These quality scores can be understood as how certain a sequencing machine is about a base. Probabilities over long strings are also used to represent the distribution of SNPs or InDels (insertions and deletions) in the population of a species. Uncertainty can arise due to a number of factors in the high-throughput sequencing technologies. NC-IUB committee standardized incompletely specified bases in DNA([84]) to address this common presence of uncertainty. Analyzing these uncertain sequences is important and more complicated than the traditional string matching problem.

We show an example uncertain string generated by aligning genomic sequence of Tree of At4g15440 from OrthologID and deterministic substring searching in the sequence. Figure 9.3 illustrates the example.

S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	S[9]	S[10]	S[11]
P 1	S .7	F 1	P 1	Q .5	P 1	A .4	I .3	A 1	S .5	A 1
	F .3			T .5		F .4	L .3		T .5	
						P .2	P .3			
							T .3			

FIGURE 9.3. Example of an uncertain string S generated by aligning genomic sequence of the tree of At4g15440 from OrthologID.

Consider the uncertain string S of Figure 9.3. A sample query can be $\{p = "AT", \tau = 0.4\}$, which asks to find all the occurrences of string AT in S having probability of occurrence more than $\tau = .4$. AT can be matched starting at position 7 and starting at position 9. Probability of occurrence for starting position 7 is $0.4 \times 0.3 = 0.12$ and for starting position 9 is $1 \times 0.5 = 0.5$. Thus position 9 should be reported to answer this query.

Automatic ECG annotations In the Holter monitor application, for example, sensors attached to heart-disease patients send out ECG signals continuously to a computer through a wireless network([39]). For each heartbeat, the annotation software gives a symbol such as N (Normal beat), L (Left bundle branch block beat), and R, etc. However, quite often, the ECG signal of each beat may have ambiguity, and a probability distribution on a few possibilities can be given. A doctor might be interested in locating a pattern such as NNAV indicating two normal beats followed by an atrial premature beat and then a premature ventricular contraction, in order to verify a specific diagnosis. The ECG signal sequence forms an uncertain string, which can be indexed to facilitate deterministic substrings searching.

Event Monitoring Substring matching over event streams is important in paradigm where continuously arriving events are matched. For example a RFID-based security monitoring system produces stream of events. Unfortunately RFID devices are error prone, associating probability with the gathered events. A sequence of events can represent security threat. The stream of probabilistic events can be modeled with uncertain string and can be indexed so that deterministic substring can be queried to detect security threats.

9.3 Preliminaries

9.3.1 Uncertain String and Deterministic String

An uncertain string $S = s_1 \dots s_n$ over alphabet Σ is a sequence of sets $s_i, i = 1, \dots, n$. Every s_i is a set of pairs of the form $(c_j, pr(c_j^i))$, where every c_j is a character in Σ and $0 \leq pr(c_j^i) \leq 1$ is the probability of occurrence of c_j at position i in the string. Uncertain string length is the total number of positions in the string, which can be less than the total number of characters in the string. Note that, summation of probability for all the characters at each position should be 1, i.e. $\sum_j pr(c_j^i) = 1$. Figure 9.3 shows an example of an uncertain string of length 11. A deterministic string has only one character at each position with probability 1. We can exclude the probability information for deterministic strings.

9.3.2 Probability of Occurrence of a Substring in an Uncertain String

Since each character in the uncertain string has an associated probability, a deterministic substring occurs in the uncertain string with a probability. Let $S = s_1 \dots s_n$ is an uncertain string and p is a deterministic string. If the length of p is 1, then probability of occurrence of p at position i of S is the associated probability $pr(p^i)$. Probability of occurrence of a deterministic substring $p = p_1 \dots p_k$, starting at a position i in S is

defined as the partial product $pr(p_1^i) \times \cdots \times pr(p_k^{i+k-1})$. For example in Figure 9.3, $SFPQ$ has probability of occurrence $0.7 \times 1 \times 1 \times 0.5 = 0.35$ at position 2.

9.3.3 Correlation Among String Positions.

We say that character c_k at position i is correlated with character c_l at position j, if the probability of occurrence of c_k at position i is dependent on the probability of occurrence of c_l at position j. We use $pr(c_k^i)^+$ to denote the probability of c_k^i when the correlated character is present, and $pr(c_k^i)^-$ to denote the probability of c_k^i when the correlated character is absent. Let $x_g \ldots x_h$ be a the substring generated from an uncertain string. c_k^i,$g \leq i \leq h$ is a character within the substring which is correlated with c_l^j. Depending on the position j, we have 2 cases:

Case 1, $g \leq j \leq h$ **:** The correlated probability of (c_k^i) is expressed by $(c_l^j \implies a\ , \neg c_l^j \implies b)$, i.e. if c_l^j is taken as an occurrence, then $pr(c_k^i) = pr(c_k^i)^+$, otherwise $pr(c_k^i) = pr(c_k^i)^-$. We consider a simple example in Figure 9.4 to illustrate this. In this string, z^3 is correlated with e^1. For the substring eqz, $pr(z^3) = .3$, and for the substring fqz, $pr(z^3) = .4$.

Case 1, $j < g$ **or** $j > h$ **:** c_l^j is not within the substring. In this case, $pr(c_k^i)$=$pr(c_l^j)$*$pr(c_k^i)^+$+$(1-pr(c_l^j))$*$pr(c_k^i)^+$. In Figure 9.4, for substring qz, $pr(z^3) = .6 * .3 + .4 * .4$.

S[1]	S[2]	S[3]
e: .6 f: .4	q: 1	z: $e^1 \implies .3, \neg e^1 \implies .4$

FIGURE 9.4. Example of an uncertain string S with correlated characters.

9.3.4 Suffix Trees and Generalized Suffix Trees

The suffix tree [116, 89] of a deterministic string $t[1 \ldots n]$ is a lexicographic arrangement of all these n suffixes in a compact trie structure of $O(n)$ words space, where the i-th leftmost leaf represents the i-th lexicographically smallest suffix of t. For a node i (i.e., node with pre-order rank i), $path(i)$ represents the text obtained by concatenating all edge labels on the path from root to node i in a suffix tree. The locus node i_p of a string p is the node closest to the root such that the p is a prefix of $path(i_P)$. The suffix range of a string p is given by the maximal range $[sp, ep]$ such that for $sp \leq j \leq ep$, p is a prefix of (lexicographically)

j-th suffix of t. Therefore, i_p is the lowest common ancestor of sp-th and ep-th leaves. Using suffix tree, the locus node as well as the suffix range of p can be computed in $O(p)$ time, where p denotes the length of p. The suffix array A of t is defined to be an array of integers providing the starting positions of suffixes of S in lexicographical order. This means, an entry $A[i]$ contains the position of i-th leaf of the suffix tree in t. For a collection of strings $\mathcal{D} = \{d_1, \ldots, d_D\}$, let $t = d_1 d_2 \ldots d_D$ be the text obtained by concatenating all the strings in $\mathcal{D}$. Each string is assumed to end with a special character \$. The suffix tree of t is called the generalized suffix tree (GST) of $\mathcal{D}$.

9.4 String Matching in Special Uncertain Strings

In this section, we construct index for a special form of uncertain string which is extended later. Special uncertain string is an uncertain string where each position has only one probable character with associated non-zero probability of occurrence. Special-uncertain string is defined more formally below.

Problem 5 (Special uncertain string)**.** A special uncertain string $X = x_1 \ldots x_n$ over alphabet Σ is a sequence of pairs. Every x_i is a pair of the form $(c_i, pr(c_i^i))$, where every c_i is a character in Σ and $0 < pr(c_i^i) \leq 1$ is the probability of occurrence of c_i at position i in the string.

Before we present an efficient index, we discuss a naive solution similar to deterministic substring searching.

9.4.1 Simple Index

Given a special uncertain string $X = x_1 \ldots x_n$, construct the deterministic string $t = c_1 \ldots c_n$ where c_i is the character in x_i. We build a suffix tree over t. We build a suffix array A which maps each leaf of the suffix tree to its original position in t. We also build a successive multiplicative probability array C, where $C[j] = \prod_{i=1}^{j} Pr(c_i^i)$, for $j = 1, \ldots, n$. For a substring $x_i \ldots x_{i+j}$, probability of occurrence can be easily computed by $C[i+j]/C[i-1]$. Given an input (p, τ), we traverse the suffix tree for p and find the locus node and suffix range of p in $O(m)$ time, where m is the length of p. Let the suffix range be $[sp, ep]$. According to the property of suffix tree, each leaf within the range $[sp, ep]$ contains an occurrence of p in t. Original positions of the occurrence in t can be found using suffix array, i.e., $A[sp], \ldots, A[ep]$. However, each of these occurrence has an associated probability. We traverse each of the occurrence in the range $A[sp], \ldots, A[ep]$. For an occurrence $A[i]$, we find the probability of occurrence by $C[A[i]+m-1]/C[A[i]-1]$. If the probability of occurrence is greater than τ, we report the position $A[i]$ as an output. Figure 9.5 illustrates this approach.

