Less Space: Indexing for Queries with Wildcards

Moshe Lewenstein[1,⋆], J. Ian Munro[2,⋆⋆], Venkatesh Raman[3,⋆⋆⋆], and Sharma V. Thankachan[4,†]

[1] Bar-Ilan University, Israel
moshe@macs.biu.ac.il
[2] University of Waterloo, Canada
imunro@uwaterloo.ca
[3] The Institute of Mathematical Sciences, India
vraman@imsc.res.in
[4] Louisiana State University, USA
thanks@csc.lsu.edu

Abstract. Text indexing is a fundamental problem in computer science, where the task is to index a given text (string) $T[1..n]$, such that whenever a pattern $P[1..p]$ comes as a query, we can efficiently report all those locations where P occurs as a substring of T. In this paper, we consider the case when P contains wildcard characters (which can match with any other character). The first non-trivial solution for the problem is given by Cole et al. [STOC 2004], where the index space is $O(n \log^k n)$ words or $O(n \log^{k+1} n)$ bits and the query time is $O(p + 2^h \log\log n + occ)$, where k is the maximum number of wildcard characters allowed in P, $h \leq k$ is the number of wildcard characters in P and occ represents the number of occurrences of P in T. Even though many indexes offering different space-time trade-offs were later proposed, a clear improvement on this result is still not known. In this paper, we first propose an $O(n \log^{k+\epsilon} n)$ bits index achieving the same query time as that of Cole et al.'s index, where $0 < \epsilon < 1$ is an arbitrary small constant. Then we propose another index of size $O(n \log^k n \log \sigma)$ bits, but with a slightly higher query time of $O(p + 2^h \log n + occ)$, where σ denotes the alphabet set size.

1 Introduction and Related Work

Text indexing is a fundamental problem in computer science, where the task is to index a given text (string) $T[1..n]$, such that whenever a pattern $P[1..p]$ comes as

⋆ The author is grateful for the support of the Binational Science Foundation (BSF) grant # 2010437 and for the support of the German Israel Foundation (GIF) grant # 1147/2011. This work was done while the author was on sabbatical at U. Waterloo, Canada.

⋆⋆ This research was funded in part by NSERC of Canada and the Canada Research Chairs program.

⋆⋆⋆ This work was done while the author was on sabbatical at U. Waterloo, Canada.

† Supported by National Science Foundation (NSF) Grants CCF–1017623 (R. Shah and J. S. Vitter) and CCF–1218904 (R. Shah). Part of this work was done while the author was visiting U. Waterloo, Canada.

a query, we can efficiently report all those locations where P occurs as a substring of T. The classic data structures for solving this problem are suffix trees [28] and suffix arrays [21]. Both these linear space ($O(n \log n)$ bits) structures can perform pattern matching in optimal $O(p + occ)$ and $O(p + \log n + occ)$ time respectively, where occ is the number of occurrences of P in T [1]. Approximate string matching and wildcard matching are natural extensions of the pattern matching problem. Both have been studied extensively. [2,11,15,8,18,26,27,14,6,19,20]. These problems have several applications in information retrieval, bioinformatics, data mining, and internet traffic analysis [7,13].

The focus of this paper is on the following problem: index T for handling matching of a query pattern P with at most k wildcards. A wildcard, also known as don't care character (represented by ϕ) can match with any other character in the alphabet set Σ (of size σ). Therefore, the pattern P can be written as $P_0\phi P_1\phi..P_{h-1}\phi P_h$, the concatenation of substrings $P_0, P_1, ...P_{h-1}, P_h$ separated by ϕ and $h \leq k$ is the number of wildcards in P. The first non-trivial solution for this problem was proposed by Cole et al. [11], where the index space is $O(n \log^k n)$ words or $O(n \log^{k+1} n)$ bits and query time is $O(p + 2^h \log\log n + occ)$. Recently, Bille et al. [6] proposed an index, which is a generalization of Cole et al.'s index. The space and query time are $O(n \log n \log_\beta^{k-1} n)$ words and $O(p+\beta^h \log\log n+occ)$ respectively, where $2 \leq \beta \leq \sigma$. Note that Cole et al.'s [11] result can be obtained by substituting $\beta = 2$. Bille et al. [6] also proposed an optimal $O(p + occ)$ time index of space $O(n\sigma^{k^2} \log^k \log n)$ words. Another space-efficient index of $O(n \log n)$ words proposed by Cole et al. [11] can answer this query in $O(p + \sigma^h \log\log n + occ)$ time, and is recently improved to $O(n)$ words without affecting the query time [6]. Several other linear space structures also exist in literature, such as the ones by Iliopouls and Rahman [22], and Lam et al. [18]. However, these indexes take $\Theta(nh)$ worst case time for answering the query. Despite all these continued efforts, a clear improvement over the seminal result by Cole et al. [11] (i.e., $O(n \log^{k+1} n)$ bits and $O(p + 2^h \log\log n + occ)$ time) is still not known.

