String Indexing for Patterns with Wildcards

Philip Bille[1], Inge Li Gørtz[1], Hjalte Wedel Vildhøj[1], and Søren Vind

Technical University of Denmark, DTU Informatics, {phbi,ilg,hwvi}@imm.dtu.dk

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 by Lam et al. [ISAAC 2007], which requires query time $\Theta(jn)$ in the worst case.
- An index with 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, generalizing the index by Cole et al. [STOC 2004].

Our results are obtained using a novel combination of well-known and new techniques, which could be of independent interest.

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 [31] combined with perfect hashing [15] 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 [11,21,29,28,6,25,26]. In this paper, 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 [18,5] 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 to be the number of characters and wildcards in p, respectively. Consequently, the length of p is $m + j$.

Previous Work Exact string matching has been generalized with error bounds in many different ways. In particular, allowing matches within a bounded hamming or edit distance, known as approximate string matching, has been subject to much research [22,23,27,12,10,29,6,25,11,26,16,2]. Another generalization was suggested by Fischer and Paterson [14], 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 [14,13,9,20,8,4]. Some solutions to the indexing problem consider the case where wildcards appear only in the indexed string [28] or in both the string and the pattern [11,21].

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. [11] 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 [31], 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 $O(m + j + occ)$.

In 2004, Cole et al. [11] gave an elegant k-bounded wildcard index using $O(n \log^k n)$ space and 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 are a new data structure for *longest common prefix (LCP) queries* and a *heavy path decomposition* [17] 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. [11] show how to reduce the size of the index by only creating additional wildcard tries for the off-path subtries. This leads to the $O(n \log^k n)$ space bound. Secondly, using the new tries, the top-down search branches at most twice for each wildcard, leading to the 2^j term in the query time. Though Cole et al. [11] did not consider unbounded wildcard indexes, the technique can be extended to this case by using only the LCP data structure. This leads to an unbounded wildcard index with query time $O(m + \sigma^j \log\log n + occ)$ using space $O(n \log n)$.

The solutions described by Cole et al. [11] all have bounds, which are exponential in the number of wildcards in the pattern. Very recently, Lewenstein [24] used similar techniques to improve the bounds to be exponential in the number of *gaps* in the pattern (a gap is a maximal substring of consecutive wildcards). Assuming that the pattern contains at most g gaps each of size at most G, Lewenstein obtains a bounded index with query time $O(m + 2^\gamma \log\log n + occ)$ using space $O(n(G^2 \log n)^g)$, where $\gamma \leq g$ is the number of gaps in the pattern.

A different approach was taken by Iliopoulos and Rahman [19], who describe an unbounded wildcard index using linear space. For a pattern p consisting of strings $p_0, p_1, \ldots, p_j$ (subpatterns) interleaved by j wildcards, the query time of the index is $O(m + \sum_{i=0}^{j} occ(p_i, t))$, where $occ(p_i, t)$ denotes the number of matches of p_i in t. This was later improved by Lam et al. [21] with an index that determines complete matches by first identifying potential matches of the subpatterns in 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.

The unbounded wildcard index by Iliopoulos and Rahman [19] was the first index to achieve query time linear in m while using $O(n)$ space. Recently, Chan et al. [6] considered the related problem of obtaining a k-mismatch index supporting queries in time linear in m and using $O(n)$ space. They describe an index with a query time of $O(m + (\log n)^{k(k+1)} \log\log n + occ)$. However, this bound assumes a constant-size alphabet and a constant number of errors. In this paper we make no assumptions on the size of these parameters.

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. [11], 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. [21], 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 [1] 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. [11] 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 using $O(n\log(n)\log_\beta^{k-1} n)$ space. The index can report the occurrences in t of a pattern with m characters and $j \leq k$ wildcards in time $O(m + \beta^j \log\log n + occ)$.*

The theorem provides a time-space trade-off for k-bounded wildcard indexes. Compared to the index by Cole et al. [11], 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 [17]. This decomposition could be of independent interest. We also show that for $\beta = 1$ the same technique yields an index with 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 linear time index with a non-trivial space bound. The result improves upon the space usage of the simple linear 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 a linear 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 closely related to the concept of *filtering search* introduced by Chazelle [7] and has previously been applied for indexing problems [3,6]. The theorem follows from applying the black-box reduction to the index of Theorem 1.