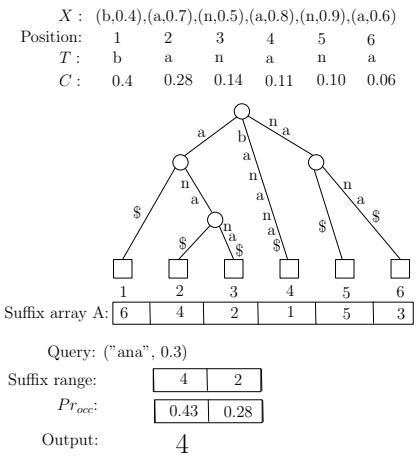

FIGURE 9.5. Simple index for special uncertain strings.

Handling Correlation: Correlation is handled when we check for the probability of occurrence. If $A[i]$ is a possible occurrence, then we need to consider any existing character within the substring $x_i \dots x_{i+m-1}$, that is correlated with another character. Let c_k is a character at position j within $x_i \dots x_{i+m-1}$, which is correlated with character $c_l^{j'}$, i.e. if $c_l^{j'}$ is included in the substring, then $pr(c_k^j)$=$pr(c_k^j)^+$, or else $pr(c_k^j)$=$pr(c_k^j)^-$. To find the correct probability of (c_k^j), if j' we check the j'-th position (j' depth character on the root to locus path in the suffix tree) of the substring. If the j'-th character is c_l, then $C[A[i]+m-1]/C[A[i]-1]$ is the correct probability of occurrence for $x_i \dots x_{i+m}$. Otherwise, $C[A[i]+m-1]/C[A[i]-1]$ contains the incorrect probability of c_k^j. Dividing $C[A[i]+m-1]/C[A[i]-1]$ by $pr(c_k^j)^+$ and multiplying by $pr(c_k^j)^-$ gives the correct probability of occurrence in this case. If c_l falls before or after the substring $x_i \dots x_{i+m-1}$, $pr(c_k^j)$=$pr(c_l^{j'})$*$pr(c_k^j)^+$+$(1-pr(c_l^{j'}))$*$pr(c_k^j)^-$. Dividing $C[A[i]+m-1]/C[A[i]-1]$ by $pr(c_k^j)^+$ and multiplying by $pr(c_k^j)$ gives the correct probability of occurrence. Note that, we can identify and group all the characters with existing correlation, and search in the suffix tree in one scan for improved efficiency.

The main drawback in this approach is the query time. Within the suffix range $[sp, ep]$, possibly very few number of positions can qualify as output because of τ. So spending time on each element of the range $[sp, ep]$ is not justifiable.

9.4.2 Efficient Index:

Bottleneck of the simple index comes from traversing each element within the suffix range. For the efficient index, we iteratively retrieve the element with maximum probability of occurrence in the range in constant time. Whenever the next maximum probability of occurrence falls below τ, we conclude our search. We use range maximum query (RMQ) data structure for our index which is briefly explained below.

Range Maximum Query

Let B be an array of integers of length n, a range maximum query(RMQ) asks for the position of the maximum value between two specified array indices $[i, j]$. i.e., the RMQ should return an index k such that $i \leq k \leq j$ and $B[k] \geq B[x]$ for all $i \leq x \leq j$. We use the result captured in following lemma for our purpose.

Lemma 43. [46, 47] By maintaining a $2n + o(n)$ bits structure, range maximum query(RMQ) can be answered in $O(1)$ time (without accessing the array).

Every leaf of the suffix tree denotes a suffix position in the original text and a root to leaf path represents the suffix. For uncertain string, every character in this root to leaf path has an associated probability which is not stored in the suffix tree. Let y_j^i, for $j = 1, \ldots, n$ denote a deterministic substring which is the i-length prefix of the j-th suffix,i.e. the substring on the root to i-th leaf path. Let Y^i is the set of y_j^i, for $j = 1, \ldots, n$.

For $i = 1, \ldots, n$, we define C_i as the successive multiplicative probability array for the substrings of Y^i. j-th element of C_i is the successive multiplicative probability of the i-length prefix of the j-th suffix. More formally $C_i[j] = \prod_{k=A[j]}^{A[j]+i-1} Pr(c_k^k) = C[A[j]+i-1]/C[A[j]-1](1 \leq j \leq n)$. For each $C_i(i = 1, \ldots, \log n)$ we use range maximum query data structure RMQ_i of n bits over C_i and discard the original array C_i. Note that, RMQ data structure can be built over an integer array. We convert C_i into an integer array by multiplying each element by a sufficiently large number and then build the RMQ_i structure over it. We obtain $\log n$ number of such RMQ data structures resulting in total space of $O(n \log n)$ bits or $O(n)$ words. We also store the global successive multiplicative probability array C, where $C[j] = \prod_{i=1}^{j} Pr(c_i^i)$. Given a query (p, τ), Our idea is to use RMQ_i for iteratively retrieving maximum probability of occurrence elements in constant time each and validate using C. To maintain linear space, we can support query substring length of $m = 0, \ldots, \log n$ in this approach. Algorithm 1 illustrates the index construction phase for short substrings.

Query Answering

Short substrings ($m \leq \log n$) : Given an input (p, τ), we first retrieve the suffix range $[l, r]$ in $O(m)$ time using suffix tree, where m is the length of p. We can find the maximum probability occurrence of p in $O(1)$ time by executing query $RMQ_m(l, r)$. Let max be the position of maximum probability occurrence and $max' = A[max]$ be the the original position in t. We can find the corresponding probability of occurrence by $C[max' + i - 1]/C[max' - 1]$. If the probability is less that τ, we

conclude our search. If it is greater than τ, we report max' as an output. For finding rest of the outputs, we recursively search in the ranges $[l, max-1]$ and $[max+1, r]$. Since each call to $RMQ_m(l,r)$ takes constant time, validating the probability of occurrence takes constant time, we spend $O(1)$ time for each output. Total query time is optimal $O(m+occ)$. Algorithm 2 illustrates the query answering for short substrings. Note that, correlation is handled in similar way as described for the naive index, and we omit the details here.

Algorithm 1: Algorithm Special-Short-Substring-Index-Construction

input : A special uncertain string X
output : suffix tree, suffix array A, successive multiplicative probability array C, RMQ_i $(i = 1, \ldots, \log n)$

Build deterministic string t from X
Build suffix tree over t
Build suffix array A over t
// Building successive multiplicative probability array
$C[1] = Pr(c_1^1)$
for $i = 2; i \leq n; i++$ **do**
  $C[i] = C[i-1] \times Pr(c_i^i)$
end
// Building C_i array for $i = 1, \ldots, \log n$
for $i = 1; i \leq \log n; i++$ **do**
  for $j = 1; j \leq n; j++$ **do**
    $C_i[j] = C[A[j]+i-1]/C[A[j]-1]$
    // Handling correlated characters
    for *all character* c_a^k *in* $t[A[j] \ldots t[A[j]+i-1]$ *that are correlated with another character* c_b^l **do**
      if ($A[j] \leq l \leq [A[j]+i-1]$ and c_b^l is not within $t[A[j] \ldots t[A[j]+i-1])$
      $C_i[j] = C_i[j]/Pr(c_a^k)^+ * Pr(c_a^k)^-$
      else
      $pr(c_a^k)$=$pr(c_b^l)$*$pr(c_a^k)^+$+$(1-pr(c_b^l))$*$pr(c_a^k)^-$
      $C_i[j] = C_i[j]/Pr(c_a^k)^+ * Pr(c_a^k)$
      end
    end
  end
  Build RMQ_i over C_i
end

Long substrings ($m > \log n$) : We use a blocking scheme for answering long query substrings ($m > \log n$). Since exhaustively enumerating all possible substrings and storing the probabilities for each of them is infeasible, we only store selective probability values at construction time and compute the others at query time. We partition the entire suffix range of suffix array into different size blocks. More formally, for $i = \log n, \ldots, n$, we divide the suffix range $[1, n]$ of suffix array $A[1, n]$ into $O(n/i)$

Algorithm 2: Algorithm Special-Short-Substring-Query-Answering

input : Query substring p, probability threshold τ)
output : Occurrence positions of p in X with probability of occurrence greater than τ
$m = length(p)$
call RecursiveRmq($m, 1, n$)
function RECURSIVERMQ(i, l, r) ▷ Recursive RMQ method
$max = RMQ_m(l, r)$
$max' = A[max]$
if $C[max' + i - 1]/C[max' - 1] > \tau$ **then**
Output max'
Call $RecursiveRmq(m, l, max - 1)$
Call $RecursiveRmq(m, max + 1, r)$
end

number of blocks each of size i. Let B_i be the set of length i blocks, i.e. B_i=$\{[A[1]...A[i]], [A[i + 1]...A[2i]], \ldots [A[n - i + 1]...A[n]]\}$ and let B=$\{B_{\log n}, \ldots, B_n\}$. For a suffix starting at $A[j]$ and for B_i, we only consider the length i prefix of that suffix, i.e. $A[j \ldots j + i]$. The idea is to store only the maximum probability value per block. For $B_i, i = \log n, \ldots, n$, we define a probability array PB_i containing n/i elements. $PB_i[j]$ is the maximum probability of occurrence of all the substrings of length i belonging to the j-th block of B_i. We build a range maximum query structure RMQ_i for PB_i. RMQ_i takes $O(n/i)$ bits, total space is bounded by $\sum_i O(n/i) = O(n \log n)$ bits or $O(n)$ words.

