MASTER'S THESIS

String Indexing for Patterns with Wildcards

Hjalte Wedel Vildhøj and Søren Vind

Technical University of Denmark
August 8, 2011

Abstract

We consider the problem of indexing a string t of length n to report the occurrences of a query pattern p containing m characters and j wildcards. Let occ be the number of occurrences of p in t, and σ the size of the alphabet. We obtain the following results.

- A linear space index with query time $O(m + \sigma^j \log\log n + occ)$. This significantly improves the previously best known linear space index described by Lam et al. [ISAAC 2007], which requires query time $\Theta(jn)$ in the worst case.
- An index with optimal query time $O(m + j + occ)$ using space $O(\sigma^{k^2} n \log^k \log n)$, where k is the maximum number of wildcards allowed in the pattern. This is the first non-trivial bound with this query time.
- A time-space trade-off for the problem which generalizes the index described by Cole et al. [STOC 2004].

The Longest Common Prefix (LCP) data structure introduced by Cole et al. is a key component in our results. We give a detailed explanation and show several new properties of the LCP data structure. Most importantly, we show that not only suffixes, but arbitrary sets of substrings of t, can be queried, and that unrooted LCP queries can be performed in linear space. Our results are obtained by combining the new properties of the LCP data structure with well-known and new techniques. In particular, we introduce a generalization of the heavy-path decomposition, which could be of independent interest. Finally, we extend our results to allow variable length gaps or optional wildcards in the query pattern, improving upon the only previously known index for this problem by Lam et al. [ISAAC 2007].

Supervisors: Philip Bille and Inge Li Gørtz

CONTENTS

1

INTRODUCTION

The *string indexing problem* is to build an index for a string t such that the occurrences of a query pattern p can be reported. The classic suffix tree data structure [38] combined with perfect hashing [17] gives a linear space solution for string indexing with optimal query time, i.e., an $O(n)$ space data structure that supports queries in $O(m + occ)$ time, where occ is the number of occurrences of p in t.

Recently, various extensions of the classic string indexing problem that allow errors or wildcards (also known as gaps or don't cares) have been studied [13, 26, 36, 35, 8, 29, 32]. In this thesis, we focus on one of the most basic of these extensions, namely, *string indexing for patterns with wildcards*. In this problem, only the pattern contains wildcards, and the goal is to report all occurrences of p in t, where a wildcard is allowed to match any character in t.

String indexing for patterns with wildcards finds several natural applications in large scale data processing areas such as information retrieval, bioinformatics, data mining, and internet traffic analysis. For instance in bioinformatics, the PROSITE data base [23, 7] supports searching for protein patterns containing wildcards.

Despite significant interest in the problem and its many variations, most of the basic questions remain unsolved. We introduce three new indexes and obtain several new bounds for string indexing with wildcards in the pattern. If the index can handle patterns containing an unbounded number of wildcards, we call it an *unbounded wildcard index*, otherwise we refer to the index as a *k-bounded wildcard index*, where k is the maximum number of wildcards allowed in p. Let n be the length of the indexed string t, and σ be the size of the alphabet. We define m and $j \leq k$ to be the number of characters and wildcards in p, respectively. We show that,

- There is an unbounded wildcard index with query time $O(m + \sigma^j \log\log n + occ)$ using linear space. This significantly improves the previously best known linear space index by Lam et al. [26], which requires query time $\Theta(jn)$ in the worst case. Compared to the index by Cole et al. [13] having the same query time, we improve the space usage by a factor $\log n$.

- There is a k-bounded wildcard index with optimal query time $O(m + j + occ)$ using space $O(\sigma^{k^2} n \log^k \log n)$. This is the first non-trivial space bound with this query time.
- There is a time-space trade-off for k-bounded wildcard indexes. This trade-off generalizes the index described by Cole et al. [13].

A key component in our solutions is the Longest Common Prefix (LCP) data structure introduced by Cole et al. [13]. We provide a detailed explanation and give new proofs for the data structure. Additionally, we show two new important properties of the LCP data structure that are essential for obtaining our results.

The above wildcard indexes are obtained by combining the LCP data structure with well-known and new techniques. In particular, we introduce the *heavy α-tree decomposition*, which is a generalization of the classic heavy-path decomposition and could be of independent interest.

Finally, we consider the *string indexing for patterns with variable length gaps problem*, which is a generalization of string indexing for patterns with wildcards. We show that our wildcard indexes can be used to solve this problem by modifying the search algorithm.

Thesis Outline Section 1.2 states our results, compares them to previous solutions, and describes the essential techniques for obtaining them. Chapter 2 covers basic definitions. Chapter 3 contains a detailed description of the LCP data structure and accounts for two new important properties. The unbounded wildcard index using linear space is described in Chapter 4. The time-space trade-off for k-bounded wildcard indexes is given in Chapter 5, and Chapter 6 covers the k-bounded wildcard index with optimal query time. In Chapter 7, we describe how our wildcard indexes can be used to solve the string indexing for patterns with variable length gaps problem.

1.1 Previous Work

Exact string matching has been generalized with error bounds in a number of different ways. In particular, allowing matches within a bounded hamming- or edit distance is known as approximate string matching, and has been subject to extensive research [27, 28, 34, 14, 12, 36, 8, 29, 13, 32, 20, 4]. Another generalization was suggested by Fischer and Paterson [16], allowing wildcards in the text or pattern.

Work on the wildcard problem has mostly focused on the non-indexing variant, where the string t is not preprocessed in advance [16, 15, 11, 25, 10, 6]. Some solutions for the indexing problem considers the case where wildcards appear only in the indexed string [35] or in both the string and the pattern [13, 26].

In the following, we summarize the known indexes that support wildcards in the pattern only. We focus on the case where $k > 1$, since for $k = 0$ the problem is classic string indexing. For $k = 1$, Cole et al. [13] describe a selection of specialized solutions. However, these solutions do not generalize to larger k.

Several simple solutions to the problem exist for $k > 1$. Using a suffix tree T for t [38], we can find all occurrences of p in a top-down traversal starting from the root. When we

reach a wildcard character in p in location $\ell \in T$, the search branches out, consuming the first character on all outgoing edges from ℓ. This gives an unbounded wildcard index using $O(n)$ space with query time $O(\sigma^j m + occ)$, where occ is the total number of occurrences of p in t. Alternatively, we can build a compressed trie storing all possible modifications of all suffixes of t containing at most k wildcards. This gives a k-bounded wildcard index using $O(n^{k+1})$ space with query time $O(m + j + occ)$, since there are $\sum_{i=0}^{k} \binom{n}{i} = O(n^k)$ possible modifications for each of the n suffixes.

In 2004, Cole et al. [13] gave an elegant k-bounded wildcard index using $O(n \log^k n)$ space with $O(m + 2^j \log\log n + occ)$ query time. For sufficiently small values of j this significantly improves the previous bounds. The key components in this solution is a new data structure for *longest common prefix (LCP) queries* and a *heavy path decomposition* [22] of the suffix tree for the text t. Given a pattern p, the LCP data structure supports efficiently inserting all suffixes of p into the suffix tree for t, such that subsequent longest common prefix queries between any pair of suffixes from t and p can be answered in $O(\log\log n)$ time. This is the $\log\log n$ term in the query time. The heavy path decomposition partitions the suffix tree into disjoint *heavy paths* such that any root-to-leaf path contains at most a logarithmic number of heavy paths. Cole et al. [13] show how reduce the size of the simple linear time index at the cost of increasing query time. The idea is to only create additional wildcard trees for the off-path subtries in the heavy path decomposition. This leads to the $O(n \log^k n)$ space bound. The construction ensures that the top-down search branches at most twice for each wildcard in the pattern, leading to the 2^j term in the query time. Though they did not consider unbounded wildcard indexes, the technique can be extended to this case by using only the LCP data structure, and not creating wildcard trees. This leads to an unbounded wildcard index with query time $O(m + \sigma^j \log\log n + occ)$ using space $O(n \log n)$.

A different approach was taken by Iliopoulos and Rahman [24], allowing them to obtain an unbounded wildcard index using linear space. For a pattern p consisting of strings $p_0, \ldots, p_j$ (subpatterns) interleaved by j wildcards, their index has query time $O(m + \sum_{i=0}^{j} occ(p_i, t))$, where $occ(p_i, t)$ denotes the number of matches of p_i in t.

This was later improved by Lam et al. [26] with an index that determines complete matches by first identifying potential matches of the subpatterns in the suffix tree for

Type	Time	Space	Solution	
Unbounded	$O(m + \sum_{i=0}^{j} occ(p_i, t))$	$O(n)$	Iliopoulos and Rahman	[24]
	$O(m + j \min_{0 \le i \le j} occ(p_i, t))$	$O(n)$	Lam et al.	[26]
	$O(\sigma^j m + occ)$	$O(n)$	Simple suffix tree index	†
	$O(m + \sigma^j \log\log n + occ)$	$O(n)$	ART decomposition	†
	$O(m + \sigma^j \log\log n + occ)$	$O(n \log n)$	Cole et al.	[13]
k-Bounded	$O(m + \beta^j \log\log n + occ)$	$O(n \log n \log_\beta^{k-1} n)$	Heavy α-tree decomposition	†
	$O(m + 2^j \log\log n + occ)$	$O(n \log^k n)$	Cole et al.	[13]
	$O(m + j + occ)$	$O(n\sigma^{k^2} \log^k \log n)$	Special index for small patterns	†
	$O(m + j + occ)$	$O(n^{k+1})$	Simple optimal time index	†

Table 1.1: † = presented in this thesis. The term $occ(p_i, t)$ denotes the number of matches of p_i in t and is $\Theta(n)$ in the worst case.

t and subsequently verifying each possible match for validity using interval stabbing on the subpatterns. Their solution is an unbounded wildcard index with query time $O\left(m + j \min_{0 \leq i \leq j} occ(p_i, t)\right)$ using linear space. However, both of these solutions have a worst case query time of $\Theta(jn)$, since there may be $\Theta(n)$ matches for a subpattern, but no matches of p. Table 1.1 summarizes the existing solutions for the problem in relation to our results.

1.2 Our Results

Our main contributions are three new wildcard indexes.

> **Theorem 1** *Let t be a string of length n from an alphabet of size σ. There is an unbounded wildcard index for t using $O(n)$ space. The index can report the occurrences of a pattern with m characters and j wildcards in time $O(m + \sigma^j \log\log n + occ)$.*

Compared to the solution by Cole et al. [13], we obtain the same query time while reducing the space usage by a factor $\log n$. We also significantly improve upon the previously best known linear space index by Lam et al. [26], as we match the linear space usage while improving the worst-case query time from $\Theta(jn)$ to $O(m + \sigma^j \log\log n + occ)$ provided $j \leq \log_\sigma n$. Our solution is faster than the simple suffix tree index for $m = \Omega(\log\log n)$. Thus, for sufficiently small j we improve upon the previously known unbounded wildcard indexes.

The main idea of the solution is to combine an ART decomposition [2] of the suffix tree for t with the LCP data structure. The suffix tree is decomposed into a number of logarithmic sized bottom trees and a single top tree. We introduce a new variant of the LCP data structure for use on the bottom trees, which supports queries in logarithmic time and linear space. The logarithmic size of the bottom trees leads to LCP queries in time $O(\log\log n)$. On the top tree we use the LCP data structure by Cole et al. [13] to answer queries in time $O(\log\log n)$. The number of LCP queries performed during a search for p is $O(\sigma^j)$, yielding the $\sigma^j \log\log n$ term in the query time. The reduced size of the top tree causes the index to be linear in size.

> **Theorem 2** *Let t be a string of length n from an alphabet of size σ. For $2 \leq \beta < \sigma$, there is a k-bounded wildcard index for t using $O\left(n \log(n) \log_\beta^{k-1} n\right)$ space. The index can report the occurrences of a pattern with m characters and $j \leq k$ wildcards in time $O\left(m + \beta^j \log\log n + occ\right)$.*

The theorem provides a time-space trade-off for k-bounded wildcard indexes. Compared to the index by Cole et al. [13], we reduce the space usage by a factor $\log^{k-1} \beta$ by increasing the branching factor from 2 to β. For $\beta = 2$ the index is identical to the index by Cole et al. The result is obtained by generalizing the wildcard index described by Cole et al. We use a *heavy α-tree decomposition*, which is a new technique generalizing the classic heavy path decomposition by Harel and Tarjan [22]. This decomposition could be of independent interest. We also show that for $\beta = 1$ the same technique yields an index with optimal query time $O(m + j + occ)$ using space $O(nh^k)$, where h is the height of the suffix tree for t.

Theorem 3 *Let t be a string of length n from an alphabet of size σ. There is a k-bounded wildcard index for t using $O(\sigma^{k^2} n \log^k \log n)$ space. The index can report the occurrences of a pattern with m characters and $j \leq k$ wildcards in time $O(m+j+occ)$.*

To our knowledge this is the first optimal time index with a non-trivial space bound. The result improves upon the space usage of the simple optimal time index when $\sigma^k < n/\log\log n$. To achieve this result, we use the $O(nh^k)$ space index to obtain a black-box reduction that can produce an optimal time index from an existing index.

The idea is to build the $O(nh^k)$ space index with support for short patterns, and query another index if the pattern is long. This technique is similar to the one used by Bille and Gørtz [5] and closely related to the concept of *filtering search* introduced by Chazelle [9]. The theorem follows from applying the black-box reduction to the index of Theorem 1.

2

PRELIMINARIES

In Section 2.1 and Section 2.2 we introduce the notation and definitions used throughout the thesis. Furthermore, we briefly review a few important data structures in Section 2.3. A complete summary of our notation is enclosed in Appendix A.

2.1 Strings and Basic Definitions

Let $p = p_0 * p_1 * \ldots * p_j$ be a pattern consisting of $j+1$ strings $p_0, p_1, \ldots, p_j \in \Sigma^*$ (subpatterns) interleaved by $j \leq k$ wildcards. The substring starting at position $l \in \{1, \ldots, n\}$ in t is an occurrence of p if and only if each subpattern p_i matches the corresponding substring in t. That is,

$$p_i = t\left[l + i + \sum_{r=0}^{i-1} |p_r|, l + i - 1 + \sum_{r=0}^{i} |p_r|\right] \quad \text{for} \quad i = 0, 1, \ldots, j\ ,$$

where $t[i, j]$ denotes the substring of t between indices i and j, both inclusive. We define $t[i, j] = \varepsilon$ for $i > j$, $t[i, j] = t[1, j]$ for $i < 1$ and $t[i, j] = t[i, |t|]$ for $j > |t|$. Furthermore $m = \sum_{r=0}^{j} |p_r|$ is the number of characters in p, and we assume without loss of generality that $m > 0$ and $k > 0$.

Let $\mathrm{pref}_i(t) = t[1, i]$ and $\mathrm{suff}_i(t) = t[i, n]$ denote the prefix and suffix of t of length i and $n - i + 1$, respectively. Omitting the subscripts, we let $\mathrm{pref}(t)$ and $\mathrm{suff}(t)$ denote the set of all non-empty prefixes and suffixes of t, respectively.

For two strings $x, y \in \Sigma^*$, we denote the maximum common prefix between the two strings as $\mathrm{maxpref}(x, y)$, i.e., $\mathrm{maxpref}(x, y)$ is the string z of maximum length such that z is a prefix of both x and y. We also refer to the length of $\mathrm{maxpref}(x, y)$ as the distance that x follows y.

We extend the definitions of pref, suff and $\mathrm{maxpref}$ to sets of strings $S \subset \Sigma^*$ as follows.

$$\mathrm{pref}_i(S) = \{\mathrm{pref}_i(x) \mid x \in S\} \qquad \mathrm{suff}_i(S) = \{\mathrm{suff}_i(x) \mid x \in S\}$$

$$\mathrm{pref}(S) = \bigcup_{x \in S} \mathrm{pref}(x) \qquad \mathrm{suff}(S) = \bigcup_{x \in S} \mathrm{suff}(x)$$

$$\mathrm{maxpref}(S, t) = \mathrm{pref}_i(t) \ \text{ where } \ i = \max_{x \in S} |\mathrm{maxpref}(x, t)|$$

A set of strings $S \subset \Sigma^*$ is *prefix-free* if no string in S is a prefix of another string in S. Any string set S can be made prefix-free by appending the same unique character $\$ \notin \Sigma$ to each string in S.

We use the symbol $\prec$ to denote the strict lexicographic order relation on strings, i.e., for $x, y \in \Sigma^*$, we write $x \prec y$ if and only if x precedes y in lexicographic order. Furthermore we write $x \preceq$ if either $x \prec y$ or $x = y$. The following lemma establishes a useful connection between the lexicographic order and the maximum prefix of strings, and we will use the corollary of the lemma in many of the following proofs.

Lemma 1 *Let $x, y, z \in \Sigma^*$ such that $x \preceq y \preceq z$ or $z \preceq y \preceq x$, then*

$$\operatorname{maxpref}(x, z) = \operatorname{minstr}\{\operatorname{maxpref}(x, y), \operatorname{maxpref}(y, z)\} ,$$

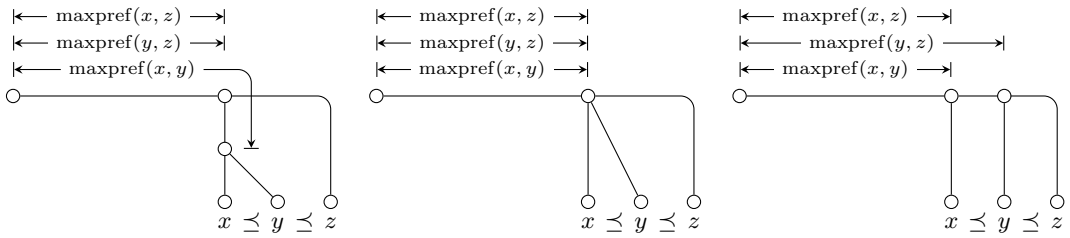
where $\operatorname{minstr} S = \operatorname{argmin}_{x \in S} |x|$ is a string in $S \subset \Sigma^*$ of minimum length.

Proof Without loss of generality, we consider the case where $x \preceq y \preceq z$. We consider the three possible cases for the maximum common prefix of x and y in relation to the maximum common prefix of y and z.

Case 1: $|\operatorname{maxpref}(x, y)| > |\operatorname{maxpref}(y, z)|$. In this case y follows x longer than z as shown in Figure 2.1(a). Thus, since x and y branches from z in the same location, $\operatorname{maxpref}(x, z) = \operatorname{maxpref}(y, z) = \operatorname{minstr}\{\operatorname{maxpref}(x, y), \operatorname{maxpref}(y, z)\}$.

Case 2: $|\operatorname{maxpref}(x, y)| = |\operatorname{maxpref}(y, z)|$. Here y follows x and z equally long as shown in Figure 2.1(b). That is, y branches from the two strings in the same location they branch from each other, so $\operatorname{maxpref}(x, z) = \operatorname{maxpref}(x, y) = \operatorname{maxpref}(y, z) = \operatorname{minstr}\{\operatorname{maxpref}(x, y), \operatorname{maxpref}(y, z)\}$.