Our three indexes also support searching for query patterns with *variable length gaps*, i.e., patterns of the form $p = p_0 *\{a_1, b_1\} p_1 *\{a_2, b_2\} \ldots *\{a_j, b_j\} p_j$, where $*\{a_i, b_i\}$ denotes a variable length gap that matches an arbitrary substring of length between a_i and b_i, both inclusive. We will not consider variable length gaps in this paper.

We have left out some proofs due to lack of space, and we refer the reader to [30], which also treats query patterns with variable length gaps.

2 Preliminaries

We introduce the following notation. 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. 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 $\text{pref}_i(t) = t[1, i]$ and $\text{suff}_i(t) = t[i, n]$ denote the prefix and suffix of t of length i and $n - i + 1$, respectively. Omitting the subscripts, we let $\text{pref}(t)$ and $\text{suff}(t)$ denote the set of all non-empty prefixes and suffixes of t, respectively. We extend the definitions of prefix and suffix to sets of strings $S \subseteq \Sigma^*$ as follows.

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

A set of strings S 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.

Trees and Tries For a tree T, the root is denoted $\text{root}(T)$, while $\text{height}(T)$ is the number of edges on a longest path from $\text{root}(T)$ 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 of 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.

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 $\text{pref}(S)$. The prefix $x \in \text{pref}(S)$ corresponding to a location $\ell \in T(S)$ is obtained by concatenating the edge labels on the path from $\text{root}(T(S))$ 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 denotes the compressed trie over all suffixes of t, i.e., $T(\text{suff}(t))$. 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

$$S_\ell = \left\{ \text{suff}_{|\ell|}(x) \mid x \in S \wedge \text{pref}_{|\ell|}(x) = \ell \right\} .$$

Heavy Path Decomposition For a vertex v in a rooted tree T, we define $\text{weight}(v)$ to be the number of leaves in T_v, where T_v denotes the subtree rooted at v. We define $\text{weight}(T) = \text{weight}(\text{root}(T))$. The *heavy path decomposition* of T, introduced by Harel and Tarjan [17], 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, $O(\log(\text{weight}(T)))$ 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.

3 The LCP Data Structure

Cole et al. [11] introduced the the *Longest Common Prefix (LCP) data structure*, which 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 and show a new property that is essential for obtaining Theorem 1.

The LCP data structure stores a collection of compressed tries $T(C_1), T(C_2), \ldots, T(C_q)$ over the string sets $C_1, C_2, \ldots, C_q \subset \Sigma^*$. Each C_i is a set of substrings of the indexed string t. The purpose of the LCP data structure is to support LCP queries

$\textsc{lcp}(x, i, \ell)$: 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*. 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 $C = \text{suff}(t)$. The following lemma is implicit in the paper by Cole et al. [11].

Lemma 1 (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 $O(\log\log|C|)$ using space $O(|C| + \sum_{i=1}^{q}|C_i|)$. Unrooted LCP queries on $T(C_i)$ can be performed in time $O(\log\log|C|)$ using $O(|C_i|\log|C_i|)$ additional space.*

We extend the LCP data structure by showing that support for slower unrooted LCP queries on a compressed trie $T(C_i)$ can be added using linear additional space.

Lemma 2. *Unrooted LCP queries on $T(C_i)$ can be performed in time $O(\log|C_i| + \log\log|C|)$ using $O(|C_i|)$ additional space.*

Proof. We initially create a heavy path decomposition for all compressed tries $T(C_1), \ldots, T(C_q)$. The search path for x starting in ℓ traverses a number of heavy paths in $T(C_i)$. Intuitively, an unrooted LCP query can be answered by following the $O(\log|C_i|)$ heavy paths that the search path passes through. For each heavy path, the next heavy path can be identified in constant time. 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 follows H. Cole et al. [11] showed that h can be determined in constant time by

performing nearest common ancestor queries on $T(C)$. To answer $\textsc{lcp}(x, i, \ell)$ we identify the heavy path H of $T(C_i)$ that ℓ is part of and compute the distance h as described by Cole et al. 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 $\textsc{lcp}(\text{suff}_{h+2}(x), i, \ell')$. 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 $\textsc{lcp}(x, i, \ell)$. 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 $O(\log \log |C|)$. Consequently, an unrooted LCP query can be answered in time $O(\log |C_i| + \log \log |C|)$ using $O(|C_i|)$ additional space to store the predecessor data structures for each heavy path. □

4 An Unbounded Wildcard Index Using Linear Space

In this section we show how to obtain 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.