For a query (p, τ), we first retrieve the suffix range $[l, r]$. This suffix range can spread over multiple blocks of B_m. We use RMQ_m to proceed to next step. Note that RMQ_m consists of N/m bits, corresponding to the N/m blocks of B_m in order. Our query proceeds by executing range maximum query in $RMQ_m(l, r)$, which will give us the index of the maximum probability element of string length m in that suffix range. Let the maximum probability element position in RMQ_m is max and the block containing this element is B_{max}. Using C array, we can find out if the probability of occurrence is greater than τ. Note that, we only stored one maximum element from each block. If the maximum probability found is greater than τ, we check all the other elements in that block in $O(m)$ time. In the next step, we recursively query $RMQ_m(l, max - 1)$ and $RMQ(max + 1, r)$ to find out subsequent blocks. Whenever RMQ query for a range returns an element having probability less than τ, we stop the recursion in that range. Number of blocks visited during query answering is equal to the number of outputs and inside each of those block we check m elements, obtaining total query time of $O(m \times occ)$.

In practical applications, query substrings are rarely longer than $\log n$ length. Our index achieves optimal query time for substrings of length less than $\log n$. We show in the experimental section that on average our index achieves efficient query time proportional to substring length and number of outputs reported.

9.5 Substring Matching in General Uncertain String

In this section we construct index for general uncertain string based on the index of special uncertain string. The idea is to convert a general uncertain string into a special uncertain string, build the data structure similar to the previous section and carefully eliminate the duplicate answers. Below we show the steps of our solution in details.

9.5.1 Transforming General Uncertain String

We employ the idea of Amihood et al [7] to transform general uncertain string into a special uncertain string. **Maximal factor** of an uncertain string is defined as follows.

Problem 6. A **Maximal factor** of a uncertain string S starting at location i with respect to a fixed probability threshold τ_c is a string of maximal length that when aligned to location i has probability of occurrence at least τ_c.

For example in figure 9.3, maximal factors of the uncertain string S at location 5 with respect to probability threshold 0.15 are "QPA", "QPF", "TPA", "TPF".

An uncertain string S can be transformed to a special uncertain string by concatenating all the maximal factors of S in order. Suffix tree built over the concatenated maximal factors can answer substring searching query for a fixed probability threshold τ_c. But this method produces a special uncertain string of $\Omega(n^2)$ length, which is practically infeasible. To reduce the special uncertain string length, Amihood et al. [7] employs further transformation to obtain a set of extended maximal factors. Total length of the extended maximal factors is bounded by $O((\frac{1}{\tau_c})^2 n)$.

Lemma 44. Given a fixed probability threshold value $\tau_c (0 < \tau_c \leq 1)$, an uncertain string S can be transformed into a special uncertain string X of length $O((\frac{1}{\tau_c})^2 n)$ such that any deterministic substring p of S having probability of occurrence greater than τ_c is also a substring of X.

Simple suffix tree structure for answering query does not work for the concatenated extended maximal factors. A special form of suffix tree, namely property suffix tree is introduced by Amihood et al. [7]. Also

substring searching in this method works only on a fixed probability threshold τ_c. A naive way to support arbitrary probability threshold is to construct special uncertain string and property suffix tree index for all possible value of τ_c, which is practically infeasible due to space usage.

We use the technique of lemma 44 to transform a given general uncertain string to a special uncertain string of length $O((\frac{1}{\tau_{min}})^2 n)$ based on a probability threshold τ_{min} known at construction time, and employ a different indexing scheme over it. Let X be the transformed special uncertain string. See Figure 9.6 for an example of the transformation. Following section elaborates the subsequent steps of the index construction.

9.5.2 Index Construction on the Transformed Uncertain String

Our index construction is similar to the index of section 9.4. We need some additional components to eliminate duplication and position transformation.

Let $N = |X|$ be the length of the special uncertain string X. Note that $N = O((\frac{1}{\tau_{min}})^2 n) = O(n)$, since τ_{min} is a constant known in construction time. For transforming the positions of X into the original position in S, we store an array Pos of size N, where $Pos[i]$=position of the i-th character of X in the original string S. We construct the deterministic string $t = c_1 \dots c_N$ where c_i is the character in X_i. We build a suffix tree over t. We build a suffix array A which maps each leaf of the suffix tree to its position in t. We also build a successive multiplicative probability array C, where $C[j] = \prod_{i=1}^{j} Pr(c_i^i)$, for $1 \leq j \leq N$. For a substring of length j starting at position i, probability of occurrence of the substring in X can be easily computed by $C[i+j-1]/C[i-1]$. For $i = 1, \dots, n$, we define C_i as the successive multiplicative probability array for substring length i i.e. $C_i[j] = \prod_{k=A[j]}^{A[j]+i-1} Pr(c_k^k) = C[A[j]+i-1]/C[A[j]-1]$ $(1 \leq j \leq n)$. Figure 9.6 shows Pos array and C array after transformation of an uncertain string. Below we explain how duplicates may arise in outputs and how to eliminate them.

Possible duplicate positions in the output arises because of the general to special uncertain string transformation. Note that, distinct positions in X can correspond to the same position in the original uncertain string S, resulting in same position possibly reported multiple times. A key observation here is that for two different substrings of length m, if the locus nodes are different than the corresponding suffix ranges are disjoint. These disjoint suffix ranges collectively cover all the leaves of the suffix tree. For each such disjoint ranges, we need to store probability values for only the unique positions of S. Without loss of generality we store the value for leftmost unique position in each range.

Algorithm 3: Algorithm General-Short-Substring-Index-Construction

input : A general uncertain string S, probability threshold τ_{min}
output : Suffix tree over t, suffix array A, Position transformation array Pos, successive multiplicative probability array C, $RMQ_i, i = 1, \ldots, \log n$

Transform S into special uncertain string X for τ_{min} using lemma 44
Build position transformation array Pos
Build deterministic string t from X
Build suffix tree over t
Build suffix array A over t
// Building successive multiplicative probability array
$C[1] = Pr(c_1^1)$
for $i = 2; i \leq n; i++$ **do**
  $C[i] = C[i-1] \times Pr(c_i^i)$
end
// Building $C_i, i = 1, \ldots, \log n$ arrays
for $i = 1; i \leq \log n; i++$ **do**
  for $j = 1; j \leq n; j++$ **do**
    $C_i[j] = C[A[j] + i - 1]/C[A[j] - 1]$
  end
end
// Duplicate elimination in C_i
for $i = 1; i \leq \log n; i++$ **do**
  Find the set of locus nodes L_i in the suffix tree Compute the set of suffix ranges corresponding to L_i
  Use Pos array for duplicate elimination in C_i for each range
end
for $i = 1; i \leq \log n; i++$ **do**
  Build RMQ_i over the array C_i
end

Algorithm 4: Algorithm General-Short-Substring-Query-Answering

input : Query substring p, probability threshold $\tau \geq \tau_{min}$
output : Occurrence positions of p in X with probability of occurrence greater than τ

$m = length(p)$
call RecursiveRmq($m, 1, n$)
  function RECURSIVERMQ(i, l, r) ▷ Recursive RMQ method
    $max = RMQ_m(l, r)$
    $max' = A[max]$
    if $C[max' + i]/C[max'] > \tau$ **then**
      Output $Pos[max']$
      Call $RecursiveRmq(m, l, max - 1)$
      Call $RecursiveRmq(m, max + 1, r)$
    end

For any node u in the suffix tree, $depth(u)$ is the length of the concatenated edge labels from root to u. We define by L_i as the set of nodes u_i^j such that $depth(u_i^j) \geq i$ and $depth(parent(u_i^j)) \leq i$. For $L_i = u_i^1, \ldots, u_i^k$, we have a set of disjoint suffix ranges $[sp_i^1, ep_i^1], \ldots, [sp_i^k, ep_i^k]$. A suffix range $[sp_i^j, ep_i^j]$ can contain duplicate positions of S. Using the Pos array we can find the unique positions for each range and store only the values corresponding to the unique positions in C_i.