Case 3: $|\operatorname{maxpref}(x, y)| < |\operatorname{maxpref}(y, z)|$. This case is symmetric to the first case, so y follows z longer than x. As shown in Figure 2.1(c), both y and z branches from x in the same location, so $\operatorname{maxpref}(x, z) = \operatorname{maxpref}(x, y) = \operatorname{minstr}\{\operatorname{maxpref}(x, y), \operatorname{maxpref}(y, z)\}$.



(a) Case 1 (b) Case 2 (c) Case 3

Figure 2.1: The three possibilities for y.

In all three cases $\operatorname{maxpref}(x, z) = \operatorname{minstr}\{\operatorname{maxpref}(x, y), \operatorname{maxpref}(y, z)\}$. The proof for the case $z \preceq y \preceq x$ is symmetrical. ∎

Corollary 1 *Let $x, y, z \in \Sigma^*$ such that $x \preceq y \preceq z$ or $z \preceq y \preceq x$, then*

$$|\operatorname{maxpref}(x, z)| = \min\big(|\operatorname{maxpref}(x, y)|, |\operatorname{maxpref}(y, z)|\big) .$$

2.2 Trees and Tries

For a tree T, the root is denoted $\operatorname{root}(T)$ and $\operatorname{height}(T)$ is the number of edges on a longest path from $\operatorname{root}(T)$ to a leaf of T. A compressed trie $T(S)$ is a tree storing a prefix-free set of strings $S \subset \Sigma^*$. The edges are labeled with substrings of the strings in S, such that a path from the root to a leaf corresponds to a unique string in S. All internal vertices (except the root) have at least two children, and all labels on the outgoing edges of a vertex have different initial characters. The outgoing edges from a vertex are sorted according to the lexicographical ordering of their labels.

A *location* $\ell \in T(S)$ may refer to either a vertex or a position on an edge in $T(S)$. Formally, $\ell = (v, s)$ where v is a vertex in $T(S)$ and $s \in \Sigma^*$ is a prefix of the label on an outgoing edge of v. If $s = \varepsilon$ we also refer to ℓ as an *explicit vertex*, otherwise ℓ is called an *implicit vertex*. There is a one-to-one mapping between locations in $T(S)$ and unique prefixes in $\operatorname{pref}(S)$. The prefix $x \in \operatorname{pref}(S)$ corresponding to a location $\ell \in T(S)$ is obtained by concatenating the edge labels on the path from $\operatorname{root}(T(S))$ to ℓ. Consequently, we use x and ℓ interchangeably, and we let $|\ell| = |x|$ denote the length of x. Since S is assumed prefix-free, each leaf of $T(S)$ is a string in S, and conversely. The *suffix tree* for t introduced by Weiner [38] denotes the compressed trie over all suffixes of t, i.e., $T(\operatorname{suff}(t))$. We define $T_\ell(S)$ as the subtrie of $T(S)$ rooted at ℓ. That is, $T_\ell(S)$ contains the suffixes of strings in $T(S)$ starting from ℓ. Formally, $T_\ell(S) = T(S_\ell)$, where

$$S_\ell = \Big\{ \operatorname{suff}_{|\ell|}(x) \mid x \in S \wedge \operatorname{pref}_{|\ell|}(x) = \ell \Big\} .$$

2.2.1 Heavy Path Decomposition

For a vertex v in a rooted tree T, we define $\operatorname{weight}(v)$ to be the number of leaves in T_v, where T_v denotes the subtree rooted at v. We define the weight of a tree as $\operatorname{weight}(T) = \operatorname{weight}(\operatorname{root}(T))$. The *heavy path decomposition* of T, introduced by Harel and Tarjan [22], classifies each edge as either *light* or *heavy*. For each vertex $v \in T$, we classify the edge going from v to its child of maximum weight (breaking ties arbitrarily) as heavy. The remaining edges are light. This construction has the property that on a path from the root to any vertex, $O(\log(\operatorname{weight}(T)))$ heavy paths are traversed. For a heavy path decomposition of a compressed trie $T(S)$, we assume that the heavy paths are extended such that the label on each light edge contains exactly one character.

2.3 Overview of Data Structures

Our solutions depend on several advanced data structures. In the following subsections, we briefly review the most important of these.

2.3.1 Predecessor

Let $R \subseteq U = \{0, \ldots, u-1\}$ be a set of integers from a universe U of size u. Given a query element $i \in U$, the *predecessor problem* is to find the maximal element in R which is smaller than i. A *predecessor data structure* stores the set R and supports predecessor and successor queries $\text{PRED}_R(i)$ and $\text{SUCC}_R(i)$. Formally, the queries are defined as follows.

$$\text{PRED}_R(i) = \max_{j \in R,\ j \leq i} j \quad \text{and} \quad \text{SUCC}_R(i) = \min_{j \in R,\ j \geq i} j\ .$$

In a paper from 1982, Willard [39] presents a solution known as a Y-fast trie having query time $O(\log\log u)$ and space usage $O(|R|)$. The data structure splits R in a number of non-overlapping consecutive parts of size $O(\log |R|)$, each stored in a balanced binary search tree. Furthermore, the data structure stores the values splitting the consecutive parts of R in a specialized trie known as a X-fast trie. A query is answered by first searching the X-fast trie for the correct split value and then searching its neighboring consecutive parts of R, each of which can be done in time $O(\log\log u)$.

2.3.2 Nearest Common Ancestor

Given two vertices u, v in a rooted tree T, the *nearest common ancestor problem* asks for the common ancestor of u and v of greatest depth. In 2004, Alstrup et al. [1] presented a labeling scheme solving the problem with query time $O(1)$ and $O(n)$ space usage, where n is the number of vertices in T. Their scheme assigns a label to every vertex $v \in T$ that encodes the path from the root of T to v, using a heavy path decomposition of T to reduce the length of each label to $O(\log n)$. By using alphabetic coding for the labels, the total label length sums to $O(n)$. A nearest common ancestor query $\text{NCA}(u, v)$ calculates the label of the nearest common ancestor vertex of two vertices $u, v \in T$ by considering the labels of u and v and finding their common prefix in constant time.

2.3.3 Weighted Ancestor

Let T be a rooted tree, where each edge $e \in T$ has a positive integer weight, denoted $\text{weight}(e)$. The *depth* of a vertex $v \in T$, denoted $\text{depth}(v)$, is the sum of the edge weights on the path from the root of T to v. The *weighted ancestor problem* is to build a data structure for T with support for *weighted ancestor queries*. A weighted ancestor query $\text{WA}(v, i)$ consists of a vertex $v \in T$ and a positive integer i. The answer to the query is the minimal-depth ancestor[1] of v with depth at least i. For our purposes, T is a compressed trie $T(S)$, and the weight of an edge $e \in T(S)$ is equal to the length of the string label e. We will assume that the query $\text{WA}(v, i)$ on a compressed trie with $i \leq \text{depth}(v)$ gives the ancestor location to v of depth exactly i, i.e., the ancestor is allowed to be an implicit vertex. We can determine this location in constant time after having found the minimal-depth explicit ancestor vertex of v having depth at least i.

Amir et al. [3] presented a data structure supporting weighted ancestor queries in time $O(\log\log n)$ and space $O(n)$ on a tree T with n vertices. Their solution uses a heavy path

[1] We assume a vertex is an ancestor of itself.

decomposition of T and performs a binary search on the $O(\log n)$ heavy paths from v to the root of T. When the search has found the correct heavy path, a predecessor query is performed on the vertices of the heavy path to determine the correct answer.

3

THE LCP DATA STRUCTURE

In this chapter we describe the *Longest Common Prefix (LCP) data structure* in detail and account for our additions that are essential for obtaining Theorem 1. In Section 3.1 we introduce and summarize the properties of the data structure, and it suffices to read this section to understand the remaining chapters of this thesis.

The subsequent sections in this chapter contain a detailed explanation and proofs of the data structure. Section 3.2 describes the important concept of the *longest prefix string*. The construction of the data structure and the necessary preprocessing of each query string is covered in Section 3.3 and Section 3.4, respectively. Section 3.5 shows how to perform rooted LCP queries. Finally, Section 3.6 details the original method as well as our new method for performing unrooted LCP queries.

3.1 Introduction

The *Longest Common Prefix (LCP) data structure*, introduced by Cole et al. [13], provides a way to traverse a compressed trie without tracing the query string one character at a time. In this section we give a brief self-contained description of the data structure.

The LCP data structure stores a collection of compressed tries $T(C_1), T(C_2), \ldots, T(C_q)$ over the string sets $C_1, C_2, \ldots, C_q$. Cole et al. [13] assumed that each set C_i was a subset of the suffixes of the indexed string t, but we show that each C_i can be an arbitrary set of substrings of t. This is important for the possible applications of the data structure and this property is exploited in Theorem 1.

The purpose of the LCP data structure is to support LCP queries:

LCP(x, i, ℓ): Returns the location in $T(C_i)$ where the search for the string $x \in \Sigma^*$ stops when starting in location $\ell \in T(C_i)$.

If ℓ is the root of $T(C_i)$, we refer to the above LCP query as a *rooted LCP query*. Otherwise the query is called an *unrooted LCP query*. For rooted queries, we sometimes omit the ℓ parameter of the LCP query, since by definition $\ell = \mathrm{root}(T(C_i))$. In addition to the compressed tries $T(C_1), \ldots, T(C_q)$, the LCP data structure also stores the suffix tree for t, denoted $T(C)$ where $C = \mathrm{suff}(t)$.

The answer to a rooted LCP query $\text{LCP}(x, i)$ corresponds to the longest common prefix of x and the strings in C_i, i.e., $\text{LCP}(x, i)$ returns the unique location $\text{maxpref}(C_i, x)$ in $T(C_i)$. Answering a rooted LCP query $\text{LCP}(x, i)$ can be done in $O(|x|)$ time by traversing $T(C_i)$ as if inserting x into the trie, and reporting the location in which the search stops. In this way, answering a very large number of LCP queries for the same string x might require $\Theta(|x|)$ time for each query. Cole et al. [13] observed that the similarity of the tries can be exploited to answer subsequent queries for the same string much faster. Essentially the first query requires $O(|x|)$ time, after which all following LCP queries with x can be answered in $O(\log \log n)$ time irrespective of which trie $T(C_i)$ it is performed on. The term $n = |C|$ is the length of the indexed string t. The following lemma is implicit in the paper by Cole et al. [13].

Lemma 2 (Cole et al.) *Provided x has been preprocessed in time $O(|x|)$, the LCP data structure can answer rooted LCP queries on $T(C_i)$ for any suffix of x in time $O(\log \log n)$ using space $O(n + \sum_{i=1}^{q} |C_i|)$.*

The full proof of this lemma is given in Section 3.5.

3.1.1 Unrooted LCP Queries

By default, the LCP data structure can only answer rooted LCP queries. Cole et al. [13] describes how support for unrooted LCP queries on a compressed trie $T(C_i)$ can be added at the cost of increasing the size of the data structure by $O(|C_i| \log |C_i|)$.

Lemma 3 (Cole et al.) *Provided x has been preprocessed in time $O(|x|)$, unrooted LCP queries on $T(C_i)$ for any suffix of x can be performed in time $O(\log \log n)$ by using $O(|C_i| \log |C_i|)$ additional space.*

The main idea in obtaining this lemma, is to create a heavy path decomposition of $T(C_i)$ and add the compressed subtries rooted in the root of every heavy path to the LCP data structure. This causes the additional $O(|C_i| \log |C_i|)$ space. An unrooted LCP query $\text{LCP}(x, i, \ell)$, where ℓ is not the root of a heavy path, can in time $O(1)$ be reduced to a rooted LCP query on the subtrie of a descendant heavy path. We give the proof of this lemma in Section 3.6.2.

We present a new result, showing that support for slower unrooted LCP queries on a compressed trie $T(C_i)$ can be added using linear additional space.

Lemma 4 *Provided x has been preprocessed in time $O(|x|)$, unrooted LCP queries on $T(C_i)$ for any suffix of x can can be performed in time $O(\log |C_i| + \log \log n)$ by using $O(|C_i|)$ additional space.*

To achieve this, we create a heavy path decomposition of $T(C_i)$, and show that an unrooted LCP query $\text{LCP}(x, i, \ell)$ can be answered by following at most $\log |C_i|$ heavy paths from ℓ to the location where the search for x stops, using constant time per heavy path. On the final heavy path, we need a $O(\log \log n)$ time predecessor query to determine the exact location where the search stops. The full proof of this lemma is given in Section 3.6.3.

3.2 The Longest Prefix String

A very central and important concept of the LCP data structure is the notion of the *longest prefix string* (identical to the *highest overlap string* in Cole et al. [13]). Given a non-empty string set $S \subset \Sigma^*$ and a string $x \in \Sigma^*$, the longest prefix string in S for x, denoted $\mathrm{lps}_S(x)$, is defined as follows:

> **Definition 1 (Longest Prefix String)** *Among those strings in S having the longest maximum common prefix with x, $\mathrm{lps}_S(x)$ is a string which is lexicographically closest to x, breaking ties arbitrarily.*

Notice that for any x there is at least one, and at most two such strings in S, assuming S is non-empty. The maximum common prefix between x and $\mathrm{lps}_S(x)$ is equal to $\mathrm{maxpref}(S, x)$, and we will use $h_S(x) = |\mathrm{maxpref}(S, x)|$ as a shorthand to denote the length of this prefix. To find the longest common prefix string for x in a set S it is convenient to consider the compressed, sorted trie for S. Refer to Figure 3.1 for an illustration of some longest common prefix strings.

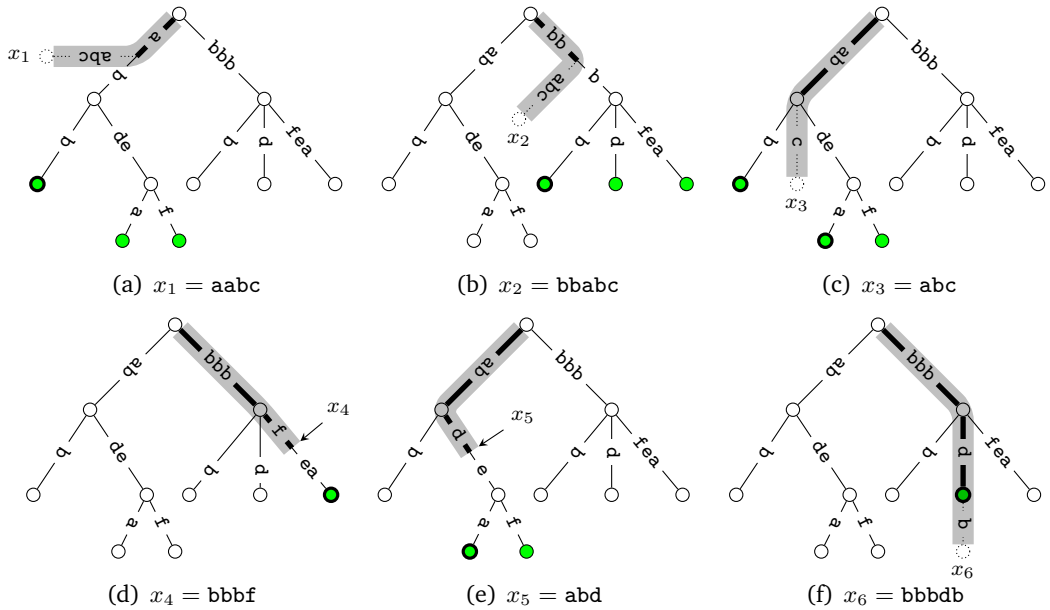


Figure 3.1: Illustrating the longest prefix string for the strings $x_1 =$ aabc, $x_2 =$ bbabc, $x_3 =$ abc, $x_4 =$ bbbf, $x_5 =$ abd and $x_6 =$ bbbdb in the string set $S = \{$abb, abdea, abdef, bbbb, bbbd, bbbfea$\}$. In the above figures of $T(S)$, the strings of S having the longest maximum common prefix with x_i are marked in green. Among these strings, the bold ones are those lexicographically closest to x_i. That is, $\mathrm{lps}_S(x_i)$ is a bold string. Recapitulating, we have that $\mathrm{lps}_S(x_1) =$ abb, $\mathrm{lps}_S(x_2) =$ bbbb, $\mathrm{lps}_S(x_4) =$ bbbfea, $\mathrm{lps}_S(x_5) =$ abdea and $\mathrm{lps}_S(x_6) =$ bbbd. There is a tie for x_3, so $\mathrm{lps}_S(x_3)$ is either abb or abdea. The length of the shared prefixes (marked in bold) are $h_S(x_1) = 1$, $h_S(x_2) = 2$, $h_S(x_3) = 2$, $h_S(x_4) = 4$, $h_S(x_5) = 3$ and $h_S(x_6) = 4$.

The cases in Figure 3.1 suggest that $\mathrm{lps}_S(x)$ is always one of the two strings lexicographically closest to x in S. This is confirmed by the following lemma.

Lemma 5 *The longest prefix string* $\mathrm{lps}_S(x)$ *is lexicographically closest to* x *among all strings in* S*.*

Proof In case $x \in S$, $\mathrm{lps}_S(x) = x$ and the lemma clearly holds. In case $x \notin S$ assume without loss of generality that $\mathrm{lps}_S(x) \prec x$. To obtain a contradiction, suppose that there is a string $y \in S$ different from $\mathrm{lps}_S(x)$, which is lexicographically closer to x than $\mathrm{lps}_S(x)$, i.e.,

$$\mathrm{lps}_S(x) \prec y \prec x \ .$$

Applying Corollary 1 yields that $|\operatorname{maxpref}(y, x)| \geq |\operatorname{maxpref}(\mathrm{lps}_S(x), x)|$. This contradicts the assumption that $\mathrm{lps}_S(x)$ is the longest prefix string for x in S. ∎

3.3 Building the LCP Data Structure

In the construction of the LCP data structure, a suffix tree $T(C)$ is built for the indexed string t, where $C = \mathrm{suff}(t)$. On $T(C)$, a nearest common ancestor labeling scheme is constructed. The construction of $T(C)$ can be done in time $O(n \log \sigma)$ because the suffix tree must be sorted lexicographically [37, 21]. The space usage is $O(n)$, and constructing the nearest common ancestor labels takes $O(n)$ time [1] as this is the number of vertices in $T(C)$.

The data structure also stores the compressed tries $T(C_1), T(C_2), \ldots, T(C_q)$, where each C_i is a set of substrings of t. On these tries, a weighted ancestor data structure is built. Furthermore, let y be a substring of t. We define $\mathrm{order}(y) \in \{1, \ldots, n(n+1)/2\}$ to be the position of y in the lexicographic ordering of all substrings of t. The value of $\mathrm{order}(y)$ for a substring $y \in C_i$ can be found by performing a pre-order traversal of the sorted suffix tree. The order set of C_i is $\mathrm{orderset}(C_i) = \{\mathrm{order}(y) \mid y \in C_i\}$. A predecessor data structure is constructed on $\mathrm{orderset}(C_i)$ for all C_i. Building the compressed tries takes time $O(\log \sigma \sum_{i=1}^{q} |C_i|)$ and space $O(\sum_{i=1}^{q} |C_i|)$. It takes $O(\sum_{i=1}^{q} |C_i|)$ time and space to construct the weighted ancestor and predecessor data structures [3, 39].