ART Decomposition The ART decomposition introduced by Alstrup et al. [1] decomposes a tree into a single *top tree* and a number of *bottom trees*. The construction is 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 3 (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.*

Obtaining the Index Applying an ART decomposition on $T(\text{suff}(t))$ with $\chi = \log n$, 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 3, 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 in an LCP data structure, noting that these compressed tries only contain substrings of t. Using Lemma 2, 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 1 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. Since the branching factor is not reduced, $O(\sigma^i)$ LCP queries, each taking time $O(\log\log n)$, are performed for the subpattern p_i. This concludes the proof of Theorem 1.

5 A Time-Space Trade-Off for k-Bounded Wildcard Indexes

In this section we will show Theorem 2. We first introduce the necessary constructions.

Heavy α-Tree Decomposition The *heavy α-tree decomposition* is a generalization of the well-known heavy path decomposition introduced by Harel and Tarjan [17]. The purpose is to decompose a rooted tree T into a number of *heavy trees* joined by light edges, such that a path 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 then constructed by classifying edges in T as being heavy or light according to the following 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 $\text{lightdepth}_\alpha(v)$ 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 4. *For any vertex v in a rooted tree T and $\alpha > 0$*

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

Lemma 4 holds for any heavy α-tree decomposition obtained using a proper weight function on T. In the remaining part of the paper we will assume that the weight of a vertex is the number of leaves in the subtree rooted at v.

We define $\text{lightheight}_\alpha(T)$ to be the maximum light depth of a vertex in T, and remark that $\text{lightheight}_0(T) = \text{height}(T)$. 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 the subtries $T_\ell(S)$ where ℓ is the first location on a light outgoing edge of v, i.e., $|\ell| = |v| + 1$.

Wildcard Trees We introduce the (β, k)-*wildcard tree*, denoted $T^k_\beta(C')$, where $1 \leq \beta < \sigma$ is a chosen parameter. This data structure stores a collection of strings $C' \subset \Sigma^+$ in a compressed trie such 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 with wildcard height i by an edge labeled $*$. This tree is searched if a wildcard is consumed in v. Formally, $T^k_\beta(C')$ 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 v to the root of $T^{i-1}_\beta(\text{suff}_2(\text{lightstrings}(v))$ by an edge labeled $*$. Let $T^0_\beta(S) = T(S)$.

Wildcard Tree Index Given a collection C' of strings 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. Consequently, 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 a query time $O(\beta^j m + occ)$. For $\beta = 1$ the query time is $O(m + j + occ)$.

Lemma 5. *For any integer $1 \le \beta < \sigma$, the wildcard tree $T^k_\beta(C')$ has query time $O(\beta^j m + j + occ)$. 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.*

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 $|T^i_\beta(S)| \le |S|\sum_{j=0}^{i} H^j$. 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,

$$\left|T^{i+1}_\beta(S)\right| \;=\; |S| + \sum_{v\in T(S)} \left|T^i_\beta(S_v)\right| \;\overset{IH}{\le}\; |S| + \sum_{v\in T(S)} |S_v| \sum_{j=0}^{i} H^j \;.$$

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 $T(S)$. S is also a set of suffixes of C', and hence H is an upper bound on the maximum light depth of $T(S)$. This establishes that $\sum_{v\in T(S)} |S_v| \;\le\; |S|H$, thus showing that $|T^{i+1}_\beta(S)| \le |S| + |S|H\sum_{j=0}^{i} H^j = |S|\sum_{j=0}^{i+1} H^j$. ◻

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

Lemma 6. *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 $O\left(\beta^j m + occ\right)$.*

Wildcard Tree Index Using the LCP Data Structure The wildcard index of Lemma 6 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 a 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 Cole et al. [11]. 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. Upon receiving the query pattern $p = p_1 * p_2 * \ldots * p_k$, each p_i is preprocessed in time $O(|p_i|)$ to support LCP queries for any suffix of p_i. The search for p proceeds as described for the normal wildcard tree, except now rooted and unrooted LCP queries are used to search for suffixes of $p_0, p_1, \ldots, p_k$.

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 $O(n \log^k_\beta n)$ for $T^k_\beta(C)$ 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, i.e., $O(n \log^k_\beta n)$. 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 $\sum_{i=0}^{q} |S_i| = |T^{k-1}_\beta(C)|$, by Lemma 1, the additional space required to support unrooted LCP queries is

$$O\big(\sum_{i=0}^{q} |S_i| \log |S_i|\big) = O\big(\log n \sum_{i=0}^{q} |S_i|\big) = O\left(\log n |T^{k-1}_\beta(C')|\right) = O\big(n \log(n) \log^{k-1}_\beta n\big)\,,$$

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. [11] is obtained as a special case of Theorem 2.