We use range maximum query data structure RMQ_i of n bits over C_i and discard the original array C_i. Note that, RMQ data structure can be built over an integer array. We convert C_i into an integer array by multiplying each element by a sufficiently large number and then build the RMQ_i structure over it. We obtain $\log n$ number of such RMQ data structures resulting in total space of $O(n \log n)$ bits or $O(n)$ words. For long substrings ($m > \log n$), we use the blocking data structure similar to section 9.4. Algorithm 3 illustrates the index construction phase for short substrings.

9.5.3 Query Answering

Query answering procedure is almost similar to the query answering procedure of section 9.4. Only difference being the transformation of position which is done using the Pos array. Algorithm 4 illustrates the query answering phase for short query substrings. See Figure 9.6 for a query answering example.

9.5.4 Space Complexity

For analyzing the space complexity, we consider each component of our index. Length of the special uncertain string X and deterministic string t are $O(n)$, where n is the number of positions in S. Suffix tree and suffix tree each takes linear space. We store a successive probability array of size $O(n)$. We build probability array C_i for $i = 1, \ldots, \log n$, where each C_i takes of $O(n)$. However we build RMQ_i of n bits over C_i and discard the original array C_i. We obtain $\log n$ number of such RMQ data structures resulting in total space of $O(n \log n)$ bits or $O(n)$ words. For the blocking scheme, we build RMQ_i data structure for $i = \log n, \ldots, n$. RMQ_i takes n/i bits, total space is $\sum_i n/i = O(n \log n)$ bits or $O(n)$ words. Since each component of our index takes linear space, total space taken by our index is $O(n)$ words.

9.5.5 Proof of Correctness

In this section we discuss the correctness of our indexing scheme. **Substring conservation property of the transformation :** At first we show that any substring of S with probability of occurrence greater than query threshold τ can be found in t as well. According to lemma 44, a substring having probability of

S[1]	S[2]	S[3]	S[4]
Q .7	Q .3	P 1	A .4
S .3	P .7		F .3
			P .2
			Q .1

(a) General uncertain string S

t:	Q	Q	P	$	Q	P	P	A	$	Q	P	P	F	$	Q	P	A	$	Q	P	F	$	T	P	A	$	T	P	F	$	P	A	$	P	F	$	P	P	$	A	$	F	$	P	$
Pos:	1	2	3	$	1	2	3	4	$	1	2	3	4	$	2	3	4	$	2	3	4	$	2	3	4	$	2	3	4	$	3	4	$	3	4	$	3	4	$	4	$	4	$	4	$
C:	.7	.21	.21	-1	.7	.49	.49	.19	-1	.7	.49	.49	.14	-1	.5	.5	.2	-1	.5	.5	.15	-1	.5	.5	.2	-1	.5	.5	.15	-1	1	.4	-1	1	.3	-1	1	.2	-1	.4	-1	.3	-1	.2	-1

(b) Deterministic string t, position transformation array Pos, successive multiplicative probability array C.

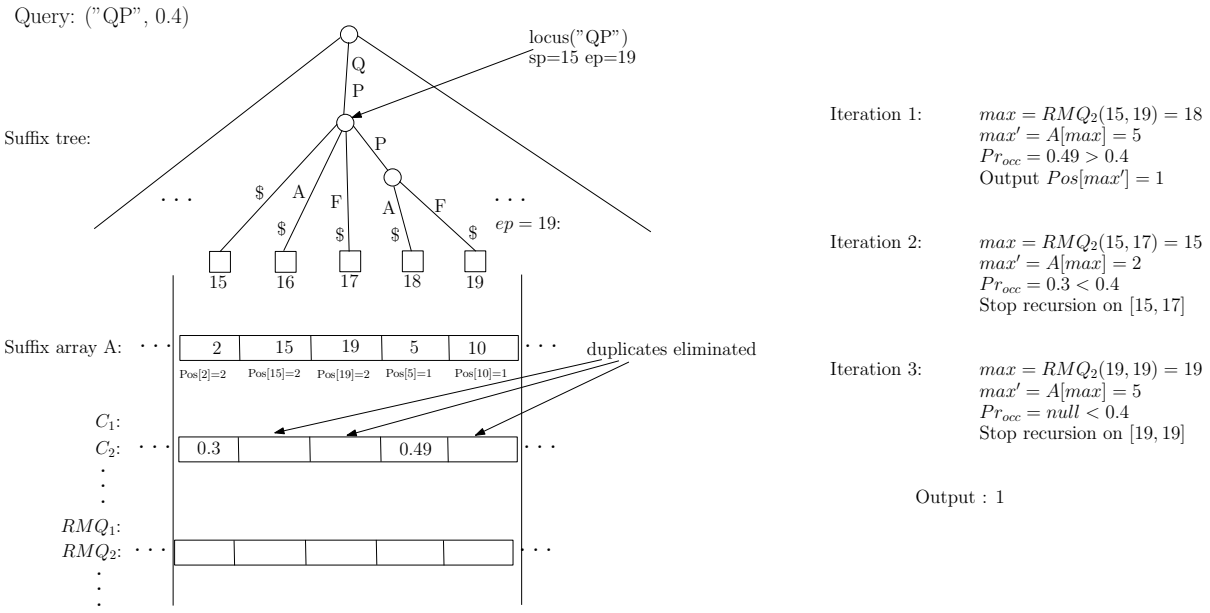

FIGURE 9.6. Running example of Algorithm 4

occurrence greater than τ_{min} in S is also a substring of the transformed special uncertain string X. Since query threshold value τ is greater than τ_{min}, and entire character string of X is same as the deterministic string t, a substring having probability of occurrence greater than query threshold τ in S will be present in the deterministic string t.

Algorithm 4 outputs the complete set of occurrences : For contradiction, we assume that an occurrence position z of substring p in S having probability of occurrence greater than τ is not included in the output. From the aforementioned property, p is a substring of t. According to the property of suffix tree, z must be present in the suffix range $[sp, ep]$ of p. Using RMQ structure, we report all the occurrence in $[sp, ep]$ in

their decreasing order of probability of occurrence value in S and stop when the probability of occurrence falls below τ, which ensures inclusion of z.

Algorithm 4 does not output any incorrect occurrence : An output z can be incorrect occurrence if it is not present in uncertain string S or its probability of occurrence is less than τ. We query only the occurrences in the suffix range $[sp, ep]$ of p, according to the property of suffix tree all of which are valid occurrences. We also validate the probability of occurrence for each of them using the successive multiplicative probability array C.

9.6 String Listing from Uncertain String Collection

In this section we propose an indexing solution for problem 3. We are given a collection of D uncertain strings $\mathcal{D} = \{d_1, \ldots, d_D\}$ of n positions in total. Let i denotes the string identifier of string d_i. For a query (p, τ), we have to report all the uncertain string identifiers j such that d_j contains p with probability of occurrence more than τ. In other words, we want to list the strings from a collection of a string, that are relevant to a deterministic query string based on probability parameter.

Relevance metric : For a deterministic string t and an uncertain string S, we define a relevance metric, $Rel(S, t)$. If t does not have any occurrence in S, then $Rel(S, t)$=0. If s has only one occurrence of t, then $Rel(S, t)$ is the probability of occurrence of t in S. If s contains multiple occurrences of t, then $Rel(S, t)$ is a function of the probability of occurrences of t in S. Depending on the application, various functions can be chosen as the appropriate relevance metric. A common relevance metric is the maximum probability of occurrence, which we denote by $Rel(S, t)_{max}$. The OR value of the probability of occurrences is another useful relevance metric. More formally, if a deterministic string t has nonzero probable occurrences at positions $i_1, \ldots, i_k$ of an uncertain string S, then we define the relevance metric of t in S as $Rel(S, t)_{OR}$ = $\sum_{j=i_1}^{i_k} pr(t_j) - \prod_{j=i_1}^{i_k} pr(t_j)$, where $pr(t_j)$ is the probability of occurrence of t in S at position j. Figure 9.7 shows an example.