Concluding, the LCP data structure takes space $O(n + \sum_{i=1}^{q} |C_i|)$, and can be built in time $O(\log \sigma(n + \sum_{i=1}^{q} |C_i|))$. Note that the data structure as described here only supports rooted LCP queries. If support for unrooted LCP queries is required, further preprocessing may be necessary depending on the type of support to add. Adding support for unrooted LCP queries is described in Section 3.6.

3.4 Preprocessing a Query String

Before performing a query with a string x, it must be preprocessed to obtain two parameters for x that is needed to perform LCP queries in $O(\log \log n)$ time. These two parameters are:

1. The longest prefix string in $C = \mathrm{suff}(t)$ for x, i.e., $\mathrm{lps}_C(x)$.
2. The length of the maximum common prefix $h_C(x) = |\operatorname{maxpref}(C, x)|$.

We preprocess x by searching $T(C)$ for x from the root, matching the characters of x one by one. Eventually the search stops, and one of the following two cases occur.

(a) The search stops in a non-branching vertex (i.e. a implicit vertex or a leaf). In this case the longest prefix string $\mathrm{lps}_C(x)$ will be either the left leaf or the right leaf in the subjacent subtree. Let c_x and c_T be the next unmatched character of x and $T(C)$, respectively. If x is a prefix of some string in C, it is fully matched when the search stops and $c_x = \varepsilon$. If the search stopped in a leaf, $c_T = \varepsilon$. We let v be the next explicit vertex in $T(C)$, descendant of the location where the search stopped. Then the longest prefix string in C for x can be determined as

$$\mathrm{lps}_C(x) = \begin{cases} \mathrm{leftleaf}(v) & \text{if } c_x \leq c_T \\ \mathrm{rightleaf}(v) & \text{if } c_x > c_T \end{cases} .$$

Notice that the only case in which $c_x = c_T$ is when $c_x = c_T = \varepsilon$ and hence $x \in C$. In case $c_T = \varepsilon$, the search stopped in a leaf, so $\mathrm{leftleaf}(v) = \mathrm{rightleaf}(v) = v = \mathrm{lps}_C(x)$.

(b) The search stops in a branching vertex $v \in T(C)$. In this case we need a predecessor query to find the longest prefix string for x, effectively determining the sorting of x in relation to the children of v. As before let c_x be the next unmatched character of x (possibly ε). Assuming a predecessor data structure has been built for v over the first character on the edges to its children, we can choose $\mathrm{lps}_C(x)$ as

$$\mathrm{lps}_C(x) \in \{\mathrm{rightleaf}(\textsc{pred}(c_x)), \mathrm{leftleaf}(\textsc{succ}(c_x))\} .$$

Notice that the set is non-empty, since either $\textsc{pred}(c_x)$ or $\textsc{succ}(c_x)$ must exist.

In both cases, the length of the maximum common prefix $h_C(x) = |\mathrm{maxpref}(C, x)|$ is found as the number of matched characters in x. We use $O(|x|)$ time to search for x and obtain $h_C(x)$. In the first case, it takes constant time to find $\mathrm{lps}_C(x)$. The predecessor query in the second case takes time $O(\log\log\sigma)$, since the alphabet is the universe for the predecessor query. Thus, we have established that preprocessing a string x requires $O(|x| + \log\log\sigma)$ time.

3.4.1 Preprocessing All Suffixes of a Query String

In order to support unrooted LCP queries for x, we will need access to $\mathrm{lps}_C(x')$ and $h_C(x')$ for an arbitrary suffix x' of x. The above method suggests that preprocessing each of the $|x|$ suffixes of x to determine these could take time $\Theta\left(\sum_{i=1}^{|x|} \log\log\sigma + |\mathrm{suff}_i(x)|\right) = \Theta\left(|x|\log\log\sigma + |x|^2\right)$. However, as shown in the following, we can exploit techniques used in linear time construction of *generalized suffix trees* to reduce the preprocessing time.

A generalized suffix tree is a trie $T(\mathrm{suff}(S))$ for a set of strings $S \subset \Sigma^*$. Ukkonen's Algorithm [37] can be used to construct generalized suffix trees on-line, inserting strings from S one at a time. The algorithm does so by extending an already created generalized suffix tree T^i for the string set $S = \{t_0, t_1, \ldots, t_i\}$ with all suffixes of a new string t_{i+1},

obtaining a new suffix tree T^{i+1} that also contains all suffixes of t_{i+1}. For our purposes, the algorithm can be changed to not modify T^i, thus only determining the locations in T^i where all suffixes of t_{i+1} branched from the tree. This can be done in time $O(|t_{i+1}|)$ [21, p 116]. Since T^i is not changed, this effectively searches for all suffixes of t_{i+1} in T^i.

Now, if we consider $T(C)$ as a generalized suffix tree, we can in time $O(|x|)$ determine the location $\ell_{x'} \in T(C)$ where each suffix x' of x branched from the tree by searching for all suffixes of x using Ukkonen's Algorithm. By storing these locations, $h_C(x') = |\operatorname{maxpref}(C, x')| = |\ell_{x'}|$ is available in constant time. When needed, we can determine $\operatorname{lps}_C(x')$ in time $O(\log\log\sigma)$ as described in the previous section, by using $\ell_{x'} \in T(C)$ as the location where the search for x' stopped. We have thus established the following lemma.

> **Lemma 6** *Provided that x has been preprocessed in time $O(|x|)$, $h_C(x')$ is available in constant time, and $\operatorname{lps}_C(x')$ can be determined in time $O(\log\log\sigma)$ for any suffix x' of x.*

This method also supports constant time lookup of $\operatorname{lps}_C(x')$ by preprocessing all suffixes of x in time $O(|x|\log\log\sigma)$.

3.5 Rooted LCP Queries

In this section we show Lemma 2. The idea in answering a rooted LCP query for x on $T(C_i)$ is to find a string z in C_i that x follows longest. We identify the leaf of $T(C_i)$ that corresponds to z and the distance $h = |\operatorname{maxpref}(x, z)|$. We can then use a weighted ancestor query $\text{WA}(z, h)$ to determine the location where x diverges from z. This location is the answer to the rooted LCP query $\text{LCP}(x, i)$. See Figure 3.2 for an illustration.

We will use the longest prefix string for x in C_i, $\operatorname{lps}_{C_i}(x)$, as the string z. The distance h is then equal to $h_{C_i}(x) = |\operatorname{maxpref}(\operatorname{lps}_{C_i}(x), x)|$. In order to determine $\operatorname{lps}_{C_i}(x)$ and $h_{C_i}(x)$, we use $\operatorname{lps}_C(x)$ and $h_C(x)$, which are available thanks to the preprocessing of x.

We find $\operatorname{lps}_{C_i}(x)$ using a number of important lemmas. First, Lemma 7 and Corollary 2 show that we can determine the distance that x follows a string $y \in C_i$ in constant time.

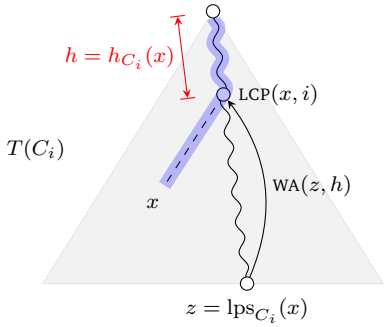


Figure 3.2: Illustrating how a rooted LCP query for x on $T(C_i)$ is answered by a weighted ancestor query on a string $z \in C_i$ that x follows longest.

Lemma 8 shows that we can identify two candidate strings in C_i in time $O(\log \log n)$, at least one of which is a valid choice for $\mathrm{lps}_{C_i}(x)$. In Lemma 9, we use Corollary 2 to determine which of the candidate strings that x follows longest, thereby obtaining a valid choice for $\mathrm{lps}_{C_i}(x)$ as well as $h_{C_i}(x)$.

Lemma 7 *Given a suffix $y \in C$ of t, the distance h that a string $x \in \Sigma^*$ follows y can be determined in constant time as*

$$h = |\operatorname{maxpref}(x, y)| = \min(|\operatorname{NCA}(\mathrm{lps}_C(x), y)|, h_C(x))$$

provided that $\mathrm{lps}_C(x)$ and $h_C(x)$ are available.

Proof By Lemma 5, $\mathrm{lps}_C(x)$ is lexicographically closest to x among all strings in C, so either

1. $x \preceq \mathrm{lps}_C(x) \preceq y$ or $y \preceq \mathrm{lps}_C(x) \preceq x$. In this case, Corollary 1 yields

$$h = |\operatorname{maxpref}(x, y)| = \min\big(|\operatorname{maxpref}(\mathrm{lps}_C(x), y)|, |\operatorname{maxpref}(x, \mathrm{lps}_C(x))|\big) .$$

2. $y \preceq x \preceq \mathrm{lps}_C(x)$ or $\mathrm{lps}_C(x) \preceq x \preceq y$. In this case, Corollary 1 yields

$$|\operatorname{maxpref}(y, \mathrm{lps}_C(x))| = \min\big(|\operatorname{maxpref}(y, x)|, |\operatorname{maxpref}(x, \mathrm{lps}_C(x))|\big) .$$

By definition, $|\operatorname{maxpref}(x, \mathrm{lps}_C(x))| \geq |\operatorname{maxpref}(y, x)|$ for any $y \in C$, so

$$\begin{aligned} h = |\operatorname{maxpref}(y, x)| &= |\operatorname{maxpref}(y, \mathrm{lps}_C(x))| \\ &= \min\big(|\operatorname{maxpref}(\mathrm{lps}_C(x), y)|, |\operatorname{maxpref}(x, \mathrm{lps}_C(x))|\big) . \end{aligned}$$

For both of the above cases we have that

$$h = \min\big(|\operatorname{maxpref}(\mathrm{lps}_C(x), y)|, |\operatorname{maxpref}(x, \mathrm{lps}_C(x))|\big) .$$

By definition, $|\operatorname{maxpref}(x, \mathrm{lps}_C(x))| = h_C(x)$, and since $\operatorname{maxpref}(\mathrm{lps}_C(x), y)$ can be determined in constant time by a nearest common ancestor query on the leaves of $T(C)$ corresponding to $\mathrm{lps}_C(x)$ and y, we have that

$$h = \min\big(|\operatorname{NCA}(\mathrm{lps}_C(x), y)|, h_C(x)\big) .$$

This concludes the proof. ∎

We extend the lemma by showing that the distance that x follows a substring of t can also be determined in constant time.

Corollary 2 *Given a suffix $y \in C$ of t, the distance h that a string $x \in \Sigma^*$ follows $\mathrm{pref}_i(y)$, can be determined in constant time as*

$$h = |\operatorname{maxpref}(x, \mathrm{pref}_i(y))| = \min(i, |\operatorname{NCA}(\mathrm{lps}_C(x), y)|, h_C(x))$$

provided that $\mathrm{lps}_C(x)$ and $h_C(x)$ are available.

Proof The distance that x follows $\mathrm{pref}_i(y)$ is at most $|\mathrm{pref}_i(y)| = i$, and since x follows y at least as long as x follows $\mathrm{pref}_i(y)$, the lemma follows from Lemma 7. ■

Using the predecessor data structure for $\mathrm{orderset}(C_i)$, we can in time $O(\log\log n^2) = O(\log\log n)$ determine the predecessor and successor string for $y \in C$ in the lexicographic ordering of C_i. We denote these strings as $\textsc{pred}_{C_i}(y)$ and $\textsc{succ}_{C_i}(y)$ and the following lemma shows that for $y = \mathrm{lps}_C(x)$, at least one of these strings is a valid choice for $\mathrm{lps}_{C_i}(x)$.

Lemma 8 *Either* $\textsc{pred}_{C_i}(\mathrm{lps}_C(x))$ *or* $\textsc{succ}_{C_i}(\mathrm{lps}_C(x))$ *is a valid choice for* $\mathrm{lps}_{C_i}(x)$.

Proof We let $x^- = \textsc{pred}_{C_i}(\mathrm{lps}_C(x))$ and $x^+ = \textsc{succ}_{C_i}(\mathrm{lps}_C(x))$. By definition, x^- and x^+ are the two strings in C_i lexicographically closest to $\mathrm{lps}_C(x)$. We consider the following cases for the lexicographic ordering of x.

1. $x^- \prec x \prec x^+$. In this case x^- and x^+ are also the two strings in C_i lexicographically closest to x, and by Lemma 5 one of them is a valid choice for $\mathrm{lps}_{C_i}(x)$.

2. $x \preceq x^-$. We show that x^- is a valid choice for $\mathrm{lps}_{C_i}(x)$. To obtain a contradiction, assume that x^- is not a valid choice for $\mathrm{lps}_{C_i}(x)$. Then there is a valid choice $z \in C_i$ different from x^- for $\mathrm{lps}_{C_i}(x)$. It must be the case that $z \prec x^-$, since otherwise x^- would be lexicographically closer to x than z, contradicting that z is a valid choice for $\mathrm{lps}_{C_i}(x)$. Since x^- is not a valid choice for $\mathrm{lps}_{C_i}(x)$, it must hold that either

   a) x follows z longer than x^-, or

   b) z is lexicographically closer to x than x^-, i.e., $x \preceq z \prec x^-$.

   Let $z' \in C$ denote a suffix of t having z as a prefix. In the first case, x follows z' longer than x^-, and thus also longer than $\mathrm{lps}_C(x)$ because of the lexicographic ordering. In the second case z' is lexicographically closer to x than $\mathrm{lps}_C(x)$ since $x \preceq z \prec x^- \preceq \mathrm{lps}_C(x)$. Both cases contradict the definition of $\mathrm{lps}_C(x)$. This shows that x^- must be a valid choice for $\mathrm{lps}_{C_i}(x)$.

3. $x \succeq x^+$. In this case x^+ is a valid choice for $\mathrm{lps}_{C_i}(x)$. The argument is symmetrical to the previous. ■

The following lemma is obtained by combining the previously shown lemmas. The lemma provides $\mathrm{lps}_{C_i}(x)$ and $h_{C_i}(x)$ which are needed in order to answer the rooted LCP query with a weighted ancestor query.

Lemma 9 *Let* $C = \mathrm{suff}(t)$ *be the set of all suffixes of* t. *Given* $\mathrm{lps}_C(x)$ *and* $h_C(x)$, *we can determine* $\mathrm{lps}_{C_i}(x)$ *and* $h_{C_i}(x)$ *in time* $O(\log\log n)$, *provided that a nearest common ancestor data structure has been built for the suffix tree* $T(C)$.

Proof We first obtain the strings $x^- = \text{PRED}_{C_i}(\text{lps}_C(x))$ and $x^+ = \text{SUCC}_{C_i}(\text{lps}_C(x))$ in time $O(\log\log n)$. It follows from Lemma 8 that at least one of x^- and x^+ is a valid choice for $\text{lps}_{C_i}(x)$. Let y^- and y^+ denote suffixes in C having x^- and x^+ as a prefix, respectively.

From Lemma 7, the distance that x follows y^- and y^+ is upper bounded by the distance that $\text{lps}_C(x)$ follows the strings. We can tell which of the strings x follows farthest by comparing the length of the maximum common prefix between $\text{lps}_C(x)$ and y^- to that between $\text{lps}_C(x)$ and y^+ as determined by two nearest common ancestor queries. From the lengths of the maximum common prefixes, we select the correct choice for $\text{lps}_{C_i}(x)$ between x^- and x^+. Thus,

$$\text{lps}_{C_i}(x) = \begin{cases} x^- & \text{if } |\,\text{NCA}(\text{lps}_C(x), y^-)| \geq |\,\text{NCA}(\text{lps}_C(x), y^+)| \\ x^+ & \text{if } |\,\text{NCA}(\text{lps}_C(x), y^-)| < |\,\text{NCA}(\text{lps}_C(x), y^+)| \end{cases}.$$

Note that in case y^- and y^+ has an equally long maximum common prefix with $\text{lps}_C(x)$, we select x^-. This is because $x \prec x^- \prec \text{lps}_C(x) \prec x^+$ is a possible lexicographical ordering for the strings. This may happen if $x^- \notin C$ is a prefix of $\text{lps}_C(x)$, since a string is lexicographically ordered before any other string of which it is a prefix. On the contrary, $x^- \prec \text{lps}_C(x) \prec x^+ \prec x$ is not a possible lexicographical ordering because there would be a string in C having x^+ as a prefix, contradicting that $\text{lps}_C(x)$ is lexicographically closest to x in C. Thus, x^- is always at least as close lexicographically to x as x^+.

The maximum distance, $h_{C_i}(x)$, that x follows a string in C_i equals the distance that x follows $\text{lps}_{C_i}(x)$. Thus, $h_{C_i}(x)$ is the maximum distance that x follows either x^- or x^+ since the maximum of these was the correct choice for $\text{lps}_{C_i}(x)$. We determine the distances using Corollary 2 as

$$\begin{aligned} h_{C_i}(x) &= \max\big(\,|\,\text{maxpref}(x, x^-)|\,,\;|\,\text{maxpref}(x, x^+)|\,\big) \\ &= \max\Big(\min\big(|\,\text{NCA}(\text{lps}_C(x), y^-)|, h_C(x), |x^-|\big)\,, \\ &\qquad\qquad \min\big(|\,\text{NCA}(\text{lps}_C(x), y^+)|, h_C(x), |x^+|\big)\,\Big)\,. \end{aligned}$$

Finding x^+ and x^- can be done in time $O(\log\log n)$. The nearest ancestor queries can be answered in constant time. Hence the total time spent is $O(\log\log n)$. ∎

We now describe how to answer a rooted LCP query on $T(C_i)$ in time $O(\log\log n)$ for a suffix x' of a string x, assuming that x has been preprocessed in time $O(|x|)$. To answer a rooted LCP query $\text{LCP}(x', i)$, we first determine $\text{lps}_C(x')$ and $h_C(x')$ in time $O(\log\log\sigma) = O(\log\log n)$ as described in Lemma 6 for use in the following lemmas. Then we determine the leaf $\text{lps}_{C_i}(x')$ in $T(C_i)$ and $h_{C_i}(x')$ in time $O(\log\log n)$ as described by Lemma 9. Knowing both of these parameters, the location where x' diverges from $T(C_i)$ can be found by a weighted ancestor query on $T(C_i)$, determining the ancestor of $\text{lps}_{C_i}(x')$ having a depth (string length) equal to $h_{C_i}(x)$, i.e., $\text{WA}(\text{lps}_{C_i}(x'), h_{C_i}(x))$ on $T(C_i)$. Thus, a rooted LCP query can be answered in time $O(\log\log n)$, concluding the proof of Lemma 2.