In this paper, we describe two results. The first one is an $O(n \log^{k+\epsilon} n)$ bits index with $O(p + 2^h \log\log n + occ)$ query time, where where $0 < \epsilon < 1$ is an arbitrary small constant. The second one is an $O(n \log^k n \log \sigma)$ bits index, but with a slightly worse query time of $O(p + 2^h \log n + occ)$, where $\Sigma = [\sigma]$ denotes the alphabet set. Notice that our first result is a clear improvement over the earlier result by Cole et al., whereas the second one provides another space-time trade-off for this problem when the alphabet set is small.

Another problem that is strongly connected to the problem under consideration is to index the text wildcards. This was solved in Cole et al. [11] as well. However, for this case, better solutions have appeared in a succession of papers and indexes with succinct space and competitive query time [18,26,27,14] are available in the literature. Yet another related problem is that of indexing with gaps. Gaps are essentially longer wildcards. In [16] an index was proposed supporting queries of patterns containing one gap with a predefined length. This

[1] All logarithms in this article are base 2.

result builds on the result of [2]. The case of one gap was further improved by Bille et al. [5] with optimal query time. In [19] results were shown for the case when there is a larger number of gaps.

Outline. Section 2 gives the preliminaries. Next, we describe a classical framework for the case where $k = 1$ and then the framework by Cole et al.'s for $k \geq 1$ in Section 3, and Section 4, respectively. Section 5 describes our space-efficient data structures.

2 Preliminaries

2.1 Suffix Trees and Suffix Arrays

Suffix trees [28] and suffix arrays [21] are two classic data structures for online pattern matching queries. For a text $T[1..n]$, substring $T[i..n]$, with $i \in [1, n]$, is called a suffix of T. The suffix tree for T is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the i^{th} leftmost leaf represents the i^{th} lexicographically smallest suffix. For each node v in the suffix tree, we use $path(v)$ to denote the concatenation of edge labels along the path from the root to v. For any pattern P (of length p), the locus of P in the suffix tree is defined to be the highest node v (i.e., the closest node from the root) such that P is a prefix of *path(v)* and can be computed in $O(p)$ time.

The suffix array $SA[1..n]$ is an array of length n, such that $SA[i]$ is the starting position of the i^{th} lexicographically smallest suffix of T. The suffix array has an important property that the starting positions of all suffixes with the same prefix are always stored in a contiguous region in SA. Based on this property, the suffix range of a pattern P in SA is defined as the the maximal range $[sp, ep]$ such that for all $j \in [ep, ep], SA[j]$ is the starting point of a suffix of T with P as a prefix. In other words, the suffix range of a string represents the set of leaves in the subtree of its locus node in suffix tree. We also define its inverse, SA^{-1} to be an array such that $SA[i] = j$ if and only if $SA^{-1}[j] = i$. Both suffix trees and suffix arrays (along with an auxiliary data structure called LCP array) take $(n \log n)$ bits space and can perform pattern matching in optimal $O(p+occ)$ and $O(p + \log n + occ)$ time respectively, where occ is the number of occurrences of P in T.