Practical motivation : Uncertain string listing finds numerous practical motivation. Consider searching for a virus pattern in a collection of text files with fuzzy information. The objective is to quarantine the files that contain the virus pattern. This problem can be modeled as a uncertain string listing problem, where the collection of text files is the uncertain string collection D, the virus pattern is the query pattern P, and τ is

Uncertain string S:

$S[1]$	$S[2]$	$S[3]$	$S[4]$	$S[5]$	$S[6]$
A .4	B .3	A .5	A .6	B .5	A .4
B .3	L .3	F .5	B .4	F .3	C .3
F .3	F .3			J .2	E .2
	J .1				F .1

$Rel(S, "BFA")_{max}$=.09
$Rel(S, "BFA")_{OR}$=$(.06 + .09 + .048) - (.06 * .09 * .048)$=.19786

FIGURE 9.7. Relevance metric for string listing.

the confidence of matching. Similarly, searching for a gene pattern in genomic sequences of different species can be solved using uncertain string listing data structure.

The index : As explained before, a naive search on each of the string will result in $O(\sum_i$search time on $d_i)$ which can be much larger than the actual number of strings containing the string. Objective of our index is to spend only one search time and time proportional to the number of output strings. We construct a generalized suffix tree so that we have to search for the string only once. We concatenate $d_1, \ldots, d_D$ by a special symbol which is not contained in any of the document and obtain a concatenated general uncertain string $S = d_1\$ \ldots \$d_D$. Next we use the transformation method described in previous section to obtain deterministic string t, construct suffix tree and suffix array for t. According to the property of suffix tree, the leaves under the locus of a query substring t contains all the occurrence positions of t. However, these leaves can possibly contain duplicate positions and multiple occurrence of the same document. In the query answering phase, duplicate outputs can arise because of the following two reasons:

1. Distinct positions in t can correspond to the same position in the original uncertain string S
2. Distinct positions in S can correspond to the same string identifier d_j which should be reported only once

Duplicate elimination is important to keep the query time proportional to the number of output strings. At first we construct the successive multiplicative probability array C_i similar to the substring searching index, then show how to incorporate $Rel(S, t)$ value for the multiple occurrences cases in the same document and duplicate elimination.

Let y_j^i, for $j = 1, \ldots, n$ denote a deterministic substring which is the i-length prefix of the j-th suffix,i.e. the substring on the root to i-th leaf path. Note that, multiple y_j^i can belong to the same locus node in the

suffix tree. Let Y^i is the set of y^i_j, for $j = 1, \ldots, n$. The i-depth locus nodes in the suffix tree constitutes disjoint partitions in Y^i. For $i = 1, \ldots, n$, we define C_i as the successive multiplicative probability array for the substrings of Y^i. j-th element of C_i is the successive multiplicative probability of the i-length prefix of the j-th suffix. More formally $C_i[j] = \prod_{k=A[j]}^{A[j]+i-1} Pr(c^k_k) = C[A[j] + i - 1]/C[A[j] - 1](1 \leq j \leq n)$.

The i-depth locus nodes in the suffix tree constitutes disjoint partitions in C_i. Let u be a i-depth locus node having suffix range $[j \ldots k]$ and root to u substring t. Then the partition $C_i[j \ldots k]$ belongs to u. For this partitions, we store only one occurrence of a string d_j with the relevance metric value $Rel(S, t)$, and discard the other occurrences of d_j in that range. We build RMQ structure similar to section 9.5.

Query Answering We explain the query answering for short substrings. Blocking scheme described in previous section can be used for longer query substrings. Given an input (p, τ), we first retrieve the suffix range $[l, r]$ in $O(m)$ time using suffix tree, where m is the length of p. We can find the maximum relevant occurrence of p in $O(1)$ time by executing query $RMQ_m(l, r)$. Let max be the position of maximum relevant occurrence and $max' = A[max]$ be the the original position in t. For relevance metric $Rel(S, t)_{max}$, we can find the corresponding probability of occurrence by $C[max' + i - 1]/C[max' - 1]$. In case of the other complex relevance metric, all the occurrences need to be considered to retrieve the actual value of $Rel(S, t)$. If the relevance metric is less that τ, we conclude our search. If it is greater than τ, we report max' as an output. For finding rest of the outputs, we recursively search in the ranges $[l, max - 1]$ and $[max + 1, r]$. Each call to $RMQ_m(l, r)$ takes constant time. For simpler relevance metrics (such as $Rel(S, t)_{max}$), validating the relevance metric takes constant time. Total query time is optimal $O(m + occ)$. However, for more complex relevance metric, all the occurrences of t might need to be considered, query time will be proportionate to the total number of occurrences.

9.7 Approximate Substring Searching

In this section we introduce an index for approximate substring matching in an uncertain string. As discussed previously, several challenges of uncertain string matching makes it harder to achieve optimal theoretical bound with linear space. We have proposed index for exact matching which performs near optimally in practical scenarios, but achieves theoretical optimal bound only for shorter query strings. To achieve optimal theoretical bounds for any query, we propose an approximate string matching solution. Our

approximate string matching data structure answers queries with an additive error ϵ, i.e. outputs can have probability of occurrence $\geq \tau - \epsilon$.

At first we begin by transforming the uncertain string S into a special uncertain string X of length $N = O((\frac{1}{\tau_{min}})^2 n)$ using the technique of lemma 44 with respect to a probability threshold value τ_{min}. We obtain a deterministic string t from X by concatenating the characters of X. We build a suffix tree for t. Note that, each leaf in the suffix tree has an associated probability of occurrence $\geq \tau_{min}$ for the corresponding suffix. Given a query p, substring matching query for threshold τ_{min} can now be answered by simply scanning the leafs in subtree of locus node i_p. We first describe the framework (based on Hon et. al. [68]) which supports a specific probability threshold τ and then extend it for arbitrary $\tau \geq \tau_{min}$.

We begin by marking nodes in the suffix tree with positional information by associating $Pos_{id} \in [1, n]$. Here, Pos_{id} indicates the starting position in the original string S. A leaf node l is marked with a $Pos_{id} = d$ if the suffix represented by l begins at position d in S. An internal node u is marked with d if it is the lowest common ancestor of two leaves marked with d. Notice that a node can be marked with multiple position ids. For each node u and each of its marked position id d, define a link to be a triplet $(origin, target, Pos_{id})$, where $origin = u$, target is the lowest proper ancestor of u marked with d, and $Pos_{id} = d$. Two crucial properties of these links are listed below.

- Given a substring p, for each position d in S where p matches with probability $\geq \tau_{min}$, there is a unique link whose origin is in the subtree of i_p and whose target is a proper ancestor of i_p, i_p being the locus node of substring p.
- The total number of links is bounded by $O(N)$.

Thus, substring matching query with probability threshold τ_{min} can now be answered by identifying/reporting the links that originate in the subtree of i_p and are targeted towards some ancestor of it. By referring to each node using its pre-order rank, we are interested in links that are stabbed by locus node i_p. Such queries can be answered in $O(m + occ)$, where $|p| = m$ and occ is the number of answers to be reported (Please refer to [68] for more details).

As a first step towards answering queries for arbitrary $\tau \geq \tau_{min}$, we associate probability information along with each link. Thus each link is now a quadruple $(origin, target, Pos_{id}, prob)$ where first three parameters remain same as described earlier and $prob$ is the probability of $prefix(u)$ matching uncertain string S at

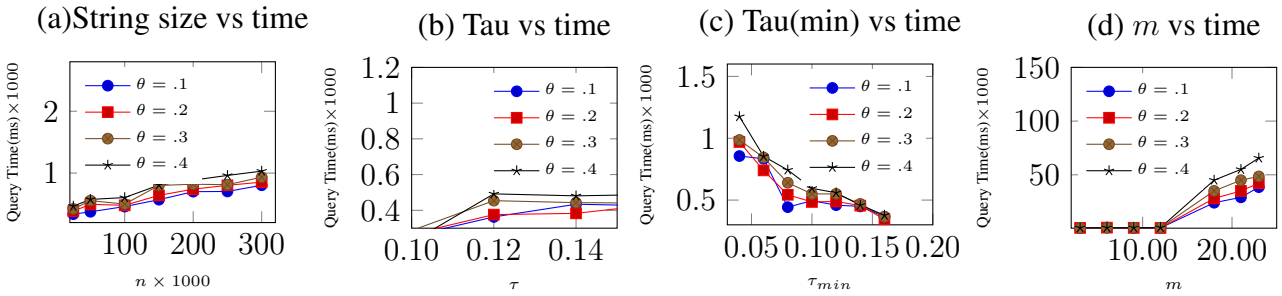

FIGURE 9.8. Substring searching query Time for different string lengths(n), query threshold value τ, construction time threshold parameter τ_{min} and query substring length m.