3.5.1 Example of a Rooted LCP Query

In this section we illustrate and describe each of the steps necessary to answer a rooted LCP query for a small example. The goal is to answer a rooted LCP query for the string $x = \texttt{cacba}$ on a compressed trie $T(C_i)$. We assume that the LCP data structure is built for the string $t = \texttt{bccbbccd}$, having the 28 unique substrings shown in Figure 3.3(a) with their lexicographic order number. The LCP data structure is built as previously described, producing the sorted suffix tree $T(C)$ for all suffixes $C = \text{suff}(t) = \{\texttt{bbccd}, \texttt{bccbbccd}, \texttt{bccd}, \texttt{cbbccd}, \texttt{ccbbccd}, \texttt{ccd}, \texttt{cd}, \texttt{d}\}$ of t. Furthermore, the suffixes in $T(C)$ are labeled by their lexicographic order number and a nearest common ancestor data structure has been built for $T(C)$.

First, we preprocess the query string x to find the longest prefix string for x among the suffixes of t, $\text{lps}_C(x)$, and the length of the maximum common prefix $h_C(x)$. As shown in Figure 3.3(b), we find that $\text{lps}_C(x) = \texttt{cbbccd}$ which has order number 19. The length of the maximum common prefix is $h_C(x) = 1$.

Next, we consider the compressed trie $T(C_i)$ (see Figure 3.3(c)) storing the substrings $C_i = \text{pref}_3(\text{suff}(t)) = \{\texttt{bbc}, \texttt{bcc}, \texttt{cbb}, \texttt{ccb}, \texttt{ccd}, \texttt{cd}, \texttt{d}\}$. We assume that $T(C_i)$ is stored in the LCP data structure, and hence a weighted ancestor data structure has been built for $T(C_i)$. Furthermore, a predecessor data structure has been prepared for $\text{orderset}(C_i) = \{3, 7, 16, 21, 26, 27, 28\}$, containing the lexicographic order numbers of the strings in C_i.

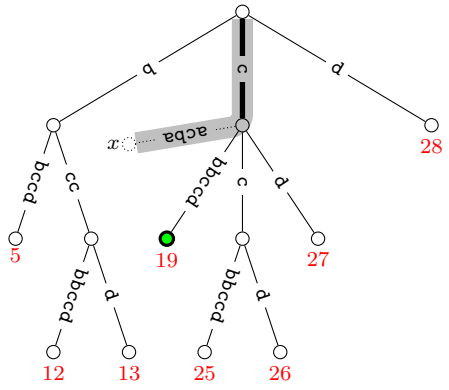
We now describe how a rooted LCP query $\text{LCP}(x, i)$ for the string x on the compressed trie $T(C_i)$ is answered. First, we identify the longest prefix string for x in C_i, $\text{lps}_{C_i}(x)$ and the length $h_{C_i}(x)$ as follows. The predecessor and successor of $\text{lps}_C(x)$ (order number 19) in the orderset of C_i are the strings $x^- = \texttt{cbb}$ (order number 16) and $x^+ = \texttt{ccb}$ (order number 21). By Lemma 8 one of these strings is a valid choice for $\text{lps}_{C_i}(x)$. To determine which, we use Corollary 2 to find the distance that x follows x^- and x^+ respectively. This step consists of performing two nearest common ancestor queries $\text{NCA}(y^-, \text{lps}_C(x))$ and $\text{NCA}(y^+, \text{lps}_C(x))$ on the suffix tree, where y^- and y^+ are suffixes of t having x^- and x^+ as a prefix, respectively. In this way, we find that x follows both x^- and x^+ for a distance of 1, and the longest prefix string in C_i for x is x^-. The answer to the rooted LCP query $\text{LCP}(x, i)$ is the ancestor of x^- of depth 1. This location can be found by a weighted ancestor query $\text{WA}(x^-, 1)$ on $T(C_i)$ as shown in Figure 3.3(d).

3.6 Unrooted LCP Queries

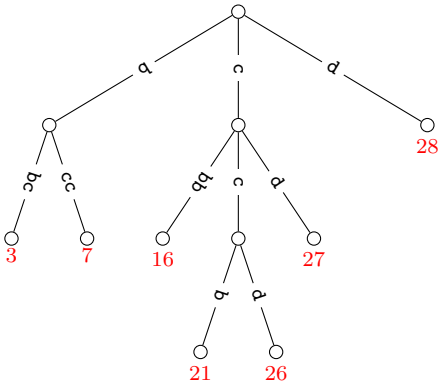
In the following two subsections, we describe two different ways of answering an unrooted LCP query $\text{LCP}(x, i, \ell)$ on a trie $T(C_i)$. The first method is the one stated by Cole et al. [13], which requires $O(|C_i| \log |C_i|)$ additional space to support unrooted queries in time $O(\log \log n)$ on a trie $T(C_i)$. This method results in Lemma 3. The second method is a new solution that requires $O(|C_i|)$ additional space to add support for unrooted LCP queries in time $O(\log |C_i| + \log \log n)$ on $T(C_i)$. This method results in Lemma 4.

1	b	15	cb
2	bb	16	cbb
3	bbc	17	cbbc
4	bbcc	18	cbbcc
5	**bbccd**	19	**cbbccd**
6	bc	20	cc
7	bcc	21	ccb
8	bccb	22	ccbb
9	bccbb	23	ccbbc
10	bccbbc	24	ccbbcc
11	bccbbcc	25	**ccbbccd**
12	**bccbbccd**	26	**ccd**
13	**bccd**	27	**cd**
14	c	28	**d**

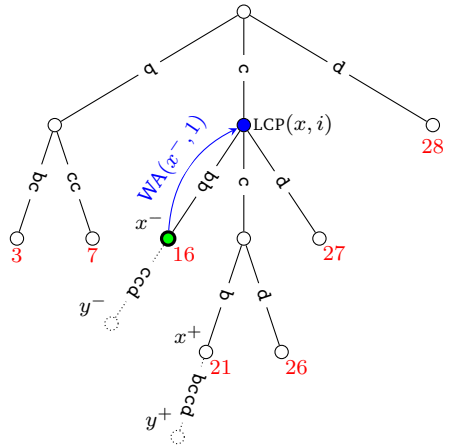
(a) The 28 unique substrings of t and their lexicographic order number. The suffixes of t are marked in bold.



(b) The position of $x = \texttt{cacba}$ in the suffix tree for t, i.e., $T(C)$ with $C = \text{suff}(t)$. The longest prefix string in C for x is $\text{lps}_C(x) = \texttt{cbbccd}$ and is highlighted in green. It is also seen that $h_C(x) = 1$.



(c) A compressed trie $T(C_i)$ containing a set C_i of substrings of t. Here $C_i = \text{pref}_3(\text{suff}(t))$.



(d) Finding $\text{LCP}(x, i)$ by a weighted ancestor query on $T(C_i)$, using that $\text{lps}_{C_i}(x) = x^-$ and $h_{C_i}(x) = 1$ has been determined.

Figure 3.3: Illustrating how to answer a rooted LCP query $\text{LCP}(x, i)$ on a compressed trie $T(C_i)$ stored in the LCP data structure. The indexed text in this example is $t = \texttt{bccbbccd}$ and the query string is $x = \texttt{cacba}$.

3.6.1 Prerequisites

Before describing the details of the two solutions, we first account for the prerequisites they share. We assume that the LCP data structure has been constructed as described in Section 3.3, i.e., in particular a nearest common ancestor data structure has been built for the suffix tree $T(C)$.

Both the method by Cole et al. [13] and our new method rely on a heavy path decomposition $\mathcal{H}$ of $T(C_i)$ to add support for unrooted LCP queries on $T(C_i)$. As described in Section 2.2.1, the top of each heavy path $H \in \mathcal{H}$ is extended until every light edge contains exactly one single character. This implies that the root of a heavy path H, which we denote $\mathrm{root}(H)$, is not necessarily an explicit vertex in $T(C_i)$. To be able to index into a heavy path $H \in \mathcal{H}$, we store an array containing for each explicit vertex $v \in H$, the string length of the string starting in $\mathrm{root}(H)$ and ending in v. By building a predecessor data structure for this array, we can find the location on H at string distance i from $\mathrm{root}(H)$ by a single predecessor query to determine the explicit parent vertex for the location. Storing these predecessor data structures requires $O(|C_i|)$ additional space, since each vertex in $T(C_i)$ is contained in at most one heavy path. Indexing into the array for H can be done in constant time, and a predecessor query on the array can be answered in time $O(\log\log \max_{x\in C_i} |x|) = O(\log\log n)$, since the size of the universe is bounded by the length of the longest string in C_i. Constructing the heavy path decomposition and the predecessor data structures takes time $O(|C_i|)$. We assume this is done when the LCP data structure is built.

For both methods of answering unrooted LCP queries, the following lemma is very central.

Lemma 10 *Given a location $\ell \in T(C_i)$ on a heavy path $H \in \mathcal{H}$ and a string $x \in \Sigma^*$, we let h denote the distance that x follows H starting in ℓ. Provided that x has been preprocessed in time $O(|x|)$, we can determine h in constant time.*

Proof Observe that the leaf of each heavy path $H \in \mathcal{H}$, $\mathrm{leaf}(H)$, corresponds to a substring of t, i.e., the string starting in $\mathrm{root}(T(C_i))$ and ending in $\mathrm{leaf}(H)$. For each heavy path $H \in \mathcal{H}$, we store the position $\mathrm{pos}(y_H)$ of a suffix y_H of t having the string $\mathrm{leaf}(H)$ as a prefix. Since $\mathrm{leaf}(H)$ is a substring of t, there must be at least one suffix $y_H \in C$. From the definition, any location ℓ on H is a prefix of both $\mathrm{leaf}(H)$ and y_H.

Let z_ℓ denote the substring of t starting in ℓ and ending in $\mathrm{leaf}(H)$ of length $|z_\ell| = |\mathrm{leaf}(H)| - |\ell|$. We will use Corollary 2 to determine the distance that x follows z_ℓ, which requires us to identify a suffix z'_ℓ of t having z_ℓ as a prefix.

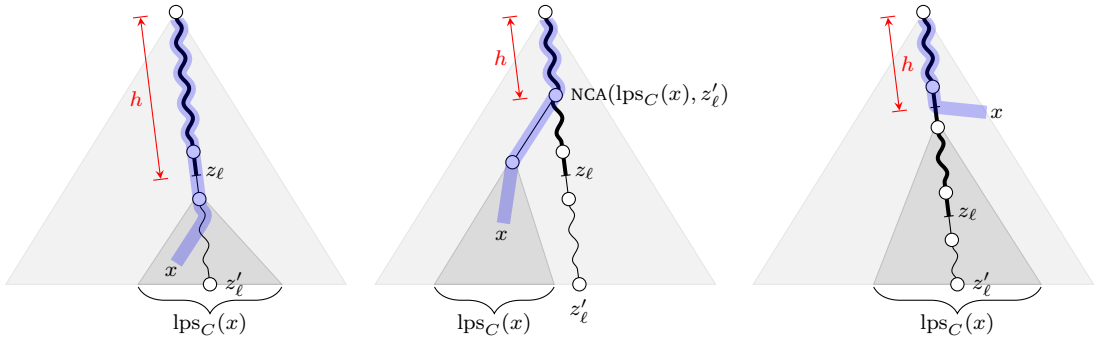
To that end, note that $\mathrm{leaf}(H) = \ell \cdot z_\ell \sqsubseteq_{\mathrm{pref}} y_H$, so z_ℓ is a prefix of the string obtained by removing the prefix ℓ from y_H. Thus, we obtain z'_ℓ as the suffix of t shifted $|\ell|$ positions to the right from y_H, effectively stripping ℓ off y_H. That is, $z'_\ell = \mathrm{suff}_{\mathrm{pos}(y_H)+|\ell|}(t)$, so we have that

$$z_\ell = \mathrm{pref}_{|z_\ell|}(z'_\ell) = \mathrm{pref}_{|\mathrm{leaf}(H)|-|\ell|}\big(\mathrm{suff}_{\mathrm{pos}(y_H)+|\ell|}(t)\big)\ .$$

Thus by Corollary 2 we can determine the distance h that x follows z_ℓ in constant time as

$$\begin{aligned} h &= |\operatorname{maxpref}(x, z_\ell)| \\ &= \min\Big(|z_\ell|\,,\ \big|\textsc{nca}\big(\operatorname{lps}_C(x), z'_\ell\big)\big|\,,\ h_C(x) \Big) \\ &= \min\Big(|\operatorname{leaf}(H)| - |\ell|\,,\ \big|\textsc{nca}\big(\operatorname{lps}_C(x), \operatorname{suff}_{\operatorname{pos}(y_H)+|\ell|}(t)\big)\big|\,,\ h_C(x) \Big)\,. \end{aligned}$$

Figure 3.4 is an illustration of the cases where h equals $|z_\ell|$, $|\textsc{nca}(\operatorname{lps}_C(x), z'_\ell)|$ and $h_C(x)$, respectively. ■



(a) $h = |z_\ell|$. (b) $h = |\textsc{nca}(\operatorname{lps}_C(x), z'_\ell)|$. (c) $h = h_C(x)$.

Figure 3.4: Determining h by considering the possible cases in the suffix tree for t, $T(C)$. If z_ℓ is a prefix of x (see Figure 3.4(a)), we have $h = |\operatorname{maxpref}(x, z_\ell)| = |z_\ell|$. In case z_ℓ is not a prefix of x, there are two cases two consider. Either x diverges from z_ℓ on a path in C (see Figure 3.4(b)). In this case $\operatorname{lps}_C(x)$ is located in the same off-path subtrie that x branches off into. Consequently, the maximum prefix between x and z_ℓ can be found by a nearest common ancestor query between $\operatorname{lps}_C(x)$ and z'_ℓ. Otherwise, x diverges from z_ℓ on a path not in C (or x is fully matched on z_ℓ). See Figure 3.4(c). In this case, the maximum common prefix between x and z_ℓ equals the maximum common prefix between x and any string in C, i.e., $h = h_C(x)$.

3.6.2 The Solution by Cole et al.

The main idea in the method described by Cole et al. [13] for answering an unrooted LCP query $\textsc{lcp}(x, i, \ell)$ is to reduce the query to a rooted LCP query on a subtrie of $T(C_i)$ included in the LCP data structure. A simple way to achieve this would be to include *every* subtrie of $T(C_i)$ in the LCP data structure. The unrooted LCP query $\textsc{lcp}(x, i, \ell)$ could then be answered by performing a rooted LCP query on the subtrie of $T(C_i)$ rooted at location ℓ. Unfortunately, this approach could require $\Theta(|C_i|n)$ space to store the additional subtries in the LCP data structure, since a string $y' \in C_i$ is present in $|y'| \leq n$ subtries.

To reduce this space usage, we will only include selected subtries of $T(C_i)$ in the LCP data structure. More precisely, for each heavy path H in the heavy path decomposition $\mathcal{H}$ of $T(C_i)$, we add the subtrie of $T(C_i)$ rooted at $\operatorname{root}(H)$ to the LCP data structure. The additional space required to store the predecessor and weighted ancestor data structures

for these subtries only amounts to $O(|C_i| \log |C_i|)$, since each of the $|C_i|$ strings is contained in at most $\log |C_i|$ of these subtries. However, this only yields support for unrooted LCP queries starting in a location ℓ, which is also the root of some heavy path $H \in \mathcal{H}$. For all other locations in $T(C_i)$, we will reduce the unrooted LCP query for x to a rooted LCP query for *a suffix of* x starting in the root of a another heavy path. More precisely, we preprocess $T(C_i)$ as follows.

> **Preprocessing of $T(C_i)$:** For all heavy paths $H \in \mathcal{H}$, $T_{\mathrm{root}(H)}(C_i)$ is added to the LCP data structure to support unrooted LCP queries starting in $\mathrm{root}(H)$.

Adding a subtrie $T_{\mathrm{root}(H)}(C_i)$ to the LCP data structure consists of constructing the predecessor data structure for the orderset over the strings starting in $\mathrm{root}(H)$ and ending in a leaf. Additionally, a weighted ancestor data structure is constructed on the vertices of $T_{\mathrm{root}(H)}(C_i)$. The additional preprocessing requires $O(|C_i|)$ additional space for each subtrie of $T(C_i)$ added to the LCP data structure.

We now describe how to answer an unrooted LCP query $\textsc{lcp}(x, i, \ell)$ when ℓ is not a top location of a heavy path in $T(C_i)$. Let H denote the heavy path in $\mathcal{H}$ that ℓ is located on. The search for x follows H some string distance $h \leq |x|$ from ℓ after which the search either stops (x is fully matched) or leaves H. Using Lemma 10 we can compute h in constant time. Knowing h, we can determine the answer to $\textsc{lcp}(x, i, \ell)$ in time $O(\log \log n)$ as follows.

Using h to Find $\textsc{lcp}(x, i, \ell)$ We first obtain the location ℓ_h on H at distance h from ℓ by performing a predecessor query on H to obtain the explicit parent vertex of ℓ_h. The answer can then be obtained as described below.

1. If $h = |x|$, x is fully matched on H and we have that $\textsc{lcp}(x, i, \ell) = \ell_h$.
2. If $h < |x|$, the search for x leaves H at location ℓ_h, and there are two possibilities to consider:
   a) ℓ_h has an outgoing light edge (ℓ_h, ℓ'_h) labeled with the next unmatched character $x[h+1]$. Since ℓ'_h is the root of another heavy path in $\mathcal{H}$, the answer to $\textsc{lcp}(x, i, \ell)$ can be found by a rooted LCP query for x's unmatched suffix $\mathrm{suff}_{h+2}(x)$ on the subtrie $T_{\ell'_h}(C_i)$. See Figure 3.5 for an illustration of this case.
   b) ℓ_h has no outgoing edge labeled $x[h+1]$, so the pattern cannot follow $T(C_i)$ longer. In this case, we have that $\textsc{lcp}(x, i, \ell) = \ell_h$.

In any case, $\textsc{lcp}(x, i, \ell)$ can be determined in $O(\log \log n)$ time. This completes the proof of Lemma 3.

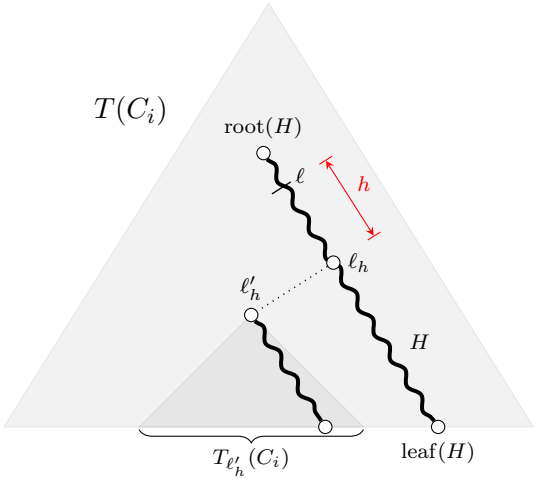


Figure 3.5: Illustrating how to answer an unrooted LCP query from the location $\ell \in T(C_i)$ using the method described by Cole et al. [13]. The illustration shows the situation where the search for x leaves the heavy path H on a light edge and proceeds in the subtrie $T_{\ell'_h}(C_i)$.

