Space-Efficient String Indexing for Wildcard Pattern Matching

Moshe Lewenstein[1], Yakov Nekrich[*2], and Jeffrey Scott Vitter[2]

1   Department of Computer Science, Bar-Ilan University
2   Department of Electrical Engineering and Computer Science, University of Kansas

───── **Abstract** ─────

In this paper we describe compressed indexes that support pattern matching queries for strings with wildcards. For a constant size alphabet our data structure uses $O(n \log^\varepsilon n)$ bits for any $\varepsilon > 0$ and reports all occ occurrences of a wildcard string in $O(m + \sigma^g \cdot \mu(n) + \mathrm{occ})$ time, where $\mu(n) = o(\log \log \log n)$, σ is the alphabet size, m is the number of alphabet symbols and g is the number of wildcard symbols in the query string. We also present an $O(n)$-bit index with $O((m + \sigma^g + \mathrm{occ}) \log^\varepsilon n)$ query time and an $O(n(\log \log n)^2)$-bit index with $O((m + \sigma^g + \mathrm{occ}) \log \log n)$ query time. These are the first non-trivial data structures for this problem that need $o(n \log n)$ bits of space.

1   Introduction

In the string indexing problem, we pre-process a source string T, so that all occurrences of a query string P in T can be reported. This is one of the most fundamental data structure problems. While handbook data structures, suffix arrays and suffix trees, can answer string matching queries efficiently, they store the source string T in $\Theta(\log n)$ bits of space per symbol. In situations when massive amounts of data must be indexed, the space usage can become an issue. Compressed indexes that use $o(\log n)$ or even H_0 bits per symbol, where H_0 denotes the zero-order entropy, were studied extensively. We refer the reader to [12] for a survey of results on compressed indexing.

In many scenarios we are interested in reporting all occurrences of strings that resemble the query string $\tilde{P}$ but do not have to be identical to $\tilde{P}$. The problem of approximate pattern matching is important for biological applications and information retrieval and has received considerable attention [3, 9, 14, 19, 1, 2]. In this paper we consider a variant of the approximate pattern matching when the query string $\tilde{P}$ may contain wildcards (don't care symbols), and the wildcard symbol matches any alphabet symbol.

The standard indexing data structures can be used to answer wildcard pattern matching queries. A pattern $\tilde{P}$ with g wildcard symbols matches σ^g different patterns, where σ denotes the size of the alphabet. We can generate all patterns that match $\tilde{P}$ and report all occ occurrences of these patterns (and hence all occurrence of $\tilde{P}$) in $O(m \cdot \sigma^g + \mathrm{occ})$ time, where m is the number of alphabet symbols. If the maximal number of wildcards in a query is bounded by k (k-bounded indexing), we can store a compressed trie with all possible combinations of k wildcard symbols for every suffix. Then a query can be answered in $O(|\tilde{P}| + \mathrm{occ})$ time, but the total space usage is $O(n^{k+1})$ words of $\Theta(\log n)$ bits.

Cole *et al.* [3] presented an elegant data structure for k-bounded indexing. Their solution needs $O(n \log^k n)$ words of space and answers wildcard queries in $O(m + 2^g \log \log n + \mathrm{occ})$ time. Very recently this has been improved in [10] to $O(n \log^{k+\varepsilon} n)$ bits of space with the

Ref.	Space Usage	Query Time
[3]	$O(n \log n)$ words	$O(m + \sigma^g \log \log n + \text{occ})$
[1]	$O(n)$ words	$O(m + \sigma^g \log \log n + \text{occ})$
New	$O(n \log^\varepsilon n)$ bits	$O(m + \sigma^g \sqrt{\log^{(3)} n} + \text{occ})$
New	$O(n(\log \log n)^2)$ bits	$O((m + \sigma^g + \text{occ}) \log \log n)$
New	$O(n)$ bits	$O((m + \sigma^g + \text{occ}) \log^\varepsilon n)$

■ **Table 1** Previous and new results on unbounded wildcard indexing; m and g denote the number of alphabet symbols and wildcards in the query pattern.

same query time as Cole *et al.* [3]. Bille *et al.* [1] obtained another trade-off: for any pre-defined k and β, their k-bounded index uses $O(n \log n \log_\beta^{k-1} n)$ words and answers queries in $O(m + \beta^g \log \log n + \text{occ})$ time. These indexes can provide fast answers to wildcard queries when the number of wildcards is small. However the space usage of the above data structures is high even when k is a constant. For super-constant values of k (for instance, when the maximal number of wildcards is bounded by $\log \log n$) the cost of storing the data structure may become prohibitive.

Another line of research is the design of data structures that use linear or almost-linear space and support queries with an arbitrarily large number of wildcards. Cole *et al.* [3] describe a data structure that uses $O(n \log n)$ words and answers queries in $O(m + \sigma^g \log \log n + \text{occ})$ time. Iliopoulos and Rahman [14] and Lam *et al.* [9] describe linear-space indexes; however, their data structures need $\Theta(n)$ worst-case time to answer a query. Recently, Bille *et al.* [1] described an $O(n)$-words data structure that answers queries in $O(m + \sigma^g \log \log n + \text{occ})$ time.

When the amount of stored data is very large, even linear space usage can be undesirable. While numerous compressed indexes for exact pattern matching are known, there are no previously described data structures for wildcard indexing that use $o(n \log n)$ bits. In this paper we present sublinear space indexes for wildcard pattern matching. Our results are especially conspicuous when the alphabet size is constant. Our first data structure uses $O(n \log^\varepsilon n)$ bits and reports occurrences of a wildcard pattern in $O(m + \sigma^g \sqrt{\log^{(3)} n} + \text{occ})$ time[1]; henceforth ε denotes an arbitrarily small positive constant. Thus we improve both the space usage and the query time of the previous best data structure [1]. The space usage can be further decreased at cost of slightly increasing the query time. We describe two indexes that use $O(n)$ and $O(n(\log \log n)^2)$ bits of space; queries are supported in $O((m + \sigma^g + \text{occ}) \log^\varepsilon n)$ and $O((m + \sigma^g + \text{occ}) \log \log n)$ time respectively. Previous and new results with worst-case efficient query times are listed in Table 1.