position $Pos_{id} = d$. It is evident that for substring p and arbitrary $\tau \geq \tau_{min}$, a link stabbed by locus node i_p with $prob \geq \tau$ corresponds to an occurrence of p in S at position d with probability $\geq \tau$. However, a link stabbed by i_p with $prob < \tau$ can still produce an outcome since $prefix(i_P)$ contains additional characters not included in p, which may be responsible for matching probability to drop below τ. Even though we are interested only in approximate matching this observation leads up the next step towards the solution. We partition each link $(origin = u, target = v, Pos_{id} = d, prob)$ into multiple links $(or_1 = u, tr_1, d, prob_1)$, $(or_2 = tr_1, tr_2, d, prob_2)$, ..., $(or_k = tr_{k-1}, tr_k = v, d, prob_k)$ such that $prob_j - prob_{j-1} \leq \epsilon$ for $2 \leq j \leq k$. Here $or_2, \ldots, or_k$ may not refer to the actual node in the suffix tree, rather it can be considered as a dummy node inserted in-between an edge in suffix tree. In essence, we move along the path from node $u = or_1$ towards its ancestors one character at a time till the probability difference is bounded by ϵ i.e., till we reach node tr_1. The process then repeats with tr_1 as the origin node and so on till we reach the node v. It can be see that the total number of links can now be bounded by $O(N/\epsilon)$. In order to answer a substring matching query with threshold $\tau \geq \tau_{min}$, we need to retrieve all the links stabbed by i_p with $prob \geq \tau$. Occurrence of substring p in S corresponding to each such link is then guaranteed to have its matching probability at-least $\tau - \epsilon$ due to the way links are generated (for any link with (u, v) as origin and target probability of $prefix(v)$ matching in S can be more than that of $prefix(v)$ only by ϵ at the most).

9.8 Experimental Evaluation

In this section we evaluate the performance of our substring searching and string listing index. We use a collection of query substrings and observe the effect of varying the key parameters. Our experiments show that, for short query substrings, uncertain string length does not affect the query performance. For long query substrings, our index fails to achieve optimal query time. However this does not deteriorate the average query

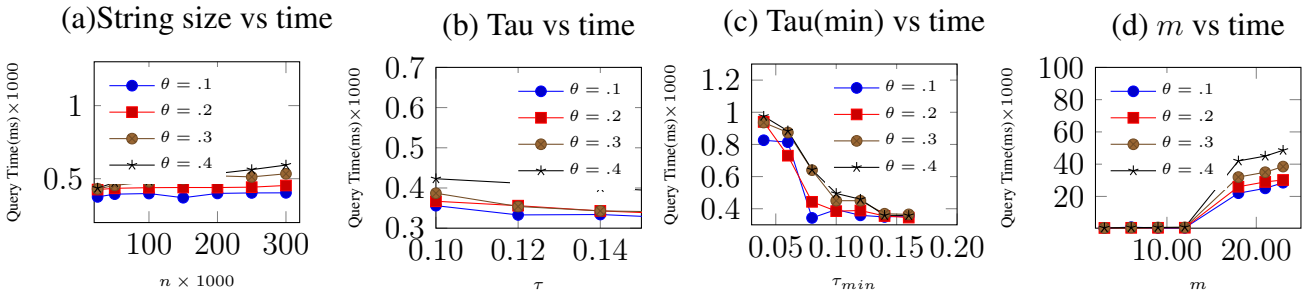

FIGURE 9.9. String listing query Time for different string lengths(n), query threshold value τ, construction time threshold parameter τ_{min} and query substring length m.

time by big margin, since the probability of match also decreases significantly as substring gets longer. Index construction time is proportional to uncertain string size and probability threshold parameter τ_{min}.

We have implemented the proposed indexing scheme in C++. The experiments are performed on a 64-bit machine with an Intel Core i5 CPU 3.33GHz processor and 8GB RAM running Ubuntu. We present experiments along with analysis of performance.

9.8.1 Dataset

We use a synthetic datasets obtained from their real counterparts. We use a concatenated protein sequence of mouse and human (alphabet size $|\Sigma| = 22$), and break it arbitrarily into shorter strings. For each string s in the dataset we first obtain a set $A(s)$ of strings that are within edit distance 4 to s. Then a character-level probabilistic string S for string s is generated such that, for a position i, the pdf of $S[i]$ is based on the normalized frequencies of the letters in the i-th position of all the strings in $A(s)$. We denote by θ the fraction of uncertain characters in the string. θ is varied between 0.1 to 0.5 to generate strings with different degree of uncertainty. The string length distributions in this dataset follow approximately a normal distribution in the range of $[20, 45]$. The average number of choices that each probabilistic character $S[i]$ may have is set to 5.

9.8.2 Query Time for Different String Lengths(n) and Fraction of Uncertainty(θ)

We evaluate the query time for different string lengths n, ranging from $2K$ to $300K$ and θ ranging from 0.1 to 0.5. Figure 9.8(a) and Figure 9.9(a), shows the query times for substring searching and string listing. Note that, n is number of positions in the uncertain string where each position can have multiple characters. We take the average time for query lengths of 10,100,500,1000. We use $\tau_{min} = 0.1$ and query threshold $\tau = 0.2$. As shown in the figures, query times does not show much irregularity in performance when the length of string goes high. This is because for shorter query length, our index achieves optimal query time. Although

for longer queries, our index achieves $O(m \times occ)$ time, longer query strings probability of occurrence gets low as string grows longer resulting in less number of outputs. However when fraction of uncertainty(θ) increases in the string, performance shows slight decrease as query time increases slightly. This is because longer query strings are more probable to match with strings with high level of uncertainty.

9.8.3 Query Time for Different τ and Fraction of Uncertainty(θ)

In Figure 9.8(b) and Figure 9.9(b), we show the average query times for string matching and string listing for probability threshold $\tau = 0.04, 0.06, 0.08, 0.1, 0.12$ for fixed $\tau_{min} = 0.1$. In terms of performance, query time increases with decreasing τ. This is because more matching is probable for smaller τ. Larger τ reduces the output size, effectively reducing the query time as well.

9.8.4 Query Time for Different τ_{min} and Fraction of Uncertainty(θ)

In Figure 9.8(c) and Figure 9.9(c), we show the average query times for string matching and string listing for probability threshold $\tau_{min} = 0.04, 0.06, 0.08, 0.1, 0.12$ which shows slight impact of τ_{min} over query time.

9.8.5 Query Time for Different Substring Length m and Fraction of Uncertainty(θ)

In figure 9.8(d) and figure Figure 9.9(d), we show the average query times for string matching and string listing. As it can be seen long pattern length drastically increases the query time.

9.8.6 Construction Time for Different String Lengths and θ

Figure 9.10(a) shows the index construction times for uncertain string length n ranging from $2K$ to $300K$. We can see that the construction time is proportional to the string length n. Increasing uncertainty factor θ also impacts the construction time as more permutation is possible with increasing uncertain positions. Figure 9.10(b) shows the impact of θ on construction time.

9.8.7 Space Usage

Theoretical bound for our index is $O(n)$. However, this bound can have hidden multiplicative constant. Here we elaborate more on the actual space used for our index.

For our indexes, we construct the regular string t of length $N = O((\frac{1}{\tau_{min}})^2 n)$ by concatenating all the extended maximal factors based on threshold τ_{min}. We do not store the string t in our index. We built RMQ structures RMQ_i for $i = 1, \ldots, \log n$ which takes $O(N \log n)$ bits. The practical space usage of

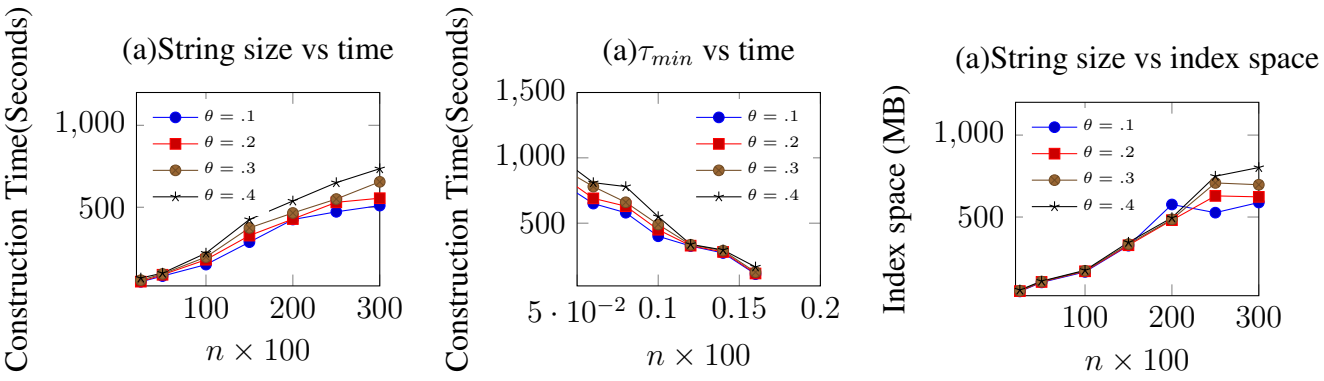

FIGURE 9.10. Construction time and index space for different string lengths(n) and probability threshold $\tau_{min} = .1$