3.6.3 A New Solution

When performing a LCP query $\text{LCP}(x, i, \ell)$, the search path for x starting in ℓ traverses a number of heavy paths in $T(C_i)$ as shown in Figure 3.6. The solution by Cole et al. [13] only considers the first of these heavy paths, and uses a rooted LCP query to complete the search. The main idea of our new solution is to follow all the $O(\log |C_i|)$ heavy paths that the search path passes through, thereby avoiding the need for a rooted LCP query and the additional space overhead caused by adding subtries of $T(C_i)$ to the LCP data structure. Intuitively, for each heavy path intersected by the search path, the next heavy path can be identified in constant time using Lemma 10. On the final heavy path, a predecessor query is needed to determine the exact location where the search path stops.

For a heavy path H, we let h denote the distance which the search path for x starting in ℓ follows H. Referring to Figure 3.6, we can compute the answer to an unrooted LCP query recursively as follows. To answer $\text{LCP}(x, i, \ell)$ we identify the heavy path H of $T(C_i)$ that ℓ is part of and compute the distance h in constant time as described by Lemma 10. If x leaves H on a light edge, indexing distance h into H from ℓ yields an explicit vertex v. At v, a constant time lookup for $x[h+1]$ determines the light edge on which x leaves H. Since the light edge has a label of length one, the next location ℓ' on that edge is the root of the next heavy path. We continue the search for the remaining suffix of x from ℓ' recursively by a new unrooted LCP query $\text{LCP}(\text{suff}_{h+2}(x), i, \ell')$. If H is the heavy path on which the search for x stops, the location at distance h (i.e., the answer to the original LCP query) is not necessarily an explicit vertex, and may not be found by indexing into H. In that case a predecessor query for h is performed on H to determine the preceding explicit vertex and thereby the location $\text{LCP}(x, i, \ell)$. Answering an unrooted LCP query entails at most $\log |C_i|$ recursive steps, each taking constant time. The final recursive step may require a predecessor query taking time $O(\log \log n)$. Consequently, an unrooted LCP

query can be answered in time $O(\log |C_i| + \log \log n)$ using $O(|C_i|)$ additional space to store the predecessor data structures for each heavy path. This completes the proof of Lemma 4.

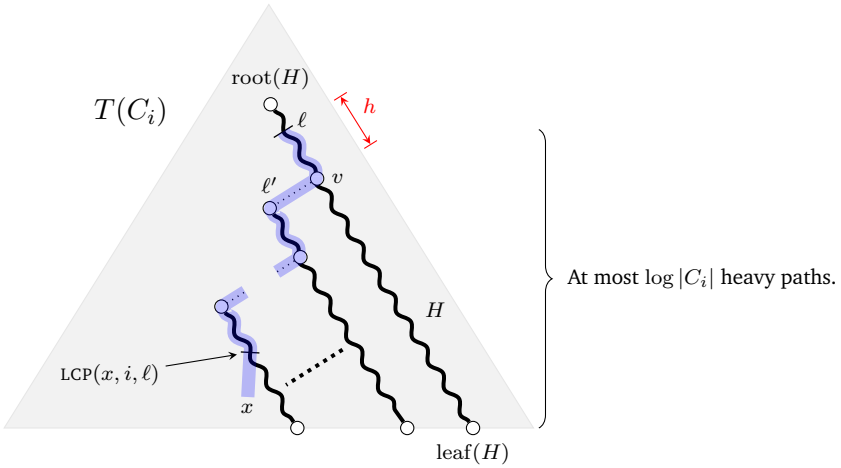


Figure 3.6: Illustrating how an unrooted LCP query LCP(x, i, ℓ) can be answered by recursively following at most $\log |C_i|$ heavy paths in $T(C_i)$.

4

AN UNBOUNDED WILDCARD INDEX USING LINEAR SPACE

In this chapter we show Theorem 1 by applying an ART decomposition on the suffix tree for t and storing the top and bottom trees in the LCP data structure. For completeness, we review the ART decomposition in Section 4.1 before describing the wildcard index in Section 4.2.

4.1 ART Decomposition

The ART decomposition introduced by Alstrup et al. [2] decomposes a tree into a single *top tree* and a number of *bottom trees*. The construction is parameterized by an integer $\chi > 0$ and defined by two rules:

1. A bottom tree is a subtree rooted in a vertex of minimal depth such that the subtree contains no more than χ leaves.
2. Vertices that are not in any bottom tree make up the top tree.

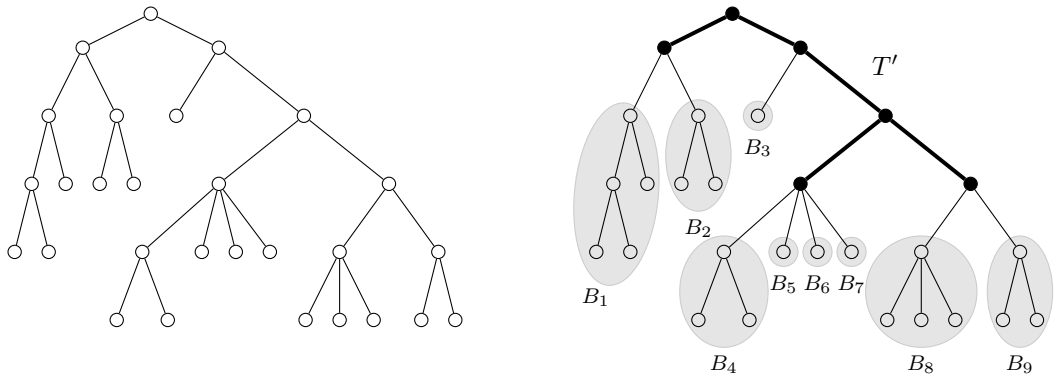
The decomposition has the following key property.

> **Lemma 11 (Alstrup et al.)** *The ART decomposition with parameter χ for a rooted tree T with n leaves produces a top tree with at most $\frac{n}{\chi+1}$ leaves.*

See Figure 4.1 for an illustration of an ART decomposition of a tree.

4.2 Obtaining the Index

Applying an ART decomposition on the suffix tree for t, $T(C)$, with $\chi = \log n$ and $C = \text{suff}(t)$, we obtain a top tree T' and a number of bottom trees $B_1, B_2, \ldots, B_q$ each of size at most $\log n$. From Lemma 11, T' has at most $\frac{n}{\log n}$ leaves and hence $O(\frac{n}{\log n})$ vertices since T' is a compressed trie. To facilitate the search, the top and bottom trees are stored



(a) A tree T. (b) ART decomposition of T.

Figure 4.1: Illustrating the ART decomposition with $\chi = \log n$ on a tree T with $n = 16$ leaves. The ART decomposition of T consists of the bottom trees $B_1, B_2, \ldots, B_9$, each having at most $\log n = 4$ leaves. The top tree T' (marked in bold) has 3 leaves, which is in accordance with Lemma 11.

in a LCP data structure, noting that these compressed tries only contain substrings of t. Using Lemma 4, we add support for unrooted $O(\log \chi + \log \log n) = O(\log \log n)$ time LCP queries on the bottom trees using $O(n)$ additional space in total. For the top tree we apply Lemma 2 to add support for unrooted LCP queries in time $O(\log \log n)$ using $O(\frac{n}{\log n} \log \frac{n}{\log n}) = O(n)$ additional space.

In order to describe the search algorithm, we introduce the following notation. The operation extract($\mathcal{A}$) removes and returns an element from the set $\mathcal{A}$. By nextlocations(ℓ'), we denote the set of all children of ℓ' where the corresponding string is one character longer than the string for ℓ'. We let hasbottom(ℓ', c) be true if there is a bottom tree attached to ℓ' with an edge labeled with character c, and bottom(ℓ', c) denotes the index of that bottom tree in the LCP data structure. The set bottoms(ℓ') contains the root of all bottom trees attached to ℓ'. For any location ℓ', we assume that the index i of the trie $T(C_i)$ containing ℓ' is denoted trie(ℓ').

Algorithm 1 shows how to perform the search in the index. LCP queries are used to search for the subpattern p_i. If p_i is not fully consumed in ℓ', the search cannot continue in the current tree. In this case, we check if there is a bottom tree B_i joined to ℓ' labeled with the next unmatched character of p_i. If so, we search for the remaining part of the subpattern from the root of B_i by a new LCP query. When p_i has been fully matched, the search consumes the next wildcard character in the pattern by branching to all children and any bottom trees of the current location. The resulting locations are added to $\mathcal{A}$, paired with the index of the next subpattern to match. The set $\mathcal{R}$ stores the locations corresponding to the occurrences of p in t.

The algorithm shows that two LCP queries may be performed for each subpattern. This, however, does not influence the asymptotic complexity, so $O(\sigma^i)$ LCP queries are performed for the subpattern p_i. They each take time $O(\log \log n)$, which concludes the proof of Theorem 1.

Algorithm 1 Searching the index from Theorem 1 to find the occurrences a pattern $p = p_0 * p_1 * \ldots * p_j$ using the LCP data structure. If a subpattern is not fully matched, the algorithm checks if there is a bottom tree in which the search can continue.

$\mathcal{A} \leftarrow \{(0, \mathrm{root}(T'))\}$
$\mathcal{R} \leftarrow \emptyset$
while $\mathcal{A} \neq \emptyset$ **do**
  $(i, \ell) \leftarrow$ extract$(\mathcal{A})$
  $\ell' \leftarrow$ LCP$(p_i, \mathrm{trie}(\ell), \ell)$
  if $|\ell'| - |\ell| = |p_i|$ **then**
    if $i = j$ **then**
      $\mathcal{R} \leftarrow \mathcal{R} \cup \{\ell'\}$
    else
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{(i+1, \ell'') \mid \ell'' \in \mathrm{nextlocations}(\ell') \vee \ell'' \in \mathrm{bottoms}(\ell')\}$
  else
    $c \leftarrow p_i[|\ell'| - |\ell| + 1]$
    if hasbottom(ℓ', c) **then**
      $\ell^B \leftarrow$ LCP$(\mathrm{suff}_{|\ell'|-|\ell|+2}(p_i), \mathrm{bottom}(\ell', c))$
      if $(|\ell'| - |\ell| + 1) + |\ell^B| = |p_i|$ **then**
        if $i = j$ **then**
          $\mathcal{R} \leftarrow \mathcal{R} \cup \{\ell^B\}$
        else
          $\mathcal{A} \leftarrow \mathcal{A} \cup \{(i+1, \ell'') \mid \ell'' \in \mathrm{nextlocations}(\ell^B)\}$
Report descendant leaves of locations in $\mathcal{R}$

5

A TIME-SPACE TRADE-OFF FOR k-BOUNDED WILDCARD INDEXES

In this chapter we will show Theorem 2. We first introduce the heavy α-tree decomposition in Section 5.1. In Section 5.2 we use the decomposition to define wildcard trees, leading to the wildcard index with a time-space trade-off described in Section 5.3. Finally in Section 5.4, we apply the LCP data structure to improve the query time and obtain a generalization of the wildcard index by Cole et al. [13].

5.1 Heavy α-Tree Decomposition

The *heavy α-tree decomposition* is a generalization of the well-known heavy path decomposition introduced by Harel and Tarjan [22]. The purpose is to decompose a rooted tree T into a number of *heavy trees* joined by light edges, such that a path from any location to the root of T traverses at most a logarithmic number of heavy trees. For use in the construction, we define a proper weight function on the vertices of T to be a function satisfying

$$\text{weight}(v) \geq \sum_{w \text{ child of } v} \text{weight}(w) \ .$$

Observe that using the number of vertices or the number of leaves in the subtree rooted at v as the weight of v satisfies this property. The decomposition is parameterized by an integer $\alpha \geq 0$ and constructed by classifying edges in T as being heavy or light according to the following rule.

> **Classification Rule:** For every vertex $v \in T$, the edges to the α heaviest children of v (breaking ties arbitrarily) are heavy, and the remaining edges are light.

Observe that for $\alpha = 1$, this results in a heavy path-decomposition. Given a heavy α-tree decomposition of T, we define $\text{lightdepth}(v)$ to be the number of light edges on a path from the vertex $v \in T$ to the root of T. The key property of this construction is captured by the following lemma.

Lemma 12 *For any heavy α-tree decomposition with $\alpha > 0$ of a rooted tree T, and for any vertex $v \in T$*

$$\text{lightdepth}(v) \leq \log_{\alpha+1} \text{weight}(\text{root}(T)) \ .$$

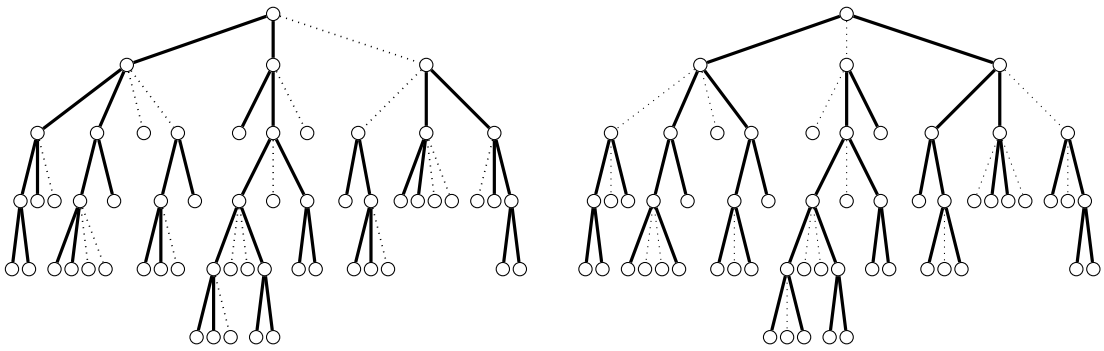
Proof Consider a light edge from a vertex v to its child w. We prove that $\text{weight}(w) \leq \frac{1}{\alpha+1}\text{weight}(v)$, implying that $\text{lightdepth}(v) \leq \log_{\alpha+1}\text{weight}(\text{root}(T))$. To obtain a contradiction, suppose that $\text{weight}(w) > \frac{1}{\alpha+1}\text{weight}(v)$. In addition to w, v must have α heavy children each of which has a weight greater than or equal to $\text{weight}(w)$. Hence

$$\text{weight}(v) \geq (1+\alpha) \cdot \text{weight}(w) > (1+\alpha) \cdot \frac{1}{\alpha+1}\text{weight}(v) = \text{weight}(v)$$

which is a contradiction. ∎

Lemma 12 holds for any heavy α-tree decomposition obtained using a proper weight function on T. In the remaining chapters of the thesis we will assume that the weight of a vertex is the number of leaves in the subtree rooted at v. Given a heavy α-tree decomposition of a rooted tree T, we define $\text{lightheight}(T)$ to be the maximum light depth of a vertex in T, and remark that for $\alpha = 0$, $\text{lightheight}(T) = \text{height}(T)$. See Figure 5.1 for two different examples of heavy α-tree decompositions of a tree.

For a vertex v in a compressed trie $T(S)$, we let $\text{lightstrings}(v)$ denote the set of strings starting in one of the light edges leaving v. That is, $\text{lightstrings}(v)$ is the union of the set of strings in $T_\ell(S)$ for all locations ℓ on the light edges from v where $|\ell| = |v| + 1$.



(a) A heavy α-tree decomposition of a tree T. (b) Another heavy α-tree decomposition of T.

Figure 5.1: Illustrating the heavy α-tree decomposition for $\alpha = 2$ on a tree with $n = 38$ leaves. In the two decompositions, the maximum light depth is 3 and 2, respectively, which is in accordance with Lemma 12.

5.2 Wildcard Trees

We introduce the (β, k)-*wildcard tree*, denoted $T^k_\beta(C')$, where $1 \leq \beta < \sigma$ and $k > 0$ are chosen integer parameters. This data structure stores a set C' of modified substrings of the

indexed text t in a compressed trie. The key property is that the search for a pattern p with at most k wildcards branches to at most β locations in $T^k_\beta(C')$ when consuming a single wildcard of p. In particular for $\beta = 1$, the search for p never branches and the search time becomes linear in the length of p.

For a vertex v, we define the *wildcard height* of v to be the number of wildcards on the path from v to the root. Intuitively, given a wildcard tree that supports i wildcards, support for an extra wildcard is added by joining a new tree to each vertex v having wildcard height i with an edge labeled $*$. This new tree is searched if a wildcard is consumed in v. More precisely, $T^k_\beta(C')$ is built recursively as follows.

> **Construction of** $T^i_\beta(S)$: Produce a heavy $(\beta - 1)$-tree decomposition of $T(S)$, then for each internal vertex $v \in T(S)$, join the root of $T^{i-1}_\beta(\mathrm{suff}_2(\mathrm{lightstrings}(v))$ to v by an edge labeled $*$. Let $T^0_\beta(S) = T(S)$.