Corollary 1 (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 Linear Query Time

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

Lemma 7. *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 $O(m + j + occ)$.*

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 8. *Let $F \geq m$ and let G be independent of m and j. Given a wildcard index $\mathcal{A}$ with query time $O(F + G + occ)$ and space usage S, there is a k-bounded wildcard index $\mathcal{B}$ with query time $O(F + j + occ)$ and taking space $O(n \min(G, h)^k + S)$, where h is the height of the suffix tree for t.*

Proof. The wildcard index $\mathcal{B}$ consists of $\mathcal{A}$ as well as a special wildcard index $T_1^k(\text{pref}_G(\text{suff}(t)))$ $\mathcal{C}$, which is a wildcard tree with $\beta = 1$ over the set of all substrings of t of length G. G can be used as an upper bound for the light height in Lemma 5, so the space required to store $\mathcal{C}$ is $O(n \min(G, h)^k)$ by using Lemma 7 if $G > h$. A query on $\mathcal{B}$ results in a query on either $\mathcal{A}$ or $\mathcal{C}$. In case $G < F+j$, we query $\mathcal{A}$ and the query time will be $O(F+G+occ) = O(F+j+occ)$. In case $G \geq F + j$, we query $\mathcal{C}$ with query time $O(m+j+occ) = O(F+j+occ)$. In any case the query time of $\mathcal{B}$ is $O(F + j + occ)$. □

Applying Lemma 8 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 linear query time using space $O(\sigma^{k^2} n \log^k \log n)$. This concludes the proof of Theorem 3.

References

1. S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th FOCS*, pages 534–543, 1998.
2. A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. In *Proc. 11th SODA*, pages 794–803, 2000.
3. P. Bille and I. L. Gørtz. Substring Range Reporting. In *Proc. 22nd CPM*, pages 299–308, 2011.
4. P. Bille, I. L. Gørtz, H. Vildhøj, and D. Wind. String matching with variable length gaps. In *Proc. 17th SPIRE*, pages 385–394, 2010.
5. 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.

6. 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*, 9(4):358–364, 2011.
7. B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.
8. 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.
9. P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007.
10. L. Coelho and A. Oliveira. Dotted suffix trees a structure for approximate text indexing. In *Proc. 13th SPIRE*, pages 329–336, 2006.
11. 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.
12. R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. In *Proc. 9th SODA*, pages 463–472, 1998.
13. R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. 34rd STOC*, pages 592–601, 2002.
14. M. J. Fischer and M. S. Paterson. String-Matching and Other Products. In *Complexity of Computation, SIAM-AMS Proceedings*, pages 113–125, 1974.
15. 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.
16. Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *ACM SIGACT News*, 17(4):52–54, 1986.
17. D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
18. K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The PROSITE database, its status in 1999. *Nucleic Acids Res.*, 27(1):215–219, 1999.
19. C. S. Iliopoulos and M. S. Rahman. Pattern matching algorithms with don't cares. In *Proc. 33rd SOFSEM*, pages 116–126, 2007.
20. A. Kalai. Efficient pattern-matching with don't cares. In *Proc. 13th SODA*, pages 655–656, 2002.
21. 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.
22. G. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoret. Comput. Sci.*, 43:239–249, 1986.
23. G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.
24. M. Lewenstein. Indexing with gaps. In *Proc. 18th SPIRE*, pages 135–143, 2011.
25. M. Maas and J. Nowak. Text indexing with errors. *J. Disc. Algorithms*, 5(4):662–681, 2007.
26. 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.
27. S. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *Proc. 37th FOCS*, pages 320–328, 1996.
28. A. Tam, E. Wu, T. Lam, and S. Yiu. Succinct text indexing with wildcards. In *Proc. 16th SPIRE*, pages 39–50, 2009.
29. D. Tsur. Fast index for approximate string matching. *J. Disc. Algorithms*, 8(4):339–345, 2010.
30. H. W. Vildhøj and S. Vind. String Indexing for Patterns with Wildcards. *Master's thesis, Technical University of Denmark*, 2011. `http://www.imm.dtu.dk/~hwvi/`.
31. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th SWAT*, pages 1–11, 1973.