RMQ is usually very small with hidden multiplicative constant of $2 - 3$. So the average space usage of our RMQ structure in total can be stated as $3N$ words. For a query string p, we find the suffix range of p in the concatenated extended maximum factor string t. For this purpose, instead of using Generalized Suffix Tree(GST), we use its space efficient version i.e., a compressed suffix array (CSA) of t. There are many versions of CSA's available in literature. For our purpose we use the one in [10] that occupies $N \log \sigma + o(N \log \sigma) + O(N)$ bits space and retrieves the suffix range of query string p in $O(p)$ time. In practice, this structure takes about $2.5N$ words space. We also store an array D of size N storing the partial probabilities, which takes approximately $4N$ bytes of space. Finally Pos array is used for position transformation, taking N words space. Summing up all the space usage, our index takes approximately $3N + 2.5N + 4N + N = 10.5N = (\frac{1}{\tau_{min}})^2 10.5n$. Figure Figure 9.10(c) shows the space usage for different string length(n) and θ.

Chapter 10
Conclusion

Growth of the internet, digital libraries, large genomic projects demands efficient indexing solution supporting various query parameters. In practical applications, the need for efficient algorithms and data structures with different constraints come to the picture. In this thesis, we tackled some of these variants, which extends our capability to satisfy wider application needs. Also existing research mainly deals with regular or deterministic strings, whereas indexing uncertain string remains largely unexplored. We have formulated two classic problems for the uncertain string setting and proposed efficient indexing solution for those. There are still many interesting variations and open problems remaining. We conclude with some of them as listed below:

- We propose index for orthogonal range reporting with shared constraints, breaking the currently known $O(N \log^{\epsilon} N)$ space barrier for four-sided queries in Word-RAM model. In Word-RAM model, we obtained linear space and optimal time index for answering SCRR queries. Our optimal I/O index in external memory takes $O(N \log \log N)$ words of space and answer queries optimally. We also present a linear space index for external memory. We leave it as an open problem to achieve optimal space bounds, avoiding the $O(\log \log N)$ blowup in external memory model. Also it will be interesting to see whether such results can be obtained in Cache Oblivious model.

- We propose linear space index for multi-pattern document retrieval and forbidden pattern document retrieval, in which we report the most relevant k documents, where the relevance is determined by a monotonic function. An interesting future direction would be to construct index where the relevance metric is not limited to monotonic functions. Similarly, for the document retrieval with forbidden extension problem, constructing index supporting more relevance metric remains undiscovered.

- We revisited the maximal generic word and minimal discriminating word problem and proposed a first succinct index for both the problems. However, our solutions takes additional $O(\log \log n)$ from the optimal time. It would be interesting to see if this non-optimal factor can be removed.

- We presented indexing framework for searching in uncertain strings. Our indexes can support arbitrary values of probability threshold parameter. Uncertain string searching is still largely unexplored area. Constructing more efficient index, variations of the string searching problem satisfying diverse query constraints are some interesting future work direction.

Bibliography

[1] Peyman Afshani. On dominance reporting in 3d. In *ESA*, pages 41–51, 2008.

[2] Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting in three and higher dimensions. In *FOCS*, pages 149–158, 2009.

[3] Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting: query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In *Symposium on Computational Geometry*, pages 240–246, 2010.

[4] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *FOCS*, pages 198–207, 2000.

[5] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 476–482, 2001.

[6] Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. *Theor. Comput. Sci.*, 395(2-3):298–310, 2008.

[7] Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. *Theor. Comput. Sci.*, 395(2-3):298–310, 2008.

[8] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. In *SIGMOD Conference*, pages 347–358, 2004.

[9] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. In *PODS*, pages 346–357, 1999.

[10] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. In *ESA*, pages 748–759, 2011.

[11] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23, 2014.

[12] Jon Louis Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

[13] Thomas Bernecker, Hans-Peter Kriegel, Matthias Renz, Florian Verhein, and Andreas Züfle. Probabilistic frequent pattern growth for itemset mining in uncertain databases. In *Scientific and Statistical Database Management - 24th International Conference, SSDBM 2012, Chania, Crete, Greece, June 25-27, 2012. Proceedings*, pages 38–55, 2012.

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

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

[16] Sudip Biswas, Arnab Ganguly, Rahul Shah, and Sharma V Thankachan. Ranked document retrieval with forbidden pattern. In *Combinatorial Pattern Matching*, pages 77–88. Springer, 2015.

[17] Sudip Biswas, Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Ranked document retrieval with forbidden pattern. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, pages 77–88, 2015.

[18] Sudip Biswas, Tsung-Han Ku, Rahul Shah, and Sharma V Thankachan. Position-restricted substring searching over small alphabets. In *String Processing and Information Retrieval*, pages 29–36. Springer, 2013.

[19] Sudip Biswas, Manish Patil, Rahul Shah, and Sharma V Thankachan. Succinct indexes for reporting discriminating and generic words. In *String Processing and Information Retrieval*, pages 89–100. Springer, 2014.

[20] Sudip Biswas, Manish Patil, Rahul Shah, and Sharma V. Thankachan. Succinct indexes for reporting discriminating and generic words. In *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, pages 89–100, 2014.

[21] Sudip Biswas, Manish Patil, Rahul Shah, and Sharma V Thankachan. Shared-constraint range reporting. In *18th International Conference on Database Theory (ICDT 2015)*, volume 31, pages 277–290. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[22] Gerth Stølting Brodal, Rolf Fagerberg, Mark Greve, and Alejandro López-Ortiz. Online sorted range reporting. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, pages 173–182, 2009.

[23] Gerth Stølting Brodal and Kasper Green Larsen. Optimal planar orthogonal skyline counting queries. *CoRR*, abs/1304.7959, 2013.

[24] Timothy M. Chan. Persistent predecessor search and orthogonal point location on the word ram. In *SODA*, pages 1131–1145, 2011.

[25] Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal range searching on the ram, revisited. In *Symposium on Computational Geometry*, pages 1–10, 2011.

[26] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 5, 2006.

[27] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[28] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

[29] Bernard Chazelle. Lower bounds for orthogonal range searching i. the reporting case. *J. ACM*, 37(2):200–212, 1990.

[30] Bernard Chazelle. Lower bounds for orthogonal range searching ii. the arithmetic model. *J. ACM*, 37(3):439–463, 1990.

[31] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[32] Reynold Cheng, Yuni Xia, Sunil Prabhakar, Rahul Shah, and Jeffrey Scott Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 876–887. VLDB Endowment, 2004.

[33] Chun Kit Chui and Ben Kao. A decremental approach for mining frequent itemsets from uncertain data. In *Advances in Knowledge Discovery and Data Mining, 12th Pacific-Asia Conference, PAKDD 2008, Osaka, Japan, May 20-23, 2008 Proceedings*, pages 64–75, 2008.

[34] Chun Kit Chui, Ben Kao, and Edward Hung. Mining frequent itemsets from uncertain data. In *Advances in Knowledge Discovery and Data Mining, 11th Pacific-Asia Conference, PAKDD 2007, Nanjing, China, May 22-25, 2007, Proceedings*, pages 47–58, 2007.

[35] Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. pages 234–242, 2010.

[36] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[37] Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, M. Sohel Rahman, German Tischler, and Tomasz Walen. Improved algorithms for the range next value problem and applications. *Theor. Comput. Sci.*, 434:23–34, 2012.

[38] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.

[39] S Dash, KH Chon, S Lu, and EA Raeder. Automatic real time detection of atrial fibrillation. *Annals of biomedical engineering*, 37(9):1701–1709, 2009.

[40] Stephane Durocher, Rahul Shah, Matthew Skala, and Sharma V. Thankachan. Linear-space data structures for range frequency queries on arrays and trees. In *Mathematical Foundations of Computer Science 2013 - 38th International Symposium, MFCS 2013, Klosterneuburg, Austria, August 26-30, 2013. Proceedings*, pages 325–336, 2013.

[41] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[42] Ahmed Fadiel, Stuart Lithwick, Gopi Ganji, and Stephen W Scherer. Remarkable sequence signatures in archaeal genomes. *Archaea*, 1(3):185–190, 2003.

[43] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[44] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398, 2000.

[45] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[46] J. Fischer and V. Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *ESCAPE*, pages 459–470, 2007.