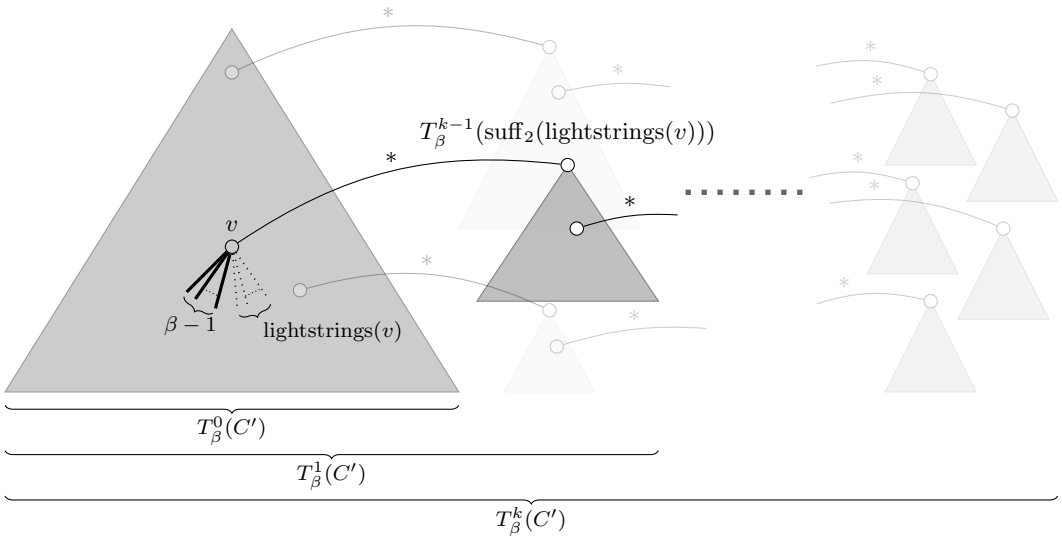
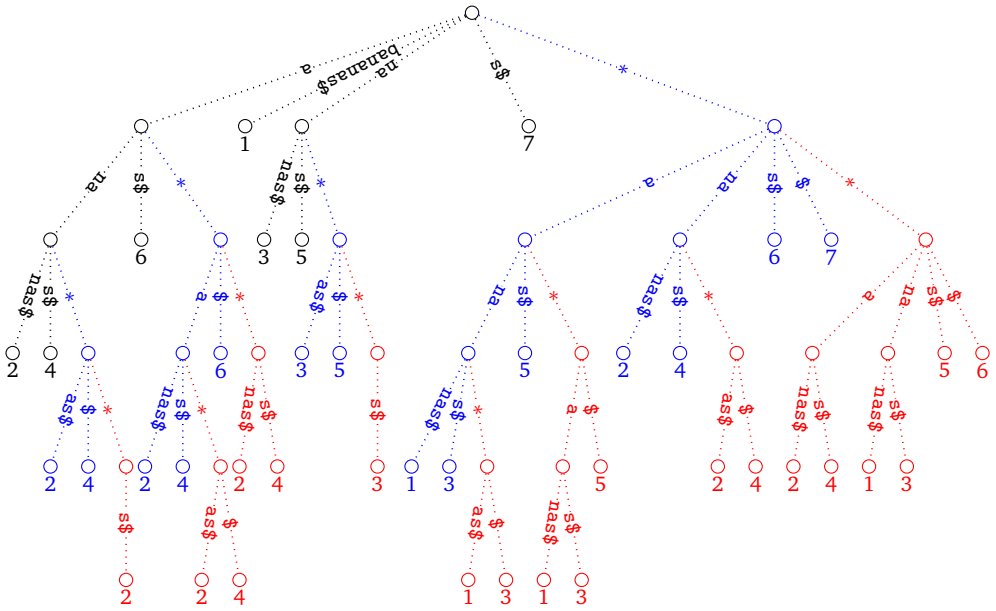


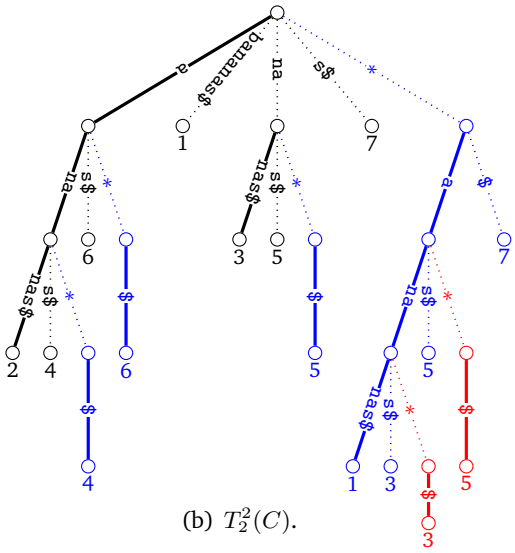
Figure 5.2: Abstract illustration of the recursive construction of the wildcard tree $T^k_\beta(C')$. The final tree consists of compressed tries joined by edges labeled $*$ and organized into k layers.

As illustrated by Figure 5.2, the final tree $T^k_\beta(C')$ can be regarded as a number of compressed tries joined by edges labeled $*$. Each compressed trie $T(S)$ stores at most $|C'|$ selected suffixes of the strings in C', since S is a subset of $\mathrm{suff}_d(C')$ for some d. We will use this observation in the following sections.

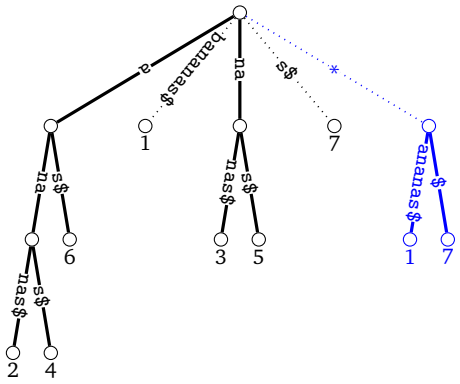
Since a leaf ℓ in a compressed trie $T(S)$ is obtained as the suffix of a string $x \in C'$, we assume that ℓ inherits the label of x in case the strings in C' are labeled. For example, when C' denotes the suffixes of t, we will label each suffix in C' with its start position in t. This immediately provides us with an k-unbounded wildcard index. Figure 5.3 shows some concrete examples of the construction of $T^k_\beta(C')$ when C' is a set of labeled suffixes.



(a) $T_1^2(C)$.



(b) $T_2^2(C)$.



(c) $T_3^2(C)$.

Figure 5.3: Illustrating the recursive construction of $T_\beta^k(C)$ for $\beta \in \{1, 2, 3\}$, $k = 2$ and $C = \text{suff}(\texttt{bananas\$})$. The recursion levels 0, 1, 2 in the construction are colored black, blue, red respectively. All edges in $T_1^2(C)$ are light, since the construction is based on a heavy α-tree decomposition with $\alpha = \beta - 1 = 0$. The leafs are labeled with the start position of their corresponding suffix in t.

5.3 Wildcard Tree Index

Given a set C' of substrings of t and a pattern p, we can identify the strings of C' having a prefix matching p by constructing $T^k_\beta(C')$. Searching $T^k_\beta(C')$ is similar to the suffix tree search, except when consuming a wildcard character of p in an explicit vertex $v \in T^k_\beta(C')$ with more than β children. In that case the search branches to the root of the wildcard tree joined to v and to the first location on the $\beta - 1$ heavy edges of v, effectively letting the wildcard match the first character on all edges from v.

More precisely, the search for p is carried out as described in Algorithm 2.

Algorithm 2 Searching a wildcard tree for a pattern $p = p_0 * p_1 * \ldots * p_j$.

$\mathcal{A} \leftarrow \{\text{root}(T^k_\beta(C'))\}$
$\mathcal{R} \leftarrow \emptyset$
while $\mathcal{A} \neq \emptyset$ **do**
  $\ell \leftarrow$ extract($\mathcal{A}$)
  if $|\ell| = |p|$ **then**
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{\ell\}$
  else
    if $p[|\ell| + 1] = *$ **then**
      $\mathcal{A} \leftarrow \mathcal{A} \cup$ nextheavylocations(ℓ)
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{$nextlocation$(\ell, p[|\ell| + 1])\}$
Report descendant leaves of locations in $\mathcal{R}$

The algorithm maintains a set $\mathcal{A}$ of active locations, which initially only contains the root of $T^k_\beta(C')$. A location ℓ is repeatedly extracted from $\mathcal{A}$, and the search continues from ℓ by adding a (possibly empty) set of new locations to $\mathcal{A}$. When matching a character c of p in a location ℓ, the location reached by following the character c from ℓ, denoted nextlocation(ℓ, c), is added to $\mathcal{A}$. Additionally, when matching a wildcard, the set of all heavy child locations of ℓ, denoted nextheavylocations(ℓ), is added to $\mathcal{A}$. The algorithm terminates with a set of locations $\mathcal{R}$, each corresponding to at least one occurrence of p. The strings in C' having p as a prefix are found by reporting the leaves below the locations in $\mathcal{R}$ in time $O(occ)$.

5.3.1 Time and Space Analysis

For each of the j wildcards in p, the size of $\mathcal{A}$ is increased by a factor of at most β. Hence the search for p branches to a total of at most $\sum_{i=0}^{j} \beta^i = O(\beta^j)$ locations, each of which requires $O(m)$ time, resulting in query time $O(\beta^j m + occ)$. For $\beta = 1$ the query time becomes $O(m + j + occ)$, since nextheavylocations$(\ell) = \emptyset$ for all ℓ.

We will now show the space required to store $T^k_\beta(C')$, resulting in Lemma 13.

Lemma 13 *For any integer $1 \leq \beta < \sigma$, the wildcard tree $T^k_\beta(C')$ has query time $O(\beta^j m + occ)$ for $1 < \beta < \sigma$ and $O(m + j + occ)$ for $\beta = 1$. The wildcard tree stores*

$O(|C'|h^k)$ strings, where h is an upper bound on the light height of all compressed tries $T(S)$ satisfying $S \subseteq \text{suff}_d(C')$ for some integer d.

In the paper by Cole et al. [13] the authors give a proof by induction of the space needed to store a wildcard tree for $\beta = 2$. See Appendix B for an explanation of their proof and a discussion of a potential problem in it. We give a more general proof of the space usage that works for any value of β.

Proof We prove that the total number of strings (leaves) in $T^i_\beta(S)$, denoted $|T^i_\beta(S)|$, is at most $|S|\sum_{j=0}^{i} h^j = O(|S|h^i)$. The proof is by induction on i. The base case $i = 0$ holds, since $T^0_\beta(S) = T(S)$ contains $|S| = |S|\sum_{j=0}^{0} h^j$ strings. For the induction step, assume that $|T^i_\beta(S)| \leq |S|\sum_{j=0}^{i} h^j$. Let $S_v = \text{suff}_2(\text{lightstrings}(v))$ for a vertex $v \in T(S)$. From the construction we have that the number of strings in $T^{i+1}_\beta(S)$ is the number of strings in $T(S)$ plus the number of strings in the wildcard trees joined to the vertices of $T(S)$. That is,

$$\big|T^{i+1}_\beta(S)\big| \;=\; \big|S\big| + \sum_{v\in T(S)} \big|T^i_\beta(S_v)\big| \;\leq\; \big|S\big| + \sum_{v\in T(S)} \big|S_v\big| \sum_{j=0}^{i} h^j \;.$$

The string sets S_v consist of suffixes of strings in S. Consider a string $x \in S$, i.e., a leaf in $T(S)$. The number of times a suffix of x appears in a set S_v is equal to the light depth of x in $T(S)$. From the construction, S is a subset of $\text{suff}_d(C')$ for some d, and hence h is an upper bound on the maximum light depth of $T(S)$. This establishes that

$$\sum_{v\in T(S)} |S_v| \;\leq\; |S|h \;,$$

thus showing that $|T^{i+1}_\beta(S)| \leq |S| + |S|h\sum_{j=0}^{i} h^j = |S|\sum_{j=0}^{i+1} h^j$. ∎

Constructing the wildcard tree $T^k_\beta(C)$, for $C = \text{suff}(t)$, we obtain a wildcard index with the following properties.

Lemma 14 *Let t be a string of length n from an alphabet of size σ. For $2 \leq \beta < \sigma$ there is a k-bounded wildcard index for t using $O\big(n \log^k_\beta n\big)$ space. The index can report the occurrences of a pattern with m characters and $j \leq k$ wildcards in time $O\left(\beta^j m + occ\right)$.*

Proof The query time follows from Lemma 13. Since $T^k_\beta(C)$ is a compressed trie, and because each edge label is a substring of t, the space needed to store $T^k_\beta(C)$ is upper bounded by the number of strings it contains which by Lemma 13 is $O(nh^k)$. Since $\alpha = \beta - 1$, it follows from Lemma 12 that $h = \log_\beta n$ is an upper bound on the light height of all compressed tries $T(S)$ satisfying $S \subseteq \text{suff}_d(C)$ for some d, since they contain at most n vertices. Consequently, the space needed to store the index is $O(n \log^k_\beta n)$. ∎

5.4 Wildcard Tree Index Using the LCP Data Structure

In this section we prove Theorem 2 by showing how LCP queries can be used to speed up the query time of the wildcard tree index described in the previous section.

The wildcard index of Lemma 14 reduces the branching factor of the suffix tree search from σ to β, but still has the drawback that the search for a subpattern p_i from an active location $\ell \in T^k_\beta(C)$ takes $O(|p_i|)$ time. This can be addressed by combining the index with the LCP data structure as in the paper by Cole et al. [13]. In that way, the search for a subpattern can be done in time $O(\log \log n)$. The index is obtained by modifying the construction of $T^i_\beta(S)$ such that each $T(S)$ is added to the LCP data structure prior to joining the $(\beta, i-1)$-wildcard trees to the vertices of $T(S)$. For all $T(S)$ except the final $T(S) = T^0_\beta(S)$, support for unrooted LCP queries in time $O(\log \log n)$ is added using additional $O(|S| \log |S|)$ space. For the final $T(S)$ we only need support for rooted queries since at this point all wildcards in p have been consumed. Upon receiving the query pattern $p = p_1 * p_2 * \ldots * p_j$, each p_i is preprocessed in time $O(|p_i|)$ to support LCP queries for any suffix of p_i.

The search for p in the modified index proceeds as described in Algorithm 3. The algorithm is similar to that for the normal wildcard tree index, except now LCP queries are used to search for $p_0, p_1, \ldots, p_j$. After performing a LCP query for p_i from ℓ, the search only continues from the new location ℓ' if the full subpattern p_i was matched. In that case, the search branches to the heavy children and the root of the wildcard tree joined to ℓ', since the next character in the search is a wildcard. If the last subpattern was matched, the resulting location is an occurrence of p in t, and all strings in C having that location as an ancestor are reported.

Algorithm 3 Searching a wildcard tree to find the occurrences a pattern $p = p_0 * p_1 * \ldots * p_j$ using the LCP data structure.

$\mathcal{A} \leftarrow \{(0, \text{root}(T^k_\beta(C')))\}$
$\mathcal{R} \leftarrow \emptyset$
while $\mathcal{A} \neq \emptyset$ **do**
  $(i, \ell) \leftarrow$ extract($\mathcal{A}$)
  $\ell' \leftarrow$ LCP(p_i, trie(ℓ), ℓ)
  if $|\ell'| - |\ell| = |p_i|$ **then**
    if $i = j$ **then**
      $\mathcal{R} \leftarrow \mathcal{R} \cup \{\ell'\}$
    else
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{(i+1, \ell'') \mid \ell'' \in$ nextheavylocations(ℓ) $\vee\ \ell'' =$ nextlocation($\ell', *$)$\}$
Report descendant leaves of locations in $\mathcal{R}$

5.4.1 Time and Space Analysis

In the search for p a total of at most $\sum_{i=0}^{j} \beta^i = O(\beta^j)$ LCP queries each taking time $O(\log \log n)$ are performed. Preprocessing $p_0, p_1, \ldots, p_j$ takes $\sum_{i=0}^{j} |p_i| = m$ time, so the

query time is $O(m + \beta^j \log\log n + occ)$. The space needed to store the index is $O(n \log_\beta^k n)$ for $T_\beta^k(C)$ plus the space needed to store the LCP data structure. Adding support for rooted LCP queries requires linear space in the total size of the compressed tries, which is $O(n \log_\beta^k n)$. Let $T(S_0), T(S_1), \ldots, T(S_q)$ denote the compressed tries with support for unrooted LCP queries. Since each S_i contains at most n strings and $\sum_{i=0}^{q} |S_i| = |T_\beta^{k-1}(C)|$, by Lemma 2, the additional space required to support unrooted LCP queries is

$$O\left(\sum_{i=0}^{q} |S_i| \log |S_i|\right) = O\left(\log n \sum_{i=0}^{q} |S_i|\right) = O\left(\log n |T_\beta^{k-1}(C')|\right) = O\big(n \log(n) \log_\beta^{k-1} n\big)\ ,$$

which is an upper bound on the total space required to store the wildcard index. This concludes the proof of Theorem 2.

The k-bounded wildcard index described by Cole et al. [13] is obtained as a special case of Theorem 2.

Corollary 3 (Cole et al.) *Let t be a string of length n from an alphabet of size σ. There is a k-bounded wildcard index for t using $O(n \log^k n)$ space. The index can report the occurrences of a pattern with m characters and $j \leq k$ wildcards in time $O(m + 2^j \log\log n + occ)$.*

6

A k-BOUNDED WILDCARD INDEX WITH OPTIMAL QUERY TIME

In this chapter we show Theorem 3. In Section 6.1 we first obtain a k-bounded wildcard index similar to Theorem 2 by using Lemma 13. We use this index to construct a black-box reduction for obtaining optimal time indexes. In Section 6.2 the black-box reduction is used on the wildcard index from Theorem 1 to obtain Theorem 3.

6.1 A Black-Box Reduction

Consider the k-bounded wildcard index which ensures a linear query time, obtained by creating the wildcard tree $T_1^k(\text{suff}(t))$ for t. We can show that the space usage for the index depends on the height of the suffix tree for t.

Lemma 15 *Let t be a string of length n from an alphabet of size σ. There is a k-bounded wildcard index for t using $O(nh^k)$ space, where h is the height of the suffix tree for t. The index can report the occurrences of a pattern with m characters and j wildcards in time $O(m + j + occ)$.*

Proof Since $\text{suff}(t)$ is closed under the suffix operation, the height of $T(\text{suff}(t))$ is an upper bound on the height of all compressed tries $T(S)$ satisfying $S \subseteq \text{suff}_d(\text{suff}(t))$ for some d. For $\beta = 1$, the light height of $T(S)$ is equal to the height of $T(S)$, so $h = \text{height}(T(\text{suff}(t)))$ can be used as an upper bound of the light height in Lemma 13, and consequently the space needed to store $T_1^k(\text{suff}(t))$ is $O(nh^k)$. ∎

In the worst case the height of the suffix tree is close to n, but combining the index with another wildcard index yields a useful black-box reduction. The idea is to query the first index if the pattern is short, and the second index if the pattern is long.

Lemma 16 *Let $F \geq m$ and let G be independent of m and j. Given a wildcard index $\mathcal{W}$ with query time $O(F + G + occ)$ and space usage $O(|\mathcal{W}|)$, there is a k-bounded wildcard index $\mathcal{W}'$ with query time $O(F + j + occ)$ and taking space $O(n \min(G, h)^k + |\mathcal{W}|)$, where h is the height of the suffix tree for t.*

Proof The wildcard index $\mathcal{W}'$ consists of $\mathcal{W}$ and a special k-bounded wildcard index $\mathcal{W}'' = T_1^k(\text{pref}_G(\text{suff}(t)))$, which is a wildcard tree with $\beta = 1$ over the set of all substrings of t of length G. Since the maximum string depth in $\mathcal{W}''$ is G, this value can be used as an upper bound for the light height in Lemma 13. Consequently, the space required to store $\mathcal{W}''$ is $O(n \min(G, h)^k)$ by using Lemma 15 if $G > h$, where h is the height of the suffix tree for t. See Figure 6.1 for an illustration of the construction of $\mathcal{W}'' = T_1^k(\text{pref}_G(\text{suff}(t)))$.