In this paper we assume, unless specified otherwise, that the alphabet size is a constant. But our techniques are also relevant for the case when the alphabet size is arbitrarily large. We can obtain an $O(n \log \sigma)$-bit data structure that answers queries in $O((m + \sigma^g + \text{occ}) \log_\sigma^\varepsilon n)$ time. We can also obtain an $O(n \log n)$-bit data structure that supports queries in $O(m + \sigma^g + \text{occ})$ time if $\sigma \geq \log \log n$. Other interesting trade-offs are possible and will be described in the full version of this paper.

In Section 2, we recall some results related to compressed suffix trees and suffix arrays and compressed data structures for a set of integers. We also define the unrooted LCP queries, introduced in Cole *et al.* [3], that are the main tool in all currently known efficient structures

---

[1] $\log^{(3)} n = \log \log \log n$.

for wildcard indexing. In Section 3 we describe data structures that answer unrooted LCP queries on a small subtree of the suffix tree. Our data structures need only a small number of additional bits if the (compressed) suffix tree and suffix array of the source text are available. In Section 4, we describe compact data structures that answer LCP queries and wildcard pattern matching queries on an arbitrarily large suffix tree. These data structures are based on a subdivision of suffix tree nodes into small subtrees. In Sections 5, 9, and 7 we show how we can speed-up the data structures from [3], [1] and retain $o(n \log n)$ space usage. The main component of our improvement is a method for processing batches of unrooted LCP queries. In previous works [3, 1] LCP queries were answered one-by-one.

2   Preliminaries

Unrooted LCP Queries. In this paper $s_1 \circ s_2$ denotes the concatenation of strings s_1 and s_2 and $\mathcal{T}$ denotes the suffix tree of the source text. A string $str(v, u)$ is obtained by concatenating labels of all edges on the path from v to u and $str(u) = str(v_r, u)$ for the root node v_r of $\mathcal{T}$. A *location* on a suffix tree $\mathcal{T}$ is an arbitrary position on an edge of $\mathcal{T}$; a location on an edge (v, u) can be uniquely identified by specifying the edge (u, v) and the offset from the upper node of (u, v). We can straightforwardly extend the definitions of $str(\tilde{v}, \tilde{u})$ and $str(\tilde{u})$ to arbitrary locations $\tilde{u}$ and $\tilde{v}$. The unrooted LCP query (v, P), defined in [3], asks for the lowest descendant location $\tilde{u}$ of a node v, such that $str(v, \tilde{u})$ is a prefix of a string P. Thus an unrooted LCP query provides the answer to the following question: if we were to search for a pattern P in a subtree with root v, where would the search end? While we can obviously answer this question in $O(|P|)$ time by traversing the trie starting at v, faster solutions are also possible.

As in the previous works [3, 1], we consider the following two-stage scenario for answering queries: during the first stage an arbitrary string P is pre-processed in $O(|P|)$ time; during the second stage, we answer queries (u, P_j) for any suffix P_j of P and any $u \in \mathcal{T}$. Cole *et al.* [3] described an $O(n \log^2 n)$-bit data structure that answers unrooted LCP queries in $O(\log \log n)$ time. Bille *et al.* [1] improved the space usage to linear ($O(n \log n)$ bits).

Compressed Suffix Arrays and Suffix Trees. The suffix array SA for a text T contains starting positions of T's suffixes sorted in lexicographic order: $SA[i] = k$ if the suffix $T[k..n]$ is the k-th smallest suffix of the text T. We will say that i is the rank of the suffix $T[k..n]$. An inverse suffix array stores information about lexicographic order of suffixes: $SA^{-1}[k] = i$ iff $SA[i] = k$. We will say that a data structure provides a suffix array functionality in time t_{SA} if it enables us to compute $SA[i]$ and $SA^{-1}[k]$ for any $1 \leq i, k \leq n$ in $O(t_{SA})$ time. A number of compressed data structures provide suffix array functionality in little time.

▶ **Lemma 1.** *If the alphabet size $\sigma = O(1)$, the following trade-offs for space usage $s(n)$ and t_{SA} are possible: (a) $s(n) = O((1/\varepsilon)n)$ and $t_{SA}(n) = O(\log^\varepsilon n)$, or (b) $s(n) = O(n \log \log n)$ and $t_{SA}(n) = O(\log \log n)$, or (c) $s(n) = O(n \log^\varepsilon n)$ and $t_{SA}(n) = O(1)$ for any constant $\varepsilon > 0$*

Proof: Result (a) is shown in [17] and results (b), (c) are from [15] □

If $SA[t] = f$ the function $\Psi^i(t)$ computes the position of the suffix $T[f + i..n]$ in the suffix array. This function can be computed in $O(t_{SA})$ time as $SA^{-1}[SA[t] + i]$. Let the string depth of a node $v \in \mathcal{T}$ be the length $str(v)$. If the suffix array functionality is available, we

can store the suffix tree in $O(n)$ additional bits, so that the string depth of any node v can be computed in $O(t_{SA})$ time [18, 4, 16].

Using $O(n)$ additional bits, we can process a string P in $O(|P|t_{SA})$ time and find for any suffix $P^j = P[j..|P|]$ of P: (i) the rank r_j of P^j in T and (ii) the longest common prefix (LCP) of P^j and the suffixes $SA[r_j]$, $SA[r_j + 1]$ of T. Using McCreight's procedure for inserting a new string into a generalized suffix tree, we find the locations where suffixes of P must be inserted into $\mathcal{T}$: first we traverse the suffix tree starting at the root and find the location corresponding to $P[1..|P|]$ in the suffix tree; then we find locations of $P[2..|P|]$, ..., $P[|P| - 1..|P|]$, $P[|P|]$ by following the suffix links. Next, we compute the string depths of these locations. The total time needed to find the locations and their depths in a compressed suffix tree is $O(|P|t_{SA})$. When the rank r_j of P^j and LCPs of P^j and its neighbors are known, we can use this information to compute the LCP of P^j and any suffix $SA[q]$ in $O(t_{SA})$ time: if $q < r_j$, $LCP(P^j, SA[q])$ is the minimum of $LCP(P^j, SA[r_j])$ and $LCP(SA[r_j], SA[q])$; the case $q > r_j$ is symmetric. Sadakane [18] showed how to compute $LCP(SA[r_j], SA[q])$ in $O(t_{SA})$ time. Hence, we can compute the LCP for any two suffixes of P and T in $O(t_{SA})$ time after $O(|P|t_{SA})$ pre-processing time.

Heavy Path Decomposition. Let $\mathcal{T}$ be an arbitrary tree. We can decompose $\mathcal{T}$ into disjoint root-to-leaf paths, called *heavy paths*. If an internal node $u \in \mathcal{T}$ is on a heavy path p, then its heaviest child u_i (that is, the child with the greatest number of leaf descendants) is also on p. If the child u_j of u is not on p, then u has at least twice as many leaf descendants as u. Therefore the heavy-path decomposition of $\mathcal{T}$ guarantees that *any* root-to-leaf path in $\mathcal{T}$ intersects with at most $\log n$ heavy paths; we refer to [8] for details.

Searching in a Small Set. We can search in a set with a poly-logarithmic number of elements using the data structure called an atomic heap [5]. An atomic heap on a set of integers S, $|S| = \log^{O(1)} n$, uses linear space and enables us to find for any integer q the largest $e \in S$ such that $e \le q$ (respectively, the smallest $e \in S$ such that $e \ge q$) in $O(1)$ time. Using the result of Grossi *et al.* [6], we can search in a small set using small additional space and only one access to elements of S.

▶ **Lemma 2** ([6], Lemma 3.3). *Suppose that $|S| = \log^{O(1)} n$ and $e \le n$ for any $e \in S$. There exists a data structure D that uses $O(|S| \log \log n)$ additional bits and answers predecessor and successor queries on S in $O(1)$ time. When a query is answered, only one element $e' \in S$ needs to be accessed.*

3 Unrooted LCP Queries on Small Sets

In this section we describe compact data structures that answer LCP queries on a small set of suffixes. We consider a set S that contains a poly-logarithmic number of consecutive suffixes from the suffix array of S. Our data structure supports queries of the form (u_0, P) where $u_0 \in \mathcal{T}_0$ and $\mathcal{T}_0$ is a subtree of the suffix tree $\mathcal{T}$ induced by suffixes from S; the query answer is the lowest location $\tilde{v} \in \mathcal{T}_0$ below $\tilde{u}$, such that $str(u_0, \tilde{v}_0)$ is a prefix of P. These data structures are an important building block of data structures that will be constructed in the following sections and a key to space-saving solution: we will show in section 4 how a suffix tree can be divided into small subtrees. In this section we show how unrooted LCP queries can be supported on such small subtrees. The main idea is to keep the (ranks of) suffixes in succinct predecessor data structures that need $O(\log \log n)$ additional bits per element; we do not have to store the ranks in these data structures because they can be retrieved in $O(t_{SA})$ time using the (compressed) suffix tree and the (compressed) suffix array. Thus we can answer unrooted LCP queries on $\mathcal{T}_0$ using $O((\log \log n)^2)$ bits per suffix. We

assume in the rest of this section that S contains $f = O(\log^3 n)$ consecutive suffixes and $\mathcal{T}_0$ is a subtree of the suffix tree induced by suffixes from S.

▶ **Lemma 3.** *There exists a data structure that uses $O(f(\log\log n)^2)$ additional bits of space and answers unrooted LCP queries on $\mathcal{T}_0$ in $O(1)$ time. We assume that our data structure can access the suffix tree of T, the suffix array of T, the inverse suffix array of T, and a universal look-up table of size $O(n^g)$ for an arbitrarily small positive constant g.*

Proof: Let $\mathcal{T}_0$ denote the part of the suffix tree induced by suffixes in S. We apply the heavy path decomposition to nodes of $\mathcal{T}_0$. Let $S(u)$ denote the set that contains all strings $str(w, v_l)$ for the parent w of u and all leaf descendants v_l of u. We remark that all elements of $S(u)$ are suffixes of T. The global rank of a suffix Suf is its position in the suffix array of T. Let $R(u)$ denote the set of global ranks of all suffixes in $S(u)$. For every node $u \in \mathcal{T}_0$ and each of its children u_i that are not on the same heavy path as u, we store a data structure $D(u_i)$. $D(u_i)$ answers predecessor queries on $R(u_i)$. It is not necessary to store the set $R(u)$ itself: an arbitrary element of $R(u)$ can be accessed using the functionality provided by the suffix array. Suppose that the global rank of the suffix corresponding to $str(w, v_p)$, where v_p is the p-th leaf descendant of $S(u)$, should be computed. Since we can access the suffix tree, we can find the rank r_1 of the suffix that ends in the leaf v_p. Then the suffix corresponding to $str(w, v_p)$ has rank $SA[SA^{-1}[r_1] + depth(w)]$ where $depth(w)$ is the string depth of the node w in the global suffix tree. By Lemma 2, $D(u_i)$ can be stored in $O(|S(u_i)|\log\log n)$ bits and answer predecessor queries in $O(1)$ time. The total number of elements in all $D(u)$ is $O(f\log f) = O(f\log\log n)$. Thus all $D(u)$ need $O(f(\log\log n)^2)$ bits or $o(f)$ words of $\log n$ bits. For every heavy path h_j on $\mathcal{T}_0$ we keep a data structure H_j that contains the depths of all nodes. H_j is also implemented as described in Lemma 2 and uses $O(\log\log n)$ bits per node.

The search for an LCP in $\mathcal{T}_0$ is organized in the same way as in [3]. To answer a query (u, P_j), $u \in \mathcal{T}_0$, we start by finding $l_0 = lcp(P_j, SA[r])$, where r is the rank of the suffix that starts at u and ends in the leaf v_h, such that u and v_h are on the same heavy path. Let u' denote the lowest node of depth $d_1 \leq depth(u) + l_0$ that is on the same heavy path h_0 in $\mathcal{T}_0$ as u. If $d_1 \neq depth(u) + l_0$, then u' is the answer to our query. If $d_1 = depth(u) + l_0$ and u' is a leaf, then again u' is the answer to our query. If $d_1 = depth(u) + l_0$ and u' is not a leaf, we identify the child u_j of u' that is labelled with $P_j[d_1 + 1]$. If such a child does not exist, then again u' is the answer. Otherwise, we find the rank r' of $P'_j = P_j[d_1 + 1..|P_j|]$. Using $D(u_j)$, we find the predecessor and the successor of r' in $S(u_j)$.