[47] J. Fischer, V. Heun, and H. M. Stühler. Practical Entropy-Bounded Schemes for $O(1)$-Range Minimum Queries. In *IEEE DCC*, pages 272–281, 2008.

[48] Johannes Fischer, Travis Gagie, Tsvi Kopelowitz, Moshe Lewenstein, Veli Mäkinen, Leena Salmela, and Niko Välimäki. Forbidden patterns. In *LATIN 2012: Theoretical Informatics - 10th Latin American Symposium, Arequipa, Peru, April 16-20, 2012. Proceedings*, pages 327–337, 2012.

[49] Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to LCA and LCE. In *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, pages 36–48, 2006.

[50] Johannes Fischer and Volker Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, First International Symposium, ESCAPE 2007, Hangzhou, China, April 7-9, 2007, Revised Selected Papers*, pages 459–470, 2007.

[51] Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in X+Y and matrices with sorted columns. *J. Comput. Syst. Sci.*, 24(2):197–208, 1982.

[52] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[53] Travis Gagie and Pawel Gawrychowski. Linear-space substring range counting over polylogarithmic alphabets. *CoRR*, abs/1202.3208, 2012.

[54] Pawel Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 455–466. 2014.

[55] Tingjian Ge and Zheng Li. Approximate substring matching over uncertain strings. *PVLDB*, 4(11):772–782, 2011.

[56] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 368–373, 2006.

[57] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 491–500, 2001.

[58] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 397–406, 2000.

[59] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 397–406, 2000.

[60] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

[61] Wing-Kai Hon, Manish Patil, Rahul Shah, and Sharma V. Thankachan. Compressed property suffix trees. *Inf. Comput.*, 232:10–18, 2013.

[62] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. String retrieval for multi-pattern queries. In *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, pages 55–66. 2010.

[63] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Document listing for queries with excluded pattern. In *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*, pages 185–195. 2012.

[64] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. On position restricted substring searching in succinct space. *J. Discrete Algorithms*, 17:109–114, 2012.

[65] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-*k* string retrieval. *J. ACM*, 61(2):9, 2014.

[66] Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Space-efficient framework for top-k string retrieval problems. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '09, pages 713–722, Washington, DC, USA, 2009. IEEE Computer Society.

[67] Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Space-efficient framework for top-k string retrieval problems. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pages 713–722, 2009.

[68] Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Space-efficient framework for top-k string retrieval problems. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 713–722. IEEE, 2009.

[69] Wing-Kai Hon, Sharma V. Thankachan, Rahul Shah, and Jeffrey Scott Vitter. Faster compressed top-k document retrieval. In *2013 Data Compression Conference, DCC 2013, Snowbird, UT, USA, March 20-22, 2013*, pages 341–350, 2013.

[70] Jeffrey Jestes, Feifei Li, Zhepeng Yan, and Ke Yi. Probabilistic string similarity joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 327–338, 2010.

[71] M. T. Juan, J. J. Liu, and Y. L. Wang. Errata for "faster index for property matching". *Inf. Process. Lett.*, 109(18):1027–1029, 2009.

[72] Bhargav Kanagal and Amol Deshpande. Indexing correlated probabilistic databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 455–468. ACM, 2009.

[73] Marek Karpinski and Yakov Nekrich. Top-k color queries for document retrieval. In *SODA*, pages 401–411, 2011.

[74] Casper Kejlberg-Rasmussen, Yufei Tao, Konstantinos Tsakalidis, Kostas Tsichlas, and Jeonghun Yoon. I/o-efficient planar range skyline and attrition priority queues. In *PODS*, pages 103–114, 2013.

[75] Tsvi Kopelowitz. The property suffix tree with dynamic properties. In *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, pages 63–75, 2010.

[76] Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Persistency in suffix trees with applications to string interval problems. In *String Processing and Information Retrieval, 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings*, pages 67–80, 2011.

[77] Gregory Kucherov, Yakov Nekrich, and Tatiana A. Starikovskaya. Computing discriminating and generic words. *SPIRE*, pages 307–317, 2012.

[78] Kasper Green Larsen, J. Ian Munro, Jesper Sindahl Nielsen, and Sharma V. Thankachan. On hardness of several string indexing problems. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*, pages 242–251. 2014.

[79] Kasper Green Larsen and Rasmus Pagh. I/o-efficient data structures for colored range and prefix reporting. In *SODA*, pages 583–592, 2012.

[80] Carson Kai-Sang Leung and Boyu Hao. Mining of frequent itemsets from streams of uncertain data. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1663–1670, 2009.

[81] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 257–266, 2008.

[82] Jian Li, Barna Saha, and Amol Deshpande. A unified approach to ranking in probabilistic databases. *The VLDB JournalThe International Journal on Very Large Data Bases*, 20(2):249–275, 2011.

[83] Yuxuan Li, James Bailey, Lars Kulik, and Jian Pei. Efficient matching of substrings in uncertain sequences. In *Proceedings of the 2014 SIAM International Conference on Data Mining, Philadelphia, Pennsylvania, USA, April 24-26, 2014*, pages 767–775, 2014.

[84] David MJ Lilley, Robert M Clegg, Stephan Diekmann, Nadrian C Seeman, Eberhard Von Kitzing, and Paul J Hagerman. Nomenclature committee of the international union of biochemistry and molecular biology (nc- iubmb) a nomenclature of junctions and branchpoints in nucleic acids recommendations 1994. *European Journal of Biochemistry, s. FEBS J*, 230(1):1–2, 1996.

[85] Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In *LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings*, pages 703–714, 2006.

[86] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

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

[88] Yossi Matias, S. Muthukrishnan, Süleyman Cenk Sahinalp, and Jacob Ziv. Augmenting suffix trees, with applications. In *Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy, August 24-26, 1998, Proceedings*, pages 67–78. 1998.

[89] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[90] J. Ian Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, pages 37–42, 1996.

[91] J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top-k term-proximity in succinct space. In *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, pages 169–180, 2014.

[92] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 657–666, 2002.

[93] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 657–666, 2002.

[94] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 657–666, 2002.

[95] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[96] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[97] Gonzalo Navarro and Yakov Nekrich. Top-*k* document retrieval in optimal time and linear space. In *SODA*, pages 1066–1077, 2012.

[98] Gonzalo Navarro and Yakov Nekrich. Top-k document retrieval in optimal time and linear space. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1066–1077. SIAM, 2012.

[99] Gonzalo Navarro and Sharma V. Thankachan. Faster top-k document retrieval in optimal space. In *String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, pages 255–262, 2013.

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

[101] Gonzalo Navarro and Sharma V. Thankachan. Bottom-k document retrieval. *Journal of Discrete Algorithms*, 32(0):69 – 74, 2015. StringMasters 2012; 2013 Special Issue (Volume 2).

[102] Yakov Nekrich. External memory range reporting on a grid. In *ISAAC*, pages 525–535, 2007.

[103] Rasmus Pagh. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[104] Manish Patil and Rahul Shah. Similarity joins for uncertain strings. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1471–1482. ACM, 2014.

[105] Manish Patil, Sharma V Thankachan, Rahul Shah, Yakov Nekrich, and Jeffrey Scott Vitter. Categorical range maxima queries. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 266–277. ACM, 2014.

[106] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. In *ACM-SIAM SODA*, pages 233–242, 2002.

[107] Christopher Re, Nilesh Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 886–895. IEEE, 2007.

[108] K. Sadakane and G. Navarro. Fully-Functional Succinct Trees. pages 134–149, 2010.

[109] Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.

[110] Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 134–149, 2010.

[111] Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *SODA*, pages 134–149, 2010.

[112] Rahul Shah, Cheng Sheng, Sharma V. Thankachan, and Jeffrey Scott Vitter. Top-k document retrieval in external memory. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 803–814, 2013.

[113] Sarvjeet Singh, Chris Mayfield, Sunil Prabhakar, Rahul Shah, and Susanne Hambrusch. Indexing uncertain categorical data. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 616–625. IEEE, 2007.

[114] Yufei Tao, Reynold Cheng, Xiaokui Xiao, Wang Kay Ngai, Ben Kao, and Sunil Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *Proceedings of the 31st international conference on Very large data bases*, pages 922–933. VLDB Endowment, 2005.

[115] Darren Erik Vengroff and Jeffrey Scott Vitter. Efficient 3-d range searching in external memory. In *STOC*, pages 192–201, 1996.

[116] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973.

Vita

Sudip Biswas was born in Chittagong, Bangladesh, in 1987. He obtained his bachelor of Science degree in Computer Science and Engineering in 2009 from Bangladesh University of Engineering and Technology (BUET), Dhaka. During his doctoral studies at Louisiana State University, he has co-authored 8 conference papers and 2 journal publications (published or accepted for publication by November 2015). His research interest falls in the area of algorithms and data structures, computational geometry.