A query on $\mathcal{W}'$ results in a query on either $\mathcal{W}$ or $\mathcal{W}''$. In case $F + j > G$, we query $\mathcal{W}$ and the query time will be $O(F + G + occ) = O(F + j + occ)$. In case $F + j \leq G$, we query $\mathcal{W}''$ with query time $O(m + j + occ) = O(F + j + occ)$. In any case, the query time of $\mathcal{W}'$ is $O(F + j + occ)$. ■

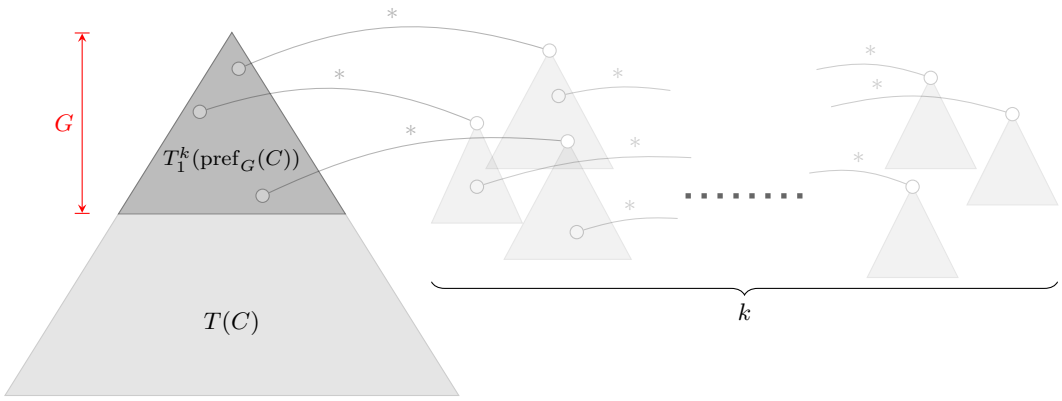


Figure 6.1: Illustrating how the wildcard tree $T_1^k(\text{pref}_G(C))$ for $C = \text{suff}(t)$ is obtained by slicing off the top of the compressed trie $T(C)$ with string depth G and appending k additional layers of wildcard trees recursively. For reference, Figure 3.3(c) shows a concrete example of constructing $T(\text{pref}_G(C))$ for $G = 3$.

6.2 Obtaining the Index

Applying Lemma 16 with $F = m$ and $G = \sigma^k \log\log n$ on the unbounded wildcard index from Theorem 1 yields a new k-bounded wildcard index with optimal query time $O(F + j + occ) = O(m + j + occ)$. The index uses space

$$O(nG^k + n) = O(n(\sigma^k \log\log n)^k) = O(\sigma^{k^2} n \log^k \log n) \ .$$

This concludes the proof of Theorem 3.

7

VARIABLE LENGTH GAPS

In this chapter we consider the *string indexing for patterns with variable length gaps problem*. We show that this problem can be solved, using the bounded and unbounded wildcard indexes described in the previous chapters, by only changing the search algorithms. Section 7.1 introduces the problem, Section 7.2 describes previous work and Section 7.3 gives an overview of our solutions. In Section 7.4 we account for the changes necessary for supporting variable length gaps and analyse the modified search algorithm.

7.1 Introduction

The string indexing for patterns with variable length gaps problem is to build an index for a string t that can efficiently report the occurrences of a query pattern p of the form

$$p = p_0 *\{a_1, b_1\}\ p_1 *\{a_2, b_2\}\ \ldots\ *\{a_j, b_j\}\ p_j\ .$$

The query pattern consists of $j + 1$ strings $p_0, p_1, \ldots, p_j \in \Sigma^*$ interleaved by j *variable length gaps* $*\{a_i, b_i\}$, $i = 1, \ldots, j$, where a_i and b_i are positive integers such that $a_i \leq b_i$. Intuitively, a variable length gap $*\{a_i, b_i\}$ matches an arbitrary string over Σ of length between a_i and b_i, both inclusive.

Example Consider the string t and pattern p over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$.

$$t = \mathtt{acbccbacccddabdaabcdccbccdaa}$$
$$p = \mathtt{b}*\{0, 4\}\mathtt{cc}*\{3, 5\}\mathtt{d}$$

The string t contains five occurrences of the query pattern p as shown in Figure 7.1.

As shown by the example, different occurrences of the query pattern p can start or end at the same position in t, and the same substring in t can contain multiple occurrences of p. Hence to completely characterize an occurrence of p in t, we need to report the positions of the individual subpatterns $p_0, p_1, \ldots, p_j$ for each full occurrence of the pattern. However, in the following we will restrict our attention to reporting the start and end position of

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
a	**c**	**b**	**c**	**c**	**b**	**a**	**c**	**c**	**c**	**d**	**d**	**a**	**b**	**d**	**a**	**a**	**b**	**c**	**d**	**c**	**c**	**b**	**c**	**c**	**d**	**a**	**a**
		b	c	c	∗	∗	∗	∗	∗	d																	
		b	∗	∗	∗	∗	c	c	∗	∗	∗	∗	∗	d													
					b	∗	c	c	∗	∗	∗	∗	∗	d													
					b	∗	∗	c	c	∗	∗	∗	∗	d													
																	b	∗	∗	c	c	∗	∗	∗	d		

Figure 7.1: The five occurrences of $p = \texttt{b}*\{0,4\}\texttt{cc}*\{3,5\}\texttt{d}$ in $t = \texttt{acbccbacccddabdaabcdccbccdaa}$.

each occurrence of p in t. For the above example, we would thus report the pairs $(3, 11)$, $(3, 15)$, $(6, 15)$ and $(18, 26)$.

String indexing for patterns with variable length gaps has applications in information retrieval, data mining and computational biology [18, 19, 31, 30, 33]. In particular, the PROSITE data base [23, 7] uses patterns with variable length gaps to identify and classify protein sequences. The problem is a generalization of string indexing for patterns with wildcards, since a wildcard $*$ is equivalent to the variable length gap $*\{1, 1\}$. Variable length gaps are also known as *bounded wildcards*, as a variable length gap $*\{a_i, b_i\}$ can be regarded as a bounded sequence of wildcards.

String indexing for patterns with variable length gaps is equivalent to string indexing for patterns with wildcards, with the addition of allowing *optional wildcards* in the pattern. An optional wildcard matches any character from Σ or the empty string, i.e., an optional wildcard is equivalent to the variable length gap $*\{0, 1\}$. Conversely, we may also consider a variable length gap $*\{a_i, b_i\}$ as a_i consecutive wildcards followed by $b_i - a_i$ consecutive optional wildcards.

In the following we let $A = \sum_{i=1}^{j} a_i$ and $B = \sum_{i=1}^{j} b_i$ denote the sum of the lower and upper bounds on the variable length gaps in p, respectively. Hence A and $B - A$ denote the number of normal and optional wildcards in p, respectively. A wildcard index with support for optional wildcards is called an *optional wildcard index*. As for wildcard indexes, we distinguish between bounded and unbounded optional wildcard indexes. A (k, o)-bounded optional wildcard index supports patterns containing $A \leq k$ normal wildcards and $B - A \leq o$ optional wildcards. An unbounded optional wildcard index supports patterns with no restriction on the number of normal and optional wildcards..

7.2 Previous Work

Lam et al. [26] introduced optional wildcards in the pattern and presented a variant of their solution for the string indexing for patterns with wildcards problem. The idea is to determine potential matches and verify complete matches using interval stabbing on the possible positions for the subpatterns. This leads to an unbounded optional wildcard index with query time $O(m + Bj \min_{0 \leq i \leq j} occ(p_i, t))$ and space usage $O(n)$. As before $occ(p_i, t)$ is $\Theta(n)$ in the worst case, so the query time is $\Theta(Bjn)$ in the worst case.

Besides the result by Lam et al., we have no knowledge of solutions for the string indexing for patterns with variable length gaps problem. Some results are known for the *string matching with variable length gaps problem*, where the text string may not be preprocessed in advance. Bille et al. [6] recently presented the best known solution for the matching problem with query time $O((n+m)\log j + occ)$ using space $O(m+A)$. We refer to their paper for a review of the previous work on the string matching with variable length gaps problem.

7.3 Our Results

By modifying the search algorithm, we show that our solutions to string indexing for patterns with wildcards also support variable length gaps in the pattern, leading to the following new theorems.

Theorem 4 *Let t be a string of length n from an alphabet of size σ. There is an unbounded optional wildcard index for t using $O(n)$ space. The index can report the occurrences of a pattern with m characters, A wildcards and $B-A$ optional wildcards in time $O(m + 2^{B-A}\sigma^B \log\log n + occ)$, where $A = \sum_{i=1}^{j} a_i$ and $B = \sum_{i=1}^{j} b_i$.*

Theorem 5 *Let t be a string of length n from an alphabet of size σ. For $2 \leq \beta < \sigma$, there is a (k,o)-bounded optional wildcard index for t using $O\big(n\log(n)\log_\beta^{k+o-1} n\big)$ space. The index can report the occurrences of a pattern with m characters, $A \leq k$ wildcards and $B - A \leq o$ optional wildcards in time $O\left(m + 2^{B-A}\beta^B \log\log n + occ\right)$, where $A = \sum_{i=1}^{j} a_i$ and $B = \sum_{i=1}^{j} b_i$.*

Theorem 6 *Let t be a string of length n from an alphabet of size σ. There is a (k,o)-bounded optional wildcard index for t using $O(\sigma^{(k+o)^2} n \log^{k+o}\log n)$ space. The index can report the occurrences of a pattern with m characters, $A \leq k$ wildcards and $B - A \leq o$ optional wildcards in time $O(2^{B-A}(m+B) + occ)$, where $A = \sum_{i=1}^{j} a_i$ and $B = \sum_{i=1}^{j} b_i$.*

These results completely generalize our previous solutions, since if the query pattern only contains variable length gaps of the form $*\{1,1\}$, the problem reduces to string indexing for patterns with wildcards. In that case $A = B = j$ and we obtain exactly Theorem 1, Theorem 2 and Theorem 3.

Compared to the only known index for the problem by Lam et al. [26], Theorem 4 gives an index that matches the $O(n)$ space usage, but improves the worst-case query time from $\Theta(Bjn)$ to $O\left(m + 2^{B-A}\sigma^B \log\log n + occ\right)$, provided that $B \leq \log_\sigma \sqrt{nj}$.

7.4 Supporting Variable Length Gaps

As remarked a variable length gap $*\{a_i, b_i\}$ is equivalent to a_i wildcards followed by $b_i - a_i$ optional wildcards. Hence to support variable length gaps, we only have to describe how the search algorithms must be modified to match an optional wildcard in p. Essentially the

difference between matching a normal wildcard and an optional one is how the search handles the location where the wildcard is matched. For a normal wildcard the location is removed from the set of active locations, but for an optional wildcard the location is kept active. A sequence of j optional wildcards is simulated as a sequence of j wildcards, where all intermediate locations reached are kept active.

As an example, consider the unbounded wildcard index obtained by adding support for unrooted LCP queries on the suffix tree $T(\mathrm{suff}(t))$ for the indexed string t. Algorithm 4 illustrates how to search $T(\mathrm{suff}(t))$ for a pattern p containing variable length gaps. The algorithm is similar to Algorithm 3, except when a variable length gap $*\{a_{i+1}, b_{i+1}\}$ is matched in a location ℓ'. In this case, the for-loop in the algorithm simulates the variable length gap as a_{i+1} normal wildcards followed by $b_{i+1} - a_{i+1}$ optional wildcards. At the start of each iteration of the for-loop f wildcards have been simulated. For each wildcard simulated, $\mathcal{B}'$ holds the resulting set of locations in $T(\mathrm{suff}(t))$ reached, while $\mathcal{B}$ contains the active locations at the beginning of each iteration for f. When a new iteration for f starts, $\mathcal{B}$ contains the active locations after f wildcards were simulated. If $a_{i+1} \leq f \leq b_{i+1}$, the currently considered location o is kept active, since the next wildcard to simulate is optional.

Extending the algorithm to support searching in wildcard trees is just a matter of changing the way the search branches. Instead of branching on all outgoing edges, it must instead branch to the heavy children and follow the edge labeled $*$ for each wildcard simulated. We refer to Algorithm 3 for how the branching should be performed.

7.4.1 Reporting Occurrences

To report the substrings in t where the query pattern occurs, we assume that each leaf ℓ in $T(\mathrm{suff}(t))$ has been labeled by the start position, $\mathrm{pos}(\ell)$, of the suffix in t it corresponds to. When Algorithm 4 terminates, each location $\ell' \in \mathcal{R}$ corresponds to one or more substrings in t, where the query pattern p occurs. To report the start and end position of these substrings, we traverse the subtree rooted at ℓ' and identify the leaves $\ell_0, \ell_1, \ldots, \ell_r$ corresponding to suffixes of t having ℓ' as a prefix. The start and end positions of these substrings are then given by

$$\big(\mathrm{pos}(\ell_0), \mathrm{pos}(\ell_0) + |\ell'|\big), \big(\mathrm{pos}(\ell_1), \mathrm{pos}(\ell_1) + |\ell'|\big), \ldots, \big(\mathrm{pos}(\ell_r), \mathrm{pos}(\ell_r) + |\ell'|\big) \ .$$

7.4.2 Analysis of the Modified Search

To analyse the running time of Algorithm 4, we will bound the maximum number of LCP queries performed during the search for the query pattern

$$p = p_0 *\{a_1, b_1\}\ p_1 *\{a_2, b_2\}\ \ldots\ *\{a_j, b_j\}\ p_j \ .$$

We define $A_i = \sum_{l=1}^{i} a_i$ and $B_i = \sum_{l=1}^{i} b_i$. The number of normal and optional wildcards preceding the subpattern p_i in p is A_i and $B_i - A_i$, respectively. To bound the number of locations in which a LCP query for the subpattern p_i can start, we choose and promote $l = 0, 1, \ldots, B_i - A_i$ of the preceding optional wildcards to normal wildcards and discard

Algorithm 4 Searching the unbounded wildcard index $T(\mathrm{suff}(t))$ to find the occurrences a pattern $p = p_0 *\{a_1, b_1\}\ p_1 *\{a_2, b_2\}\ \ldots\ *\{a_j, b_j\}\ p_j$ using the LCP data structure.

$\mathcal{A} \leftarrow \{(0, \mathrm{root}(T(\mathrm{suff}(t))))\}$
$\mathcal{R} \leftarrow \emptyset$
while $\mathcal{A} \neq \emptyset$ **do**
  $(i, \ell) \leftarrow$ extract($\mathcal{A}$)
  $\ell' \leftarrow$ LCP(p_i, trie(ℓ), ℓ)
  if $|\ell'| - |\ell| = |p_i|$ **then**
    if $i = j$ **then**
      $\mathcal{R} \leftarrow \mathcal{R} \cup \{\ell'\}$
    else
      $\mathcal{B} \leftarrow \{\ell'\}$
      for $f \leftarrow 0$ **to** $b_{i+1} - 1$ **do**
        $\mathcal{B}' \leftarrow \emptyset$
        while $B \neq \emptyset$ **do**
          $o \leftarrow$ extract($\mathcal{B}$)
          if $f < a_{i+1}$ **then**
            $\mathcal{B}' \leftarrow \mathcal{B}' \cup$ nextlocations(o)
          else
            $\mathcal{B}' \leftarrow \mathcal{B}' \cup$ nextlocations(o) $\cup \{o\}$
        $\mathcal{B} \leftarrow \mathcal{B}'$
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{(i+1, \ell'') \mid \ell'' \in \mathcal{B}\}$
For each location in $\mathcal{R}$ report the corresponding substrings in t

the rest. For a specific choice there are exactly $A_i + l$ wildcards preceding p_i, and thus the number of locations in which a LCP query for p_i can start is at most β^{A_i+l}. The term β is an upper bound on the branching factor of the search when consuming a wildcard. For the index $T(\mathrm{suff}(t))$ searched by Algorithm 4, the branching factor is $\beta = \sigma$, but indexes based on wildcard trees can have a smaller branching factor. There are $\binom{B_i - A_i}{l}$ possibilities for choosing the l optional wildcards, so the number of locations in which a LCP query for p_i can start is at most

$$\sum_{l=0}^{B_i - A_i} \binom{B_i - A_i}{l} \beta^{A_i + l} \ \leq\ 2^{B_i - A_i} \beta^{B_i}. \tag{7.1}$$

Summing over the $j + 1$ subpatterns, we obtain the following bound for the total number of LCP queries performed during a search for the query pattern p.

$$\sum_{i=0}^{j} 2^{B_i - A_i} \beta^{B_i} \ =\ O\left(2^{B-A} \beta^{B}\right)\ .$$

Since the search algorithm performs LCP queries in time $O(\log \log n)$ and has to preprocess the pattern in time $O(m)$, the total query time is $O(m + 2^{B-A}\beta^B \log \log n + occ)$. This concludes the proof of Theorem 4 and Theorem 5.

To show Theorem 6, we apply a black-box reduction very similar to Lemma 16, leading to a (k, o)-bounded optional wildcard index, where k and o are parameters chosen in advance. This index consists of the following two optional wildcard indexes. A query is performed on one of these indexes depending on the length $m + B$ of the received query pattern p.

1. The unbounded optional wildcard index given by Theorem 4. This index has query time $O(m + 2^{B-A}\sigma^B \log\log n + occ)$ and uses space $O(n)$.

2. The (k, o)-bounded optional wildcard index obtained by constructing the wildcard tree $T_1^{k+o}(\text{pref}_G(\text{suff}(t)))$ without the LCP data structure, where $G = \sigma^{k+o} \log\log n$. From Equation 7.1 with $\beta = 1$, the search for the subpattern p_i can start from at most $2^{B_i - A_i}$ locations. Searching for p_i from each of these locations takes time $O(|p_i| + b_i)$, since the LCP data structure is not used and the tree must be traversed one character at a time. Summing over the $j + 1$ subpatterns, we obtain the following query time for the index

$$O\left(\sum_{i=0}^{j} 2^{B_i - A_i}(|p_i| + b_i) + occ\right) \;=\; O\left(2^{B-A}(m + B) + occ\right) .$$

   The index is a wildcard tree and by the same argument as for Theorem 3, it can be stored using space $O(nG^{k+o})$.

In case the received query pattern p has length $m + B > G$ we query the first index. It follows that $2^{B-A}\sigma^B \log\log n \leq 2^{B-A}G < 2^{B-A}(m+B)$, so the query time is $O(2^{B-A}(m+B) + occ)$. If p has length $m + B \leq G$ all occurrences of p in t can be found by querying the second index in time $O(2^{B-A}(m + B) + occ)$. The space of the index is

$$O(n + nG^{k+o}) = O(n(\sigma^{k+o} \log\log n)^{k+o}) = O(n\sigma^{(k+o)^2} \log^{k+o} \log n) .$$

This concludes the proof of Theorem 6.

8

CONCLUSION

In this thesis we considered the problem of indexing a string to report the occurrences of a query pattern containing wildcards. The ideal index has size linear in the length of the indexed string and optimal query time, i.e., the time to report the occurrences of a pattern is linear in the length of the query pattern and the number of occurrences.

The previous work on the problem indicate that solutions have query times which are either exponential in the number of wildcards in the pattern, or linear in the length of the indexed string. It has been an open problem if non-trivial solutions with optimal query times exist. We settled this question by giving an optimal time index obtained using a black-box reduction. However, this comes at the price of increasing the size of the index to be exponential in the maximal number of wildcards allowed in the pattern. It is noteworthy that the size of the optimal time index is automatically improved by using an index with a faster query time in the black-box reduction.