Let S_l and S_r denote the corresponding suffixes of $D(u_j)$. We can compute $l_l = lcp(P'_j, S_l)$ and $l_r = lcp(P'_j, S_r)$. Suppose that $l_l \geq l_r$. Let u_l be the node of depth at most $depth(u_j) + l_j$ on the path from u_j to the leaf l_l containing S_l. The node u_l, that can be found by answering an appropriate level ancestor query for l_l, is the answer to the original LCP query. The case when $l_r > l_l$ is handled in the same way. □

In the following two Lemmas we extend the result of Lemma 3 to the situation when the data structure is stored in compressed form. We assume that we can compute $SA[i]$, $SA^{-1}[i]$ for any i, $1 \leq i \leq n$, in $O(t_{SA})$ time; we also assume that compressed suffix tree with functionality described in Section 2 is available. Only additional bits necessary to support queries on $\mathcal{T}_0$ are counted.

▶ **Lemma 4.** *There exists a data structure that uses $O(f(\log\log n)^3)$ additional bits of space and answers unrooted LCP queries on $\mathcal{T}_0$ in $O(t_{SA})$ time. Our data structure uses a universal look-up table of size $O(n^g)$ for an arbitrarily small positive constant g.*

Proof: We use the same data structure as in the proof of Lemma 4, but $SA[SA^{-1}[r_1] + depth(w)]$ and $depth(u)$ are computed in $O(t_{SA})$ time. It is not necessary to store $\mathcal{T}$. Information about the heavy path decomposition of $\mathcal{T}_0$ can be stored in $O(f)$ bits. We show how this can be done in [11]. Data structures H_i need $O(\log \log n)$ bits per node. Since queries on H_j and $D(u)$ are answered in $O(t_{SA})$ time, an unrooted LCP query is also answered in $O(t_{SA})$ time. □

The following Lemma is proved in [11].

▶ **Lemma 5.** *There exists a data structure that uses $O(f)$ additional bits of space and answers unrooted LCP queries on $\mathcal{T}_0$ in $O((t_{SA}(\log \log \log n))$ time. Our data structure uses a universal look-up table of size $O(n^g)$ for an arbitrarily small positive constant g.*

4 Wildcard Pattern Queries in Less Space

Now we are ready to describe the compact data structure for wildcard indexing. Our approach is as follows. We divide the suffix tree $\mathcal{T}$ into subtrees, so that each subtree has a poly-logarithmic number of nodes and results of Section 3 can be applied to each subtree. We also keep a tree $\mathcal{T}_m$ that has one representative node for each subtree and stores information about positions of small subtrees in $\mathcal{T}$. Unrooted LCP queries are answered in two steps. First, we identify the small subtree that contains the answer using data structures on $\mathcal{T}_m$. Then we search in the small subtree using the data structure of Section 3. We select the size of subtrees so that $\mathcal{T}_m$ and data structures for $\mathcal{T}_m$ use $O(n)$ bits. A detailed description of our data structure is given below.

Data Structure. Let $\tau = \sigma \log^2 n$. We visit all leaves of the suffix tree $\mathcal{T}$ in left-to-right order and mark every τ-th leaf. We visit all internal nodes of $\mathcal{T}$ in bottom-to-top order and mark each node u such that at least two children of u have marked descendants. Finally the root node is also marked.

We divide the nodes of the suffix tree into groups as follows. Let u be a marked internal node, such that all its non-leaf descendants are unmarked. Each child u_i of u contains at most one marked leaf (because otherwise the subtree rooted at u_i would contain marked internal nodes). The subtrees rooted at children $u_i, \ldots, u_d$ of u are distributed among groups $G_j(u)$. We select indices $i_1 = 1, i_2, \ldots, i_t = m$ such that exactly one node among $u_{i_j}, \ldots, u_{i_{j+1}-1}$ has a marked leaf descendant. For each j, $1 \le j < t$, all nodes in the subtrees of $u_{i_j}, \ldots, u_{i_{j+1}-1}$ are assigned to group $G_j(u)$. Every $G_j(u)$ contains $O(\tau)$ nodes. Now suppose that a marked node u has marked descendants. We divide the children of u into groups $G(u, v)$ such that exactly one child u_i of u in each $G(u, v)$ has exactly one direct marked descendant. That is, in every $G(u, v)$ there is exactly one child u_i of u satisfying one of the following two conditions: (i) u_i is marked (in this case u_i is assigned to the group $G(u, u_i)$) or (ii) u_i has exactly one marked descendant v such that there are no other marked nodes between u_i and v. The group $G(u, v)$ also contains all nodes that are descendants of u_i but are not proper descendants of v. To make nodes of $G(u, v)$ a subtree, we also include u into $G(u, v)$. The number of nodes in $G(u, v)$ is also bounded by $O(\tau)$.

Each node $w \in \mathcal{T}$ belongs to some group $G_j(u)$ or $G(v, u)$. The total number of groups is $O(n/\tau)$ because each group can be associated with one marked node. Since every $G_j(u)$ is a subtree, we can answer unrooted LCP queries on the nodes (and locations) of $G_j(u)$ implemented according to Lemma 4. Furthermore we divide every $G(v, u)$ into two overlapping subgroups: $G_l(v, u)$ contains all nodes of $G(v, u)$ that are on the path from v to u or to the left of this path; $G_r(v, u)$ contains all nodes of $G(v, u)$ that are on the path from v to u or to

the right of this path. We also add the leftmost and rightmost leaf descendants of the node u, where u is the marked node in $G(v, u)$, to $G_l(v, u)$ and $G_r(v, u)$ respectively. The leaves in each group $G_l(v, u)$ and $G_r(v, u)$ correspond to τ consecutive suffixes. Therefore we can answer unrooted LCP queries on $G_l(u, v)$ and $G_r(u, v)$ using Lemmas 4 or 5. The answer to an unrooted LCP query on $G(u, v)$ can be obtained from answers to the same query on $G_l(u, v)$ and $G_r(u, v)$. The data structures for unrooted LCP queries on $G_j(u)$, $G_l(u, v)$ and $G_r(u, v)$ will be denoted $D_j(u)$, $D_l(u, v)$ and $D_r(u, v)$ respectively. Each node belongs to at most two groups; therefore all group data structures need $O(n)$ bits of space.

The nodes of the suffix tree are stored in compressed form described in Section 2. The depth and the string depth of any node can be computed in $O(t_{SA})$ time. We can also pre-process an arbitrary pattern in $O(|P|t_{SA})$ time, so that the LCP of any suffixes $P[j..|P|]$ and $T[i..n]$ can be found in $O(t_{SA})$ time.

Moreover, we keep all suffixes that are stored in marked leaves of the suffix tree in a compressed trie $\mathcal{T}_m$. Nodes of $\mathcal{T}_m$ correspond to marked nodes of $\mathcal{T}$. Unrooted LCP queries on $\mathcal{T}_m$ can be answered in $O(\log \log n)$ time using $O((n/\tau) \log^2 n) = O(n/\sigma)$ bits; see Lemma 11 in [11].

In every node of $\mathcal{T}_m$ we store a pointer to the corresponding marked node of $\mathcal{T}$. We also keep a bit vector B that keeps data about marked and unmarked nodes of $\mathcal{T}$; the order of nodes is determined by a pre-order traversal of $\mathcal{T}$. The i-th entry $B[i]$ is set to 1 if the i-th node (in pre-order traversal) is marked, otherwise $B[i]$ is set to 0. Using $o(n)$ additional bits, we can compute the number of preceding 1's for any position in B in $O(1)$ time [13]. Hence for any node $u \in \mathcal{T}$, we can find the number of marked nodes that precede u in the pre-order traversal of $\mathcal{T}$. We also store an array A_m; the i-th entry of A_m contains a pointer to the node of $\mathcal{T}_m$ that corresponds to the i-th marked node in $\mathcal{T}$. Using B and A_m, we can find the node of $\mathcal{T}_m$ that corresponds to a given marked node of $\mathcal{T}$ in $O(1)$ time. We will also need another data structure to facilitate the navigation between marked nodes and its children. For every marked node u with marked internal descendants and for all groups $G(u, v)$, we store the first character on the label of the edge from u to its leftmost child $u_i \in G(u, v)$ in a predecessor data structure.

Queries. Consider an unrooted LCP query (u, P). If u is marked, we find the lowest marked descendant u' of u, such that $str(u, u')$ is a prefix of P. We find the child u_i of u' such that the edge from u' to u_i is labelled with a string s_i and $str(u, u') \circ s_i$ is a prefix of P. Then we use the data structure $D_j(u)$ (respectively $D_l(u, w)$ and $D_r(u, w)$) for the subtree that contains u_i and answer an unrooted LCP query (u_i, P') for P' satisfying $str(u, u') \circ s_i \circ P' = P$. The answer to the latter query provides the answer to the original query (u, P). If u is unmarked, we start by answering the query (u, P) using the data structure for the group that contains u. If the answer is an unmarked node u_1 (or a location $\tilde{u}_1$ on an edge that starts in an unmarked node), then u_1 (respectively $\tilde{u}_1$) is the answer to our query. If u_1 is marked, we answer the query (u_1, P_1), where P_1 is the remaining suffix of P, as described above. Again we obtain the answer to the original query (u, P).

We can report all occurrences of $\tilde{P} = \phi P_1 \phi P_2 \ldots \phi P_d$ by answering at most σ^d unrooted LCP queries and σ^d accesses to the compressed suffix tree. For all alphabet symbols a we find the location of the pattern aP_1 by answering a wildcard LCP query. For each symbol a, such that the location $\tilde{u}_a$ of aP in $\mathcal{T}$ was found, we continue as follows. If $\tilde{u}_a$ is a position on an edge (u_a, u'_a), we check whether the remaining part of the edge label equals aP'_2 for some symbol a and a prefix P'_2 of P_2. If this is the case, we answer a query (u'_a, P''_2) where P''_2 satisfies $P_2 = P'_2 \circ P''_2$. If $\tilde{u}_a$ is a node, we find the loci of patterns $str(\tilde{u}_a) \circ xP_2$, where x denotes any alphabet symbol, as described above. We proceed in the same way until

the loci of all $x_1 P_1 \ldots x_m P_m$ for any alphabet symbol x_i are found. This approach can be straightforwardly extended to reporting occurrences of a general wildcard expression $\tilde{P} = \phi^{k_1} P_1 \phi^{k_2} P_2 \ldots \phi^{k_d} P_d$, where ϕ^{k_i} denotes an arbitrary sequence of k_i alphabet symbols and $k_i \geq 0$ for $1 \leq i \leq d$.

▶ **Theorem 6.** *There exists an $O(n + s_{small} n)$-bit data structure that reports all* occ *occurrences of a wildcard pattern $\phi^{k_1} P_1 \phi^{k_2} P_2 \ldots \phi^{k_d} P_d$ in $O(\sum_{i=1}^{d} |P_i| t_{SA} + \sigma^g t_{small}(n) + \text{occ} \cdot t_{SA})$ time, where $g = \sum_{i=1}^{m} k_i$; s_{small} and t_{small} denote the average space usage and query time of the data structures described in Lemmas 3 or 4.*

Two interesting corollaries of this result are the following indexes. We use the same notation as in Theorem 6. If we combine Lemma 1, (a) with Lemma 5 we get $t_{small} = O(\log^\varepsilon n)$ and $s_{small} = O(1)$ (the query time $O(\log^\varepsilon n \log^{(3)} n)$ can be simplified to $O(\log^\varepsilon n)$ by replacing ε with some $\varepsilon' < \varepsilon$). If we plug in this result into Theorem 6, we obtain our first main data structure.

▶ **Corollary 7.** *There exists an $O(n)$-bit data structure that answers wildcard pattern matching queries in $O((\sum_{i=1}^{d} |P_i| + \sigma^g + \text{occ}) \log^\varepsilon n)$ time.*

We remark that the result of Corollary 7 can be also extended to the case of an arbitrarily large alphabet. In this case the index uses $O(n \log \sigma)$ bits and queries are answered in $(\sum_{i=1}^{d} |P_i| + \sigma^g + \text{occ}) \log^\varepsilon_\sigma n)$ time. This variant can be obtained by using the suffix array of Grossi *et al.* [7]; the compressed suffix tree uses $O(n \log \sigma)$ bits in this case.

If we combine Lemma 1, (b) with Lemma 5 and plug in the result into Theorem 6, we obtain our second main data structure.

▶ **Corollary 8.** *There exists an $O(n(\log \log n)^2)$-bit data structure that answers wildcard pattern matching queries in $O((\sum_{i=1}^{d} |P_i| + \sigma^g + \text{occ}) \log \log n)$ time.*

5   LCP Queries for Patterns with Wildcards, $\sigma = \log \log n$

In the remaining part of this paper we describe faster solutions that use linear or sublinear space. In sections 5 and 6 we describe an $O(n \log n)$-bit data structure for $\sigma \geq \log \log n$. In section 7 we use a more technically involved variant of the same approach to obtain fast solutions for $\sigma < \log \log n$.

In this section we will show how to answer a batch of LCP queries called wildcard LCP queries. A wildcard LCP query $(u, \phi P)$ returns the loci of $str(u) \circ aP$ in the suffix tree of a source text T for all $a \in \Sigma$ such that $str(u) \circ aP$ occurs in T. As before, we assume that we can preprocess some pattern $\overline{P}$ in $O(\overline{P})$ time; then, queries (u, P) where P is a suffix of $\overline{P}$ are answered. The pre-processing is the same as in Section 3.

A leaf descendant v_l of a node u is a light descendant of u if v_l and u are not on the same heavy path. A wildcard tree $\mathcal{T}_u$ for a node u is a compressed trie that contains all strings s satisfying $a \circ s = str(u, v_l)$ for some symbol a and some light leaf descendant v_l of u. The main idea of our approach is to augment the suffix tree $\mathcal{T}$ with wilcard trees in order to accelerate the search. To avoid logarithmic increase in space usage, only selected nodes of wilcard trees will be stored. We explain our method for the case $\sigma = \log \log n$.

Let $\tau = \sigma \log^2 n$. We mark the nodes of the suffix tree in the same way as described in Section 4. Every τ-th leaf of $\mathcal{T}$, each internal node with at least two children that have marked descendants, and the root of $\mathcal{T}$ are marked. The nodes of $\mathcal{T}$ will be called the *alphabet nodes*. We also store selected nodes from wildcard trees, further called *wildcard nodes*. A truncated wildcard tree $\mathcal{T}_u$ is a compressed trie containing all strings s, such that $a \circ s = str(u, v_l)$ for

some marked light leaf descendant v_l of u. Each leaf-to-root path intersects $O(\log n)$ heavy paths. Therefore each marked leaf occurs in $O(\log n)$ truncated wildcard trees. Hence the total number of wildcard nodes is $O((n/\tau)\log n)$. Every node in each truncated wildcard tree contains pointers to some alphabet nodes or locations on edges between alphabet nodes. Suppose that a node v is in a wildcard subtree $\mathcal{T}_w$, the parent of $\mathcal{T}_w$ is some node w, and the label of v in $\mathcal{T}_w$ is s. For every symbol a such that $s_a = str(w) \circ a \circ s$ occurs in the source text, we store a pointer from u to the location u_a of s_a. The total number of pointers is equal to $O(n \log n(\sigma/\tau))$. We distribute alphabet nodes into groups $G_j(u)$ and $G(v, u)$ as described in Section 4; data structures $D_j(u)$, $D_l(v, u)$, and $D_r(v, u)$ are also defined in the same way as in Section 4. Every pointer from a wildcard node to an alphabet node w (or edge (u, w)) contains a reference to the group that contains w. Moreover, both alphabet and wildcard nodes of our extended suffix tree are kept in the data structure, described in Lemma 11 in [11], that answers unrooted LCP queries in $O(\log \log n)$ time.

Queries. Suppose that a wildcard LCP query $(u, \phi P)$ must be answered. Let a_h be the first symbol in $str(u, u_h)$, where u_h is the child of u that is on the same heavy path. We answer a query $a_h \circ P$ in $O(\log \log n)$ time using the result of [1]. Next, we must find the locus nodes of all patterns $a_j \circ P$, $a_j \neq a_h$. We answer an LCP query P in the truncated wildcard tree $\mathcal{T}_u$ of the node u. Let w denote the node where the search for P in $\mathcal{T}_u$ ends and let w_r denote the root node of $\mathcal{T}_u$. The node w can also be found in $O(\log \log n)$ time.

1. Suppose that $str(w_r, w) = P$. We follow pointers from w to alphabet nodes w_1, ..., w_σ marked with alphabet symbols $a_1,$..., a_σ. For each $1 \leq j \leq \sigma$ we find the group $G_r(u_j)$ (or $G(u_j, v_j)$) that contains w_j and answer an LCP query (w_j, P_j) on the tree induced by $G(u_j)$ (respectively $G(u_j, v_j)$). The string P_j is a suffix of P that satisfies $str(u, u_j) \circ P_j = a_j \circ P$. Using information in the pointer from w to u_j, we can find P_j in $O(1)$ time.

2. The pattern P can be also located between two nodes w' and w of $\mathcal{T}_u$ such that $str(w_r, w')$ is prefix of P and P is a prefix of $str(w_r, w)$. For every j, we follow the pointers marked with alphabet symbol a_j. Suppose that pointers from w' and w lead to locations $\tilde{w}'_j$ and $\tilde{w}_j$ respectively. Let w'_j be the lower node on the edge of $\tilde{w}'_j$ and let w_j be the upper node on the edge of $\tilde{w}_j$. There are no marked nodes between w'_j and w_j. Therefore we only need to search in the group that contains w_j to complete the LCP query.

The total search time is $O(\log \log n + \sigma \cdot t_{\text{small}})$ where t_{small} is the time needed to answer an LCP query on a subtree of τ nodes. We use Lemma 3; hence $t_{\text{small}} = O(1)$. Since $\sigma = \log \log n$, a wildcard LCP query is answered in $O(\log \log n) = O(\sigma)$ time.

6　Wildcard Pattern Matching Queries for $\sigma \geq \log \log n$

Wildcard LCP Queries. We can modify the data structure of Section 5 for the case when the alphabet size $\sigma \geq \log \log n$. We divide the alphabet Σ into groups such that every group, except the last one, contains $\log \log n$ elements. The last group contains at most $\log \log n$ elements. We will denote these groups Σ^1, ..., Σ^g for $g = \lceil \sigma/\log \log n \rceil$. Instead of one wildcard tree $\mathcal{T}_u$, we will store g modified wildcard trees $\mathcal{T}_u^1, \ldots, \mathcal{T}_u^g$ in every node $u \in \mathcal{T}$. A wildcard tree $\mathcal{T}_u^i$ for a node u is a compressed trie that contains all strings s satisfying $a \circ s = str(u, v_l)$ for some symbol $a \in \Sigma^i$ and some marked light leaf descendant v_l of u. We keep the same data structure for every $\mathcal{T}_u^i$ as in Section 5. Thus we answer LCP queries for each group of $\log \log n$ alphabet symbols in $O(\log \log n)$ time. The total time needed to answer a wildcard LCP query is $O(\lceil \sigma/\log \log n \rceil \log \log n) = O(\sigma)$.

Indexing. Consider a query $\tilde{P} = \phi P_1 \phi P_2 \ldots \phi P_d$. If $\sigma \geq \log \log n$, then our data

structure for wildcard LCP queries enables us to find all occurrences of $\tilde{P}$ by answering wildcard LCP queries. We find the loci of all $a_i P_1$ for every $a_i P_1$ that occurs in the source text T. This is achieved by answering a wildcard LCP query $(u_r, \phi P_1)$. For every found location u_i^1 we proceed as follows. If u_i^1 is in a middle of an edge e, we move one symbol down and then check whether the remaining symbols of an e are labelled with a prefix of P_2. If this is the case and the remaining part of e is labelled with P_2', we answer a regular LCP query (w_i^1, P_2'') such that w_i^1 is the node at the lower end of e and $P_2 = P_2' \circ P_2''$. Using the data structure of Bille *et al.* [1], an LCP query can be answered in $O(\log \log n)$ time. If u_i^1 is a node in the suffix tree, then we answer a wildcard LCP query $(u_i^1, \phi P_2)$. We continue in the same manner until the loci of all $x P_1 \ldots x P_m$, where x denotes an arbitrary symbol in Σ, are found. A general wildcard pattern $\phi^{k_1} P_1 \ldots \phi^{k_d} P_d$ is processed in the same way.

Since the maximum number of wildcard LCP queries and standard LCP queries does not exceed σ^g, the total query time is $O(\sigma^g)$. Preprocessing stage for all wildcard LCP queries takes $O(\Sigma_{i=1}^d |P_i|)$ time.

▶ **Lemma 9.** *Suppose that the alphabet size $\sigma \geq \log \log n$. Using an $O(n \log n)$-bit data structure, we can report all occurrences of a pattern $\tilde{P} = \phi^{k_1} P_1 \phi^{k_2} P_2 \ldots \phi^{k_d} P_d$ in $O(\sum_{i=1}^d |P_i| + \sigma^g + \text{occ})$ time, where* occ *is the number of times $\tilde{P}$ occurs in the text and $g = \sum_{i=1}^d k_i$.*

7 Wildcard Pattern Matching Queries for Small Alphabets

In this section we consider the case when the alphabet size $\sigma < \log \log n$. We use the approach of Sections 5 and 6, but the notion of wildcard LCP queries is generalized. A t-wildcard LCP query $(u, \tilde{P})$ for a wildcard string $\tilde{P} = \phi^{k_1} P_1 \phi^{k_2} P_2 \ldots \phi^{k_d} P_d$ such that $\sum k_i = t$, finds locations of all patterns $str(u) \circ P$, where $P = s_1 s_2 \ldots s_{k_1} P_1 s_{k_1+1} \ldots s_{k_2} P_2 \ldots s_{t-1} s_t P_d$ and $s_i, 1 \leq i \leq t$, are arbitrary alphabet symbols, in the suffix tree. A 1-wildcard LCP query, used in the previous sections, takes $O(\log \log n)$ time and can replace up to σ standard wildcard queries. Hence, when the alphabet size σ is small, we cannot achieve noteworthy speed-up in this way. A t-wildcard LCP query can replace up to σ^t regular LCP queries and lead to more significant speed-up even when σ is very small. We will use iterated wildcard subtrees in order to support s-wildcard LCP queries efficiently. Our construction consists of two parts. We mark selected nodes in the suffix tree $\mathcal{T}$ and divide it into subtrees $\mathcal{T}_i$ of size $O(\tau_1)$; we keep a data structure that supports t_1-wildcard LCP queries on the subtree $\mathcal{T}^m$ induced by marked nodes of $\mathcal{T}$. We also mark selected nodes, further called secondary marked nodes, in each subtree $\mathcal{T}_i$ and divide $\mathcal{T}_i$ into $\mathcal{T}_{i,j}$ of size $O(\tau_2)$. Let $\mathcal{T}_i^m$ be the subtree induced by secondary marked node of $\mathcal{T}_i$; we keep a data structure that answers standard wildcard LCP queries on $\mathcal{T}_i^m$. Details of our data structure and parameter values can be found below.

Trees $\mathcal{T}_i$ and $\mathcal{T}^m$. Let $t_1 = \log_{\sigma/2} \log \log n$ and $\tau_1 = \sigma^{t_1} \log^{t_1+1} n$. We use the same scheme as in Section 4 to mark every τ_1-th leaf and selected internal nodes, so that the suffix tree $\mathcal{T}$ is divided into subtrees $\mathcal{T}_i$ of size $O(\tau_1)$ and the number of marked nodes is $O(n/\tau_1)$. Trees $\mathcal{T}_i$ correspond to groups $G_j(u)$ and $G(u, v)$ defined in section 4.

Let $\mathcal{T}^m$ be the tree induced by marked nodes. We iteratively augment $\mathcal{T}^m$ with wildcard subtrees. For any marked internal node u, the (level-1) wildcard subtree $\mathcal{T}_u$ is a compressed trie containing all strings s, such that $a \circ s = str(u, v_l)$ for some marked light leaf descendant v_l of u. We also keep a level-$(i+1)$ wildcard subtree $\mathcal{T}_w$ for every node w in a level-i wildcard subtree $\mathcal{T}_u$. $\mathcal{T}_w$ contains all strings s such that $a \circ s = str(u, v_l)$ for some alphabet symbol a and a light leaf descendants v_l of w. We construct level-i wildcard subtrees for $1 \leq i \leq t_1$. The parameter t_1 is chosen in such way that $\sigma^{t_1} = 2^{t_1} \log \log n$ and $t = \log_\sigma \log \log n$. Every

node in all level-i wildcard trees has pointers to the corresponding locations in the alphabet tree $\mathcal{T}$. Each pointer also contains information about the subtree $\mathcal{T}_i$

The total number of nodes and pointers in wildcard subtrees is $(n/\tau_1)\sigma^{t_1}\log^{t_1} n$. Level-$t$ wildcard subtrees can be used to answer unrooted t-wildcard LCP queries on $\mathcal{T}_m$ in $O(2^t\log\log n)$ time; our method is quite similar to the procedure for answering wildcard queries in [3]. Consider a query $(\tilde{u}, \tilde{P})$, where $\tilde{u}$ is a location in the alphabet tree or in some i-wilcard subtree. We distinguish between the following four cases. (i) If $\tilde{u}$ is on a tree edge and the next symbol is a wildcard, we simply move down by one symbol along that edge. (ii) Suppose that $\tilde{u}$ is on a tree edge e and the next symbols are a string P_n of alphabet symbols. Let l denote the string label of the part of e below $\tilde{u}$, $l = str(\tilde{u}, u')$ where u' is the lower node on e. We compute $o = LCP(P_n, l)$. and move down by $\min(|l|, o)$ symbols along e. (iii) If $\tilde{u}$ is a node and the next unprocessed symbol in $\tilde{P}$ is a wildcard, our procedure branches and visits two locations: we move down by one symbol along the edge to the heavy child of $\tilde{u}$ and visit the root of the wildcard tree $\mathcal{T}_{\tilde{u}}$ (if $\tilde{u}$ is on a level-i wildcard tree, we visit the root of the $(i+1)$-subtree $\mathcal{T}_{\tilde{u}}$). (iv) If $\tilde{u}$ is a node and the next symbols are a string P_n of alphabet symbols, we answer a standard LCP query $(\tilde{u}, P_n)$. The procedure is finished when we cannot move down from any location that is currently visited. The number of branching points is 2^t and we answer 2^t standard LCP queries. We need $O(\sigma^t)$ time to return from locations in wildcard trees to the corresponding locations in the alphabet tree. Thus the total time is $O(2^t\log\log n + \sigma^t) = O(\sigma^t)$. When the search in $\mathcal{T}^m$ is completed we can continue searching in subtrees $\mathcal{T}_j$. Data structures for subtrees $\mathcal{T}_i$ are described in [11], where we show that an unrooted LCP query on $\mathcal{T}_i$ can be answered in $O((\log^{(3)} n)^{1/2})$ time.

Wildcard String Matching. It follows from the above description that we can answer t_1-wildcard LCP queries in $O(\sigma^{t_1}\sqrt{\log^{(3)} n})$ time. Consider now an arbitrary pattern $\tilde{P} = \phi^{k_1}P_1\phi^{k_2}P_2\ldots\phi^{k_d}P_d$. We divide it into chunks $\tilde{P}[1]$, $\tilde{P}[2]$, ..., $\tilde{P}[r]$, such that each chunk $\tilde{P}[i]$, $i \geq 2$, contains exactly t_1 wildcard symbols. The chunk $P[1]$ contains $v \leq t_1$ wildcard symbols.

We start at the root and find locations of all $\tilde{P}[1] = \phi^{k_1}P_1\ldots\phi^{k_f}P_f\phi^r$ where $r \leq k_{f+1}$. If $\sum_{i=1}^{f}|P_f| > (\log\log n)\cdot\sigma^t$, we answer at most σ^t standard LCP queries in $O(\sigma^t\log\log n) = O(\sum_{i=1}^{f}|P_i|)$ time. If $\sum_{i=1}^{f}|P_i| \leq (\log\log n)\cdot\sigma^t$, then the total length of $\tilde{P}[1]$ is at most $\ell = (\log\log n)\cdot\sigma^t + t$. Since $\sigma < \log\log n$, there are $O((\log\log n)^{\ell})$ different patterns and each of this patterns fits into one machine word. Hence, all string patterns P_s that match $\tilde{P}[1]$ can be generated in $O(\sigma^v)$ time. We keep a look-up table with locations of all strings P, such that $|P| \leq \ell$ in $\mathcal{T}$. Using this table we find locations of all P_s that match $\tilde{P}[1]$ and occur in the source text. For every such location $\tilde{u}$, we answer queries $(\tilde{u}_1, \tilde{P}[2])$, $(\tilde{u}_2, \tilde{P}[3])$, ..., where $\tilde{u}_1 = \tilde{u}$ and $\tilde{u}_i$ for $i > 1$ is an answer to some query $(\tilde{u}_{i-1}, \tilde{P}[i])$. It is easy to show that the total query time is $O(\sum_{i=1}^{d}|P_i| + \sigma^g\sqrt{\log^{(3)} n} + \mathrm{occ})$.

▶ **Theorem 10.** *If the alphabet size $\sigma = O(1)$ and $\sigma > 2$, then there exists an $O(n\log^{\varepsilon} n)$-bit data structure that reports all occ occurrences of a wildcard pattern $\phi^{k_1}P_1\phi^{k_2}P_2\ldots\phi^{k_d}P_d$ in $O(\sum_{i=1}^{d}|P_i| + \sigma^g\sqrt{\log^{(3)} n} + \mathrm{occ})$ time.*

We remark that the same query time as in Theorem 10 can be also achieved for a non-constant σ; the space usage would grow to $O(n\log n)$ bits, however. To obtain this result, we would need to use standard (uncompressed) suffix tree and suffix array for the source data.

References

1 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. In *Proc. 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2012)*, pages 283–294, 2012.

2 Ho-Leung Chan, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. A linear size index for approximate pattern matching. *J. Discrete Algorithms*, 9(4):358–364, 2011.

3 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th Annual ACM Symposium on Theory of Computing (STOC 2004)*, pages 91–100, 2004.

4 Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.

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

6 Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009)*, pages 517–528, 2009.

7 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

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

9 Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Siu-Ming Yiu. Space efficient indexes for string matching with don't cares. In *Proc. 18th International Symposium on Algorithms and Computation (ISAAC 2007)*, pages 846–857, 2007.

10 Moshe Lewenstein, J. Ian Munro, Venkatesh Raman, and Sharma V. Thankachan. Less space: Indexing for queries with wildcards. In *Proc. 24th International Symposium on Algorithms and Computation (ISAAC 2013)*, pages 89–99, 2013.

11 Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. Space-efficient string indexing for wildcard pattern matching. *CoRR*, abs/1401.0625, 2014.

12 Veli Mäkinen and Gonzalo Navarro. Compressed text indexing. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

13 J. Ian Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1996)*, pages 37–42, 1996.

14 M. Sohel Rahman and Costas S. Iliopoulos. Pattern matching algorithms with don't cares. In *Proc. 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007)*, pages 116–126, 2007.

15 S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *Inf. Process. Lett.*, 82(6):307–311, 2002.

16 Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53, 2011.

17 Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th International Conference on Algorithms and Computation (ISAAC 2000)*, pages 410–421, 2000.

18 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

19 Alan Tam, Edward Wu, Tak Wah Lam, and Siu-Ming Yiu. Succinct text indexing with wildcards. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE 2009)*, pages 39–50, 2009.