2.2 Heavy Path and Heavy Path Decomposition

Let $\mathcal{T}$ be a tree with n nodes. We define the *size* of an internal node v to be the number of leaves in the subtree rooted at v. Then the *heavy path* of the tree $\mathcal{T}$ is the path starting from the root, where each node v on the path is the largest-size child of its parent. The *heavy path decomposition* of the tree $\mathcal{T}$ is the operation where we decompose each off-path subtree of the heavy path recursively; as a result, the edges in $\mathcal{T}$ will be partitioned into disjoint heavy paths. In [25], Sleator and Tarjan proved that the path from the root of $\mathcal{T}$ to any node v traverses at most $\log n$ heavy paths.

2.3 Two-Dimensional Orthogonal Range Reporting

Let $\mathcal{R} = \{(x_1, y_1), (x_2, y_2), .., (x_n, y_n)\}$ be a set of n points in an $[1, n] \times [1, n]$ grid. Without loss of generality, we assume that $x_i \leq x_{i+1}$. An orthogonal range reporting query on $\mathcal{R}$ is defined as follows: Given a query range $[x', x''] \times [y', y'']$, report all points (x_i, y_i) such that $x_i \in [x', x'']$ and $y_i \in [y', y'']$. Such a query can be answered optimally in $O(\log\log n + occ)$ time using an $O(n \log^\epsilon n)$-word space structure, where $\epsilon > 0$ is any arbitrary small constant [1]. See [20,10] for connections between text indexing and range searching.

2.4 Partial Rank Queries

Let $E[1..n]$ be an array of n characters taken from an alphabet set $\Sigma = [\sigma]$. Then $rank_E(i, c)$ where $c \in \Sigma$ is defined as the number of occurrences of c in $E[1..i]$. There exists $n \log\sigma + o(n \log\sigma)$-bit representations of E which can answer *rank* queries in $O(\log\log\sigma)$ time [12]. Rank queries of the type $rank_E(i, E[i])$ (or simply $prank_E(i)$) are called partial rank queries (also known as special rank queries [17]), and can be supported in constant time by maintaining an additional $o(n \log\sigma)$ bits structure [3,4].

3 The Classical Framework for $k = 1$

In this section, we describe a simple index for pattern matching with exactly one wildcard character. In this case, P can be written as $P_0\phi P_1$, where P_0 and P_1 are the longest prefix and suffix respectively of P which do not contain any wildcard. The index is based on the following idea by Amir et al. [2]: if there exists an occurrence of P in T with the wildcard character ϕ matching exactly at the location $i \in [1, n]$ in T, then P_0 must be a suffix of $T[1..i-1]$ and P_1 must be a prefix of $T[i+1..n]$. All such i's can be quickly computed by maintaining the following structures:

1. Suffix tree of T (ST)
2. Suffix tree of T^R (RST), where T^R is the reverse of T. i.e., $T^R[i] = T[n-i+1]$.
3. A two-dimensional orthogonal range reporting structure (RR2D) over a set of n points of the form (x_i, y_i), where x_i is the lexicographic rank of $T[i+1..n]$ among all suffixes of T, and y_i is the lexicographic rank of $T[1..i-1]^R$ among all suffixes of T^R.

The index space can be bounded by $O(n \log^\epsilon n)$ words, where ST and RST takes $O(n)$-word space and RR2D structure (Section 2.3) takes $O(n \log^\epsilon n)$-word space. The query corresponding to an input $P = P_0\phi P_1$ can be answered as follows: first find the suffix range $[sp, ep]$ of P_1 in ST, and the suffix range $[sp', ep']$ of P_0^R in RST in $O(|P_0| + |P_1|)$ time. Then, we issue a 2-dimensional orthogonal range reporting query on *RR2D* structure with $[sp, ep] \times [sp', ep']$ as the query range. The required time will be $O(\log\log n)$ plus the number of outputs. Corresponding to each point (x_j, y_j) reported as an output, there exists a match of P in T at the position $j - |P_0|$. Putting everything together, the total query time can be bounded as $O(|P_0| + |P_1| + \log\log n + occ)$. Bille et al. [5] showed that the $\log\log n$ additive factor in time can be removed by maintaining an $O(n \log\log n)$-word and optimal query time structure for $p < \log\log n$.

Theorem 1. *A given text $T[1..n]$ can be indexed in $O(n \log^{\epsilon} n)$ words, and all occurrences of a query pattern $P[1..p] = P_0 \phi P_1$ can be retrieved in $O(p + occ)$ time, where $0 < \epsilon < 1$ is an arbitrary small constant.*

Unfortunately, this approach cannot be generalized for $k \geq 2$.

4 Cole et al.'s Framework

In this section, we briefly describe the structure (we name it as STR_k) by Cole et al. [11] for handling pattern matching with at most k number of wildcards. The exact pattern matching problem (i.e., $k = 0$) can be answered using a suffix tree data structure, and for consistency we denote the suffix tree of T by STR_0. We shall call the nodes in STR_0 as *level*-0 *nodes.* The structure STR_k can be constructed in a recursive manner. We start with the description of STR_1, which is essentially an STR_0 with each of its nodes augmented with a compact trie called a side tree as follows: for every node u in STR_0 (i.e., level-0 nodes), with v being a child on the same heavy path as that of u, we choose all suffixes in the subtree of u [2], but not in the subtree of v, delete their first $|Path(u)| + 1$ characters [3] and maintain them as a compact trie. We call this compact trie as the side tree of u and is represented by $Sidetree(u)$. Then u is connected to the root of $Sidetree(u)$ via an edge with label ϕ (we fix the root of $Sidetree(u)$ as the last child of u). We now call a node a *level*-1 *node*, if it belongs to any *Sidetree* associated with a level-0 node. Using the same procedure as described above for constructing side trees from level-0 nodes, we construct side trees from level-1 nodes and call the newly formed nodes as *level*-2 *nodes.* Then we construct side trees from level-2 nodes and obtain *level*-3 *nodes* as so on until *level*-k *nodes.* The number of level-j nodes is given by $O(n \log^j n)$, therefore STR_k consists of $O(n \sum_{j=1}^{k} \log^j n) = O(n \log^k n)$ nodes and it can be maintained in $O(n \log^k n)$ words or $O(n \log^{k+1} n)$ bits. For every node u in STR_k, $path(u)$ represents the concatenation of edge labels on the path from the root of STR_k to u. Let ℓ_i represents the ith leftmost leaf node in STR_k. Notice that $path(\ell_i)$ corresponds to a suffix of T and we use $pos(\ell_i)$ to denote the starting position of that suffix[4]. Moreover if ℓ_i is a level-0 node, then $path(\ell_i) = T[pos(\ell_i)..n]$, whereas if it is a level-j node for $j \in [1, k]$, then $path(\ell_i)$ is given by $T[pos(\ell_i)..n]$ with its j characters replaced by ϕ.

Now a query corresponding to a pattern $P = P_0 \phi P_1 \phi .. \phi P_h$ can be answered as follows: start navigating the structure STR_k from its root by matching the

[2] This means all suffixes corresponding to the leaves in the subtree of u.

[3] which is the same as removing $|Path(u)| + 1$ characters from the prefix of all those suffixes, yet again, a collection of suffixes

[4] In the case of suffix tree STR_0, $pos(\ell_i) = SA[i]$.

characters in P one by one. Note, because ϕ is a wildcard it can match with any other character. However, if we have reached up to a node u in STR_k by matching a prefix of P and the next character to be matched is ϕ, by continuing to match ϕ with any other character will branch out the search into $degree(u)$ paths, where $degree(u) \leq \sigma + 1$ represents the number of outgoing edges from u. Cole et al. [11] observed that instead of matching in $degree(u)$ paths, it is enough to take only the following two paths (i) the outgoing path from u with its first character being ϕ and (ii) the heavy path on which u is sitting. Thus due to a single wildcard, the query will branch out to two paths, and in general for h wildcards, query will branch out to at most 2^h paths, ending up in $O(2^h)$ locus nodes. However, the time required for finding those $O(2^h)$ locus nodes is $O(|P_0| + 2|P_1| + 4|P_2| + ... + 2^h|P_h|) = O(2^h p)$. Using some auxiliary data structures, which are called *LCP data structures* occupying $O(n \log^{k+1} n)$ bits, this time complexity can be improved to $O(p + 2^h \log\log n)$. Then for every leaf ℓ_i in the subtree of a locus node, $pos(\ell_i)$ represents an occurrence of P in T. Thus all occurrences can be reported by spending another $O(occ)$ time.

Theorem 2. *([11]) A given text T of length n can be indexed in $O(n \log^{k+1} n)$ bits, such that all those occ occurrences of a pattern P containing $h \leq k$ wildcards can be reported in $O(p + 2^h \log\log n + occ)$ time.*

4.1 Finding Locus Nodes without *LCP Data Structures*

Even without the *LCP data structures*, the locus nodes can be computed efficiently using an $O(p + 2^h \log n)$ time algorithm. We start with the following definition: let $loc(u, d)$ refers to the location on the path from the root of STR_k to node u, such that the string obtained by concatenating edge labels on the path from the root of STR_k to $loc(u, d)$ (denoted by $path(loc(u, d))$) is the prefix of $path(u)$ of length $|path(u)| - d$. Notice that, $loc(u, 0)$ refers to node u itself. The maximum value of d for a particular node u is restricted by the following condition that there exits no other node on the path from $loc(u, d)$ to u. We now prove the following result.

Lemma 1. *Let $loc(u', d')$ represents a location in STR_k, which can be reached if we start matching a pattern P' from the location $loc(u, d)$. Then, given $loc(u, d)$ and the suffix range $[L', R']$ of a pattern P' in the suffix tree STR_0, we can find $loc(u', d')$ (if it exists) in $O(\log n)$ time.*

Proof. Let $\{\ell_i | i \in [x, y]\}$ and $\{\ell_i | i \in [x', y']\}$ represent the set of leaves in the subtree of u and u' respectively. Notice that $x \leq x' \leq y' \leq y$. Since the first $|path(loc(u, d))|$ characters are the same for all strings corresponding to $path(\ell_i)$ for $i \in [x, y]$, the lexicographic ordering among these strings will remain unchanged even if we remove their first $|path(loc(u, d))|$ characters. This means the function $SA^{-1}[pos(\ell_i) + |path(loc(u, d))|]$ is monotonically increasing with respect to $i \in [x, y]$. Moreover their next $|P'|$ characters match with P' iff $SA^{-1}[pos(\ell_i) + |path(loc(u, d))|] \in [L', R']$. Therefore x' and y' are the minimum and the maximum values of j satisfying this condition respectively, and they can be computed in $O(\log n)$ time using a binary search. Once x' and y' have been identified, u' can be computed in $O(1)$ by taking the lowest common ancestor of $\ell_{x'}$ and $\ell_{y'}$, and d' is given by $|path(u')| - |path(u)| + d' - |P'|$. Notice that $|path(\cdot)|$ for every node can be stored explicitly without changing the space bounds. □

Using the above result, we can compute the locus nodes as follows: for $i = 0, 1, 2, ..., h$, find the suffix ranges $[sp_i, ep_i]$ of P_i in the suffix tree STR_0 in overall $O(p)$ time. Now start navigating STR_k from its root by matching the characters of P_0. Whenever the query branch out to two paths, and if the next character to be matched is a wildcard character, it takes $O(1)$ per match. After that if we want to match the next P_i characters for some $i \in [1, h]$, we simply use the result in Lemma 1. Therefore, total time for pattern search can be bounded by $O(p + \log n + 2\log n + 4\log n + ... + 2^h \log n) = O(p + 2^h \log n)$. By putting every thing together, we have the following result.

Lemma 2. *There exists an $O(p + 2^h \log n)$ time algorithm for finding the locus nodes of P in STR_k.* □

5 New Space-Efficient Indexes

5.1 An $O(n \log^{k+\epsilon} n)$-bit Index

This result is achieved by a simple combination of the classical framework and Cole et al.'s framework. If there exists an occurrence of P in T with the first wildcard character ϕ matching exactly at the location $i \in [1, n]$ in T, then P_0 must be a suffix of $T[1..i-1]$ and $P_1\phi..\phi P_h$ must be a prefix of $T[i+1..n]$. All such i's can be quickly computed by maintaining the following structures:

- Cole et al.'s structure (STR_{k-1} of space $O(n \log^k n)$ bits) for handling the case only up to $k-1$ wildcards (along with the LCP data structures). The number of nodes in this structure is $O(n \log^{k-1} n)$. Here we use ℓ_i to denote the ith leftmost leaf in STR_{k-1}.
- Suffix tree of T^R (RST).
- Let L_i represents the set of leaves in STR_{k-1} with its $pos(\cdot) = i + 1$ and let i' be the lexicographic rank of $T[1..i-1]^R$ among all suffixes of T^R. Construct the set S_i of two dimensional points (j, i') corresponding to each leaf $\ell_j \in L_i$. Note that $|L_i| = |S_i| = O(\log^{k-1} n)$. We then maintain an orthogonal range reporting structure RR2D (refer to section 2.3) over a set $\cup_{i=2}^{n-1} S_i$ of $O(n \log^{k-1} n)$ two dimensional points. The space required for this component is $O(n \log^{k+\epsilon} n)$ bits.

Now the pattern matching query can be answered as follows: if $h \leq k-1$, the query can be answered using STR_{k-1} in $O(p + 2^h \log\log n + occ)$ time. If $h = k$, we spilt the pattern P into $P_{suf} = P_1\phi..\phi P_h$ and $P_{pre} = P_0^R$. Then, search for P_{suf} in STR_{k-1} and compute $O(2^{h-1})$ locus nodes $u_P^1, u_P^2, u_P^3, ..$ and their corresponding suffix ranges $[L_1, R_1], [L_2, R_2], [L_3, R_3], ..$ etc (here ℓ_{L_z} and ℓ_{R_z} represents the leftmost and the rightmost leaves in the subtree of u_P^z) in $O(p+2^h \log\log n)$ time (using *LCP* data structures). Then search for P_{pre} in RST and obtain the suffix range $[sp', ep']$. Finally the occurrences can be computed by issuing $O(2^{h-1})$ two-dimensional range reporting queries on RR2D corresponding to the ranges $[L_1, R_1] \times [sp', ep'], [L_2, R_2] \times [sp', ep'], [L_3, R_3] \times [sp', ep']$... It can be easily verified that for every point $(j, .)$ reported as an output by the structure, there exists an occurrence of P_{suf} starting at the location $pos(\ell_j)$ and an occurrence of P_{pre} ending at the location $pos(\ell_j) - 2$ in T. Hence an occurrence of P at the location $pos(\ell_j) - |P_0| - 1$. By combining the above pieces, we have the following theorem.

Theorem 3. *A given text T of length n can be indexed in $O(n \log^{k+\epsilon} n)$ bits, such that all those occurrences of a pattern P containing $h \leq k$ wildcards can be retrieved in $O(p + 2^h \log\log n + occ)$ time, where $0 < \epsilon < 1$ is an arbitrary small constant.* □

By using an alternative RR2D structure of $O(n)$-word space with query time $O((1 + output) \log^{\epsilon} n)$ [9], we can obtain another space-time trade-off as follows:

Corollary 1 *A given text T of length n can be indexed in $O(n \log^k n)$ bits, such that all those occurrences of a pattern P containing $h \leq k$ wildcards can be retrieved in $O(p + (2^h + occ) \log^{\epsilon} n)$ time, where $0 < \epsilon < 1$ is an arbitrary small constant.*

Remark. Our techniques can be combined with the result by Bille et al. [6], and an $O(n \log^{1+\epsilon} n \log_{\beta}^{k-2} n)$-word index with $O(p + \beta^{h-1} \log\log n + occ)$ query time can be obtained, where $0 < \epsilon < 1$ is an arbitrary small constant and $2 \leq \beta \leq \sigma$.

5.2 An $O(n \log^k n \log \sigma)$-bit Index via Side Tree Compression

First we maintain the structure STR_{k-1} (as described before) in $O(n \log^k n)$ bits space. Therefore, the string matching case where the number of wildcards is at most $k-1$ can be handled efficiently. In order to handle the k-wildcard case (i.e., $h = k$), we augment the side trees with every level-$(k-1)$ node in STR_{k-1} and obtain STR_k. The explicit storage of these side trees requires $O(\log n)$ bits per node. However, the desired storage space of $O(\log \sigma)$ bits per node is achieved via a novel encoding technique. For every level-$(k-1)$ node u in STR_k, we define the followings:

- ℓ_i^u represents the ith leftmost leaf in $Sidetree(u)$
- $E_u[1..n_u]$ be an array of characters, where $E_u[i] = T[pos(\ell_i^u) + |path(u)| + 1]$, and n_u represents the number of leaves in $Sidetree(u)$.
- $B_u[1..\sigma]$ be a bit vector of length σ, where $B_u[z] = 1$ if and only if there exists an outgoing edge from u with $z \in \Sigma$ as the leading character.

The following lemma summarizes the key idea behind our result.

Lemma 3. *For any level-$(k-1)$ node u and $i \in [1, n_u]$, $pos(\ell_i^u)$ is the same as $pos(\cdot)$ of the $prank_{E_u}(i)$th leftmost leaf node in the subtree of node w, where w is a child of u, with $E_u[i]$ the leading character on the edge connecting u and w.*

Proof. Corresponding to every leaf node in the subtree of any child node of u, except the one on the same heavy path as that of u, there exists another unique leaf node in $Sidetree(u)$, such that both have the same $pos(\cdot)$ value. Then the lemma follows from the fact that, the character at the position $|path(u)|+1$ is the same for the suffix corresponding to any two leaves in the subtree of w, and therefore the lexicographic ordering of those suffixes remains unchanged even after replacing the $(|path(u)|+1)$th character by ϕ. □

Based on the key observation in the above lemma, we obtain the following result.

Lemma 4. *By maintaining an $O(n \log^k n \log \sigma)$ bits structure, we can compute $pos(\ell_{u_i})$ for any $i \in [1, n_u]$ for any level-$(k-1)$ node u in $O(1)$ time.*

Proof. First we maintain the tree structure of STR_k using succinct data structures [24] in $O(n \log^k n)$ bits of space. Then for every level-$(k-1)$ node u, we maintain E_u and the supporting structures for constant time partial rank queries (refer to Section 2.4) on E_u, in total $O(\sum n_u \log \sigma) = O(n \log^k n \log \sigma)$ bits. Also maintain $B_u[1..\sigma]$ corresponding to all level-$(k-1)$ nodes u, where $B_u[1...\sigma]$ for a particular node u can be maintained in $O(degree(u) \log(\sigma/degree(u)))$ bits or $O(degree(u))$ words of space using an indexible dictionary [23]. Notice that the total space (in words) for maintaing all such bit vectors can be asymptotically bounded by the number of level-$(k-1)$ nodes, which is $O(n \log^{k-1} n)$. By combining the above pieces, the overall space can be bounded by $O(n \log^k n \log \sigma)$ bits. Using these structure, combined with the result in Lemma 3, $pos(\ell_{u_i})$ for any $i \in [1, n_u]$ for any level-$(k-1)$ node u can be answered in $O(1)$ time as follows:

– Find the child node w of u, such that the leading character on the edge connecting u and w is $E_u[i]$ using the following steps: find $k = rank_{B_u}(E_u[i])$ (notice that $B_u[E_u[i]] = 1$, therefore k can be computed in $O(1)$ from B_u, which is maintained using an indexible dictionary) and w is given by the kth leftmost child of u (which can be identified in constant time using the tree structure of STR_k).
– Report $pos(\cdot)$ of the $(prank_{E_u}(i))$th leaf node in the subtree of w, which is again a constant time operation. □

Lemma 5. *The structure STR_k can be encoded in $O(n \log^k n \log \sigma)$ bits such that $pos(\cdot)$ of any of its leaf node can be computed in $O(1)$ time.*

Proof. The $pos(\cdot)$ values corresponding to all those leaves, which are not a *level-k* node can be maintained explicitly in $O(n \log^k n)$ bits. This is because the number of such leaves is $O(n \log^{k-1} n)$. In order to encode these values efficiently for *level-k* leaf nodes, we first mark all those nodes in STR_k which are not *level-k*. The information whether a node in STR_k is marked or not can be maintained using a bit vector B of length equal to the number of nodes in STR_k, where $B[i] = 1$ iff the ith node (in terms of pre-order rank) is marked. Then, $pos(\ell_j)$ of any *level-k* (i.e., unmarked) leaf node can be computed as follows: first find the lowest marked ancestor u of ℓ_j. Let ℓ_j be the ith leftmost leaf in the subtree of u, where $i = j - f + 1$ and ℓ_f is the leftmost leaf in the subtree of u (notice that f can be computed in $O(1)$ time). Therefore $pos(\ell_j) = pos(\ell_i^u)$ and can be decoded in $O(1)$ time using the result of Lemma 4. □

A pattern matching query on our encoded STR_k can be performed in the same standard way. Notice that using an LCP data structure, the locus nodes can be identified in $O(p + 2^h \log\log n)$ time. However its space occupancy is $O(n \log^{k+1} n)$ bits and we cannot afford to maintain it within the desired space complexity. Therefore, (although slower) we use the $O(p + 2^h \log n)$ time algorithm described in Section 4.1 for identifying the locus nodes. As $pos(\cdot)$ for any leaf node can be decoded in $O(1)$ time (refer to Lemma 5), after finding the locus nodes, it takes only $O(occ)$ time to report the occurrences. By combining the above pieces, we have the following final result.

Theorem 4. *A given text T of length n can be indexed in $O(n \log^k n \log\sigma)$ bits, such that all those occurrences of a pattern P containing $h \le k$ wildcards can be retrieved in $O(p + 2^h \log n + occ)$ time.* □

References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: FOCS, pp. 198–207 (2000)
2. Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Text indexing and dictionary matching with one error. J. Algorithms 37(2), 309–325 (2000)
3. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: searching a sorted table with O(1) accesses. In: SODA, pp. 785–794 (2009)
4. Belazzougui, D., Navarro, G., Valenzuela, D.: Improved compressed indexes for full-text document retrieval. J. Algorithms 18, 3–13 (2013)
5. Bille, P., Gørtz, I.L.: Substring range reporting. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 299–308. Springer, Heidelberg (2011)
6. Bille, P., Gørtz, I.L., Vildhøj, H.W., Vind, S.: String indexing for patterns with wildcards. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. 7357, pp. 283–294. Springer, Heidelberg (2012)
7. Bucher, P., Bairoch, A.: A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In: ISMB, pp. 53–61 (1994)
8. Chan, H.-L., Lam, T.-W., Sung, W.-K., Tam, S.-L., Wong, S.-S.: Compressed indexes for approximate string matching. Algorithmica 58(2), 263–281 (2010)
9. Chan, T.M., Larsen, K.G., Patrascu, M.: Orthogonal range searching on the RAM, revisited. In: Symposium on Computational Geometry, pp. 1–10 (2011)
10. Chien, Y.-F., Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: Geometric BWT: Compressed text indexing via sparse suffixes and range searching. Algorithmica (2013)

11. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC, pp. 91–100 (2004)
12. Golynski, A., Ian Munro, J., Srinivasa Rao, S.: Rank/select operations on large alphabets: a tool for text indexing. In: SODA, pp. 368–373 (2006)
13. Hofmann, K., Bucher, P., Falquet, L., Bairoch, A.: The prosite database, its status in 1999. Nucleic Acids Research 27(1), 215–219 (1999)
14. Hon, W.-K., Ku, T.-H., Shah, R., Thankachan, S.V., Vitter, J.S.: Compressed text indexing with wildcards. J. Discrete Algorithms 19, 23–29 (2013)
15. Huynh, T.N.D., Hon, W.-K., Lam, T.-W., Sung, W.-K.: Approximate string matching using compressed suffix arrays. Theoretical Comp. Science 352(1), 240–249 (2006)
16. Iliopoulos, C.S., Rahman, M.S.: Indexing factors with gaps. Algorithmica 55(1), 60–70 (2009)
17. Kärkkäinen, J., Puglisi, S.J.: Medium-space algorithms for inverse BWT. ESA (1), 451–462 (2010)
18. Lam, T.-W., Sung, W.-K., Tam, S