It has also been unknown if it is possible to obtain linear size indexes which avoid a query time linear in the length of the indexed string. We described a linear size index having a query time exponential in the number of wildcards. The base of the exponentiation is the size of the alphabet. Furthermore, we showed how to reduce the base of the exponentiation in the query time, but doing so increased the size of the index to be exponential in the maximal number of wildcards allowed in the pattern. This is the result of creating dedicated subtrees to handle the wildcards. However, examples indicate that these subtrees may be very similar, and it could be of interest to investigate if compression techniques can be applied to reduce the size of the index.

Moreover, we showed that our indexes can solve a generalized version of the problem where the pattern contains variable length gaps or optional wildcards. However, the price of doing so is that the query times become exponential in the number of optional wildcards in the pattern. It might be interesting to investigate if an optimal time index for this problem can be obtained.

The Longest Common Prefix data structure is a key component in our solutions. We gave a detailed explanation and new proofs of the data structure. Additionally, we showed two new properties that allow the data structure to be used in a more general context. We also introduced a generalization of the classic heavy path decomposition. This new

technique might be of independent interest for problems that can be solved efficiently on trees with bounded out-degree.

Other future work could investigate if our techniques can be applied when wildcards or variable length gaps are allowed in the indexed string. Another question is how the problem and the techniques for solving it relates to approximate text indexing. For wildcard indexes having a query time sublinear in the length of the indexed text, it remains an open problem if there is an index where neither the size nor the query time is exponential in the number of wildcards in the pattern.

Hjalte Wedel Vildhøj Søren Vind

BIBLIOGRAPHY

[1] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory Comput. Systems*, 37(3):441–456, 2004.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th FOCS*, pages 534–543, 1998.

[3] A. Amir, G. Landau, M. Lewenstein, and D. Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2), 2007.

[4] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. In *Proc. 11th SODA*, pages 794–803, 2000.

[5] P. Bille and I. L. Gørtz. Substring Range Reporting. In *Proc. 22nd CPM*, pages 299–308, 2011.

[6] P. Bille, I. L. Gørtz, H. Vildhøj, and D. Wind. String matching with variable length gaps. In *Proc. 17th SPIRE*, pages 385–394, 2010.

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

[8] H. L. Chan, T. W. Lam, W. K. Sung, S. L. Tam, and S. S. Wong. A linear size index for approximate pattern matching. *J. Disc. Algorithms*, 2011. To appear, announced at CPM 2006.

[9] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.

[10] G. Chen, X. Wu, X. Zhu, A. Arslan, and Y. He. Efficient string matching with wildcards and length constraints. *Knowl. Inf. Sys.*, 10(4):399–419, 2006.

[11] P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007.

[12] L. Coelho and A. Oliveira. Dotted suffix trees a structure for approximate text indexing. In *Proc. 13th SPIRE*, pages 329–336, 2006.

[13] R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th STOC*, pages 91–100, 2004.

[14] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. In *Proc. 9th SODA*, pages 463–472, 1998.

[15] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. 34rd STOC*, pages 592–601, 2002.

[16] M. J. Fischer and M. S. Paterson. String-Matching and Other Products. In *Complexity of Computation, SIAM-AMS Proceedings*, pages 113–125, 1974.

[17] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a Sparse Table with O(1) Worst Case Access Time. *J. ACM*, 31:538–544, 1984.

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

[19] K. Fredriksson and S. Grabowski. Nested counters in bit-parallel string matching. *Proc. 3rd LATA*, pages 338–349, 2009.

[20] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *ACM SIGACT News*, 17(4):52–54, 1986.

[21] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[22] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

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

[24] C. S. Iliopoulos and M. S. Rahman. Pattern matching algorithms with don't cares. In *Proc. 33rd SOFSEM*, pages 116–126, 2007.

[25] A. Kalai. Efficient pattern-matching with don't cares. In *Proc. 13th SODA*, pages 655–656, 2002.

[26] T. W. Lam, W. K. Sung, S. L. Tam, and S. M. Yiu. Space efficient indexes for string matching with don't cares. In *Proc. 18th ISAAC*, pages 846–857, 2007.

[27] G. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoret. Comput. Sci.*, 43:239–249, 1986.

[28] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.

[29] M. Maas and J. Nowak. Text indexing with errors. *J. Disc. Algorithms*, 5(4):662–681, 2007.

[30] G. Mehldau and G. Myers. A system for pattern matching applications on biosequences. *CABIOS*, 9(3):299–314, 1993.

[31] E. Myers. Approximate matching of network expressions with spacers. *J. Comput. Bio.*, 3(1):33–51, 1996.

[32] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.

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

[34] S. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *Proc. 37th FOCS*, pages 320–328, 1996.

[35] A. Tam, E. Wu, T. Lam, and S. Yiu. Succinct text indexing with wildcards. In *Proc. 16th SPIRE*, pages 39–50, 2009.

[36] D. Tsur. Fast index for approximate string matching. *J. Disc. Algorithms*, 8(4):339–345, 2010.

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

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

[39] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Process. Lett.*, 17(2):81 – 84, 1983.

Appendices

A

SUMMARY OF NOTATION AND DEFINITIONS

Strings and Sets	
Σ	The alphabet.
σ	The size of the alphabet, i.e., $\sigma = \lvert\Sigma\rvert$.
c	A character in the alphabet Σ.
t	The text string to build a wildcard index for. The characters in t are numbered from 1 to $\lvert t\rvert$.
n	The length of t, i.e., $n = \lvert t\rvert$.
x, y, z	Strings of lengths $\lvert x\rvert, \lvert y\rvert$ and $\lvert z\rvert$, respectively.
$x[a, b]$	The substring of x from index a to b, both inclusive.
$\mathrm{pref}_i(x)$	The prefix $x[1, i]$ of x of length i.
$\mathrm{suff}_i(x)$	The suffix $x[i, \lvert x\rvert]$ of x of length $n - i + 1$.
$\mathrm{pref}(x)$	The set of all non-empty prefixes of x.
$\mathrm{suff}(x)$	The set of all non-empty suffixes of x.
$\mathrm{maxpref}(x, y)$	The string z of maximum length where z is a prefix of both x and y.
S	A set of strings, i.e., $S \subset \Sigma^*$.
C	The set of suffixes of t, i.e., $C = \mathrm{suff}(t)$.
$C', C_1, \ldots, C_q$	Sets of substrings of t.
$\mathrm{pref}_i(S)$	The set obtained by taking the prefix $\mathrm{pref}_i(x)$ for each string $x \in S$.
$\mathrm{suff}_i(S)$	The set obtained by taking the suffix $\mathrm{suff}_i(x)$ for each string $x \in S$.
$\mathrm{pref}(S)$	The union over the set of all non-empty prefixes for all strings in S.
$\mathrm{suff}(S)$	The union over the set of all non-empty suffixes for all strings in S.
$\mathrm{maxpref}(S, x)$	The longest string z such that $z = \mathrm{maxpref}(x, y)$ for any string $y \in S$.

Patterns	
p	A query pattern.
$p_1, \ldots, p_j$	Subpatterns of p consisting of consecutive characters.
k	Maximum number of $*$ characters allowed in p in a bounded wildcard index.
j	The number of wildcards in p.

m	The number of characters in p.
occ	The number of occurrences of p in t
$occ(p_i, t)$	The number of occurrences of subpattern p_i in t.

Trees and Tries	
T	A rooted tree.
ℓ	A location in a compressed trie (an explicit vertex or a position on an edge). Also used to denote the string starting in the root and ending in ℓ.
u, v	Explicit vertices in a tree.
e	An edge in a tree.
$T(S)$	A compressed trie over the string set S.
$T_\ell(S)$	The subtrie of $T(S)$ rooted in ℓ.
$T(C), T(\text{suff}(t))$	The suffix tree for t.
$\text{root}(T)$	The root vertex of T.
$\text{height}(T)$	The maximum number of edges on a path from $\text{root}(T)$ to a leaf.
$\text{weight}(v)$	The number of leaves in the subtree of T rooted in v.
$\text{weight}(e)$	The length of the label on edge $e \in T(S)$.
$\text{depth}(\ell)$	The sum of edge weights on the path from $\text{root}(T(S))$ to $\ell \in T(S)$, i.e., the length of the string ℓ)

The LCP Data Structure	
$\mathcal{H}$	A heavy path decomposition for a tree T.
H	A heavy path in a heavy path decomposition $\mathcal{H}$ for a tree T.
$\text{root}(H)$	The root of a heavy path.
$\text{leaf}(H)$	The leaf of a heavy path.
$\text{pos}(x)$	The start position of a suffix x of t.
$\text{order}(x)$	The position of x in the lexicographic ordering of all substrings of t.
$\text{orderset}(C_i)$	The set of $\text{order}(x)$ for all strings $x \in C_i$.
$h_S(x)$	The maximum distance that x follows a string in S, i.e., $h_S(x) = \lvert\text{maxpref}(S, x)\rvert$.
$\text{lps}_S(x)$	A string $y \in C$ having $\lvert\text{maxpref}(x, y)\rvert = h_S(x)$ and being lexicographically closest to x among all strings in S.
$\text{leftleaf}(v)$	The left leaf in the subtree of T rooted at $v \in T$.
$\text{rightleaf}(v)$	The right leaf in the subtree of T rooted at $v \in T$.

Variable Length Gaps	
$*\{a_i, b_i\}$	A variable length gap, corresponding to a_i wildcards followed by $b_i - a_i$ optional wildcards.
A_i	The sum of the lower bounds in the variable length gaps in the pattern preceding subpattern p_i.
B_i	The sum of the upper bounds in the variable length gaps in the pattern preceding subpattern p_i.

A	The sum of the lower bounds a_i for all j variable length gaps in the pattern (the minimum number of wildcards in p), i.e., $A = A_j$.
B	The sum of the upper bounds b_i for all j variable length gaps in the pattern (the maximum number of wildcards), i.e., $B = B_j$.
o	The maximum number of optional wildcards allowed in the pattern for a bounded optional wildcard index.

Construction Parameters	
α	Maximum out-degree of a heavy tree in a heavy α-tree decomposition.
β	Maximum branching factor for a search in a wildcard tree.
χ	The maximum number of leaves in a bottom tree.

Integers	
h	The distance a string follows another string. Also used to denote the height of a tree.
i, j, r	Indices.

Queries	
$\text{NCA}(u, v)$	The nearest common ancestor of the two vertices $u, v \in T$.
$\text{WA}(v, i)$	The ancestor location ℓ of v having $\text{depth}(\ell) = i$.
$\text{PRED}_U(i)$	The predecessor for i in the set U.
$\text{SUCC}_U(i)$	The successor for i in the set U.
$\text{LCP}(x, i, \ell)$	The location reached from $\ell \in T(C_i)$ by matching the string x.

Algorithms	
extract($\mathcal{A}$)	Remove and return a single element from the set $\mathcal{A}$.
trie(ℓ)	The index i of the compressed trie $T(C_i)$ containing ℓ.
bottoms(ℓ)	The roots of the bottom tries $B_1, \ldots, B_q$ joined to location ℓ.
hasbottom(ℓ, c)	True if ℓ has a bottom trie reachable by an edge labeled c.
bottom(ℓ, c)	The root of the bottom trie reachable from ℓ by an edge labeled c.
nextlocation(ℓ, c)	The location ℓ' reachable from ℓ by an edge labeled c where $\|\ell'\| = \|\ell\| + 1$.
nextlocations(ℓ)	The set of child locations ℓ' of ℓ where $\|\ell'\| = \|\ell\| + 1$.
nextheavylocations(ℓ)	The set of child locations ℓ' of ℓ where $\|\ell'\| = \|\ell\| + 1$ reachable on a heavy path from ℓ.

B

THE PROOF OF LEMMA 4 BY COLE ET AL.

Lemma 4 in the paper by Cole et al. [13] states that the size of the structure supporting k wildcards in the pattern is $O(n + x\frac{(k+\log x)^k}{k!})$, where the term n comes from storing the suffix tree. Using the terminology from this thesis, the lemma states that the size of the wildcard tree $T^k_\beta(C)$ for $\beta = 2$ is $O\left(|C|\frac{(k+\log |C|)^k}{k!}\right)$. Cole et al. sketch a proof by induction in their paper. In the following we fill in the details of the proof, and highlight a potential problem.

The Proof

The proof can be divided into the following two claims.

Claim 1: Let $S_k(x)$ denote the space taken by the structure for k wildcards on a collection of x strings, then

$$S_k(x) \leq x + x\frac{S_{k-1}(\frac{x}{2})}{\frac{x}{2}} + x\frac{S_{k-1}(\frac{x}{4})}{\frac{x}{4}} + \cdots + x\frac{S_{k-1}(1)}{1} \quad \text{for} \quad k \geq 1$$

and $S_0(x) = x$.

Claim 2: If Claim 1 is true then by induction, one can show that

$$\begin{aligned} S_k(x) &\leq x + x\frac{\log x}{1!} + x\frac{\log x(\log x + 1)}{2!} + \cdots + x\frac{\log x(\log x + 1)\cdots(\log x + k - 1)}{k!} \\ &= O\left(x\frac{\log x(1 + \log x)\cdots(\log x + k - 1)}{k!}\right) = O\left(x\frac{(k + \log x)^k}{k!}\right) \end{aligned}$$

We investigate each of the claims separately.

Claim 1

In their paper Cole et al. prove this claim as follows. Consider a compressed trie $T(C)$ over the string set C containing $|C| = x$ strings. Let H be the heavy path starting from the root of $T(C)$. The wildcard trees hanging from the vertices on H each contain at most $x/2$ strings, so one of these trees has size at most $S_{k-1}(x/2)$. Hence the cost per string stored in a wildcard tree hanging from H is at most $\frac{S_{k-1}(x/2)}{x/2}$, assuming that $S_k(x)/x$ is non-decreasing function of x. At most x strings are stored in the wildcard trees hanging from H, since they constitute a partition of the strings in C. Consequently, the combined size of the wildcard trees hanging from H is at most $x\frac{S_{k-1}(x/2)}{x/2}$. Now consider the heavy paths hanging off H. Each wildcard tree joined to a vertex on one of these paths contains at most $x/4$ strings, and in total at most x strings are stored in the wildcard trees hanging from these paths. Therefore, the size of these wildcard trees is at most $x\frac{S_{k-1}(x/4)}{x/4}$. Repeating the argument for each of the $\log x$ layers of heavy paths leads to the bound in Claim 1, where the initial term x is the strings stored in $T(C)$.

Claim 2

Cole et al. do not prove this claim in their paper, but remark that it is easy to verify by induction. We give the proof here. For the base case $k = 0$, the claim is that $S_0(x) \leq x$, which is true since $S_0(x) = x$. For the inductive step, we assume that Claim 2 holds for $k = j$, and show that it also holds for $k = j + 1$. From Claim 1 we have that

$$S_{j+1}(x) \leq \quad x \quad + \overbrace{\quad x\frac{S_j(\frac{x}{2})}{\frac{x}{2}} \quad + \quad x\frac{S_j(\frac{x}{4})}{\frac{x}{4}} \quad + \quad \cdots \quad + \quad x\frac{S_j(1)}{1} \quad}^{\log x \text{ terms}}$$

Using the induction hypothesis on the $\log x$ terms involving $S_j(\cdot)$ yields

$$\begin{aligned}
S_{j+1}(x) \leq \quad & x \quad + \\
\text{(term 1)} \quad & x\left(1 + \frac{\log(\frac{x}{2})}{1!} + \frac{\log(\frac{x}{2})(\log(\frac{x}{2})+1)}{2!} + \cdots + \frac{\log(\frac{x}{2})(\log(\frac{x}{2})+1)\cdots(\log(\frac{x}{2})+j-1)}{j!}\right) \quad + \\
\text{(term 2)} \quad & x\left(1 + \frac{\log(\frac{x}{4})}{1!} + \frac{\log(\frac{x}{4})(\log(\frac{x}{4})+1)}{2!} + \cdots + \frac{\log(\frac{x}{4})(\log(\frac{x}{4})+1)\cdots(\log(\frac{x}{4})+j-1)}{j!}\right) \quad + \\
& \vdots \\
\text{(term } \log(x)-1) \quad & x\left(1 + \frac{\log(2)}{1!} + \frac{\log(2)(\log(2)+1)}{2!} + \cdots + \frac{\log(2)(\log(2)+1)\cdots(\log(2)+j-1)}{j!}\right) \quad + \\
\text{(term } \log x) \quad & x\left(1 + \frac{\log(1)}{1!} + \frac{\log(1)(\log(1)+1)}{2!} + \cdots + \frac{\log(1)(\log(1)+1)\cdots(\log(1)+j-1)}{j!}\right)
\end{aligned}$$

Adding the $\log x$ terms in each of the $j+1$ columns yields

$$\begin{aligned}
\text{Column 1:} \quad & \sum_{i=1}^{\log x} 1 \;=\; \log x \;=\; \frac{\log x}{1!} \\
\text{Column 2:} \quad & \sum_{i=1}^{\log x} \frac{\log(x)-i}{1!} = \sum_{i=1}^{\log(x)-1} \frac{i}{1!} \overset{\star}{=} \frac{(\log(x)-1)\log x}{2!} \leq \frac{\log x(\log x+1)}{2!} \\
& \vdots \\
\text{Column } j+1: \quad & \sum_{i=1}^{\log x} \frac{(\log(x)-i)(\log(x)-i+1)\cdots(\log(x)-i+j-1)}{j!} = \\
& \sum_{i=1}^{\log(x)-1} \frac{i(i+1)\cdots(i+j-1)}{j!} \overset{\star}{=} \frac{(\log(x)-1)\log(x)(\log(x)+1)\cdots(\log(x)+j-1)}{(j+1)!} \\
& \leq \frac{\log(x)(\log(x)+1)(\log(x)+2)\cdots(\log(x)+j)}{(j+1)!}
\end{aligned}$$

where $\star$ follows from the fact that

$$\sum_{i=1}^{\ell} \frac{i(i+1)\cdots(i+j-1)}{j!} = \frac{\ell(\ell+1)\cdots(\ell+j)}{(j+1)!} \ .$$

Adding the upper bounds for the $j+1$ columns, we have that

$$S_{j+1} \leq x + x\left(\frac{\log x}{1!} + \frac{\log x(\log x+1)}{2!} + \cdots + \frac{\log x(\log x+1)\cdots(\log x+j)}{(j+1)!}\right) \ ,$$

and that concludes the inductive step.

A Potential Problem

There is a problem in the proof of Claim 1. Recall that a wildcard tree joined to a vertex v on a heavy path H consists of the merge of the off-path substrees (i.e., the subtrees reached by following a light edge from v), where the first character has been replaced by $*$. A single off-path subtree hanging from a vertex v on the first heavy path H contains at most $x/2$ strings. However, it is not true that the wildcard tree hanging from v contains at most $x/2$ strings. In the worst case the wildcard tree could be the merge of $\sigma-1$ subtrees each containing at most $x/2$ strings, only excluding the heaviest subtree. Thus, the wildcard tree hanging from v could store $(1-1/\sigma)x$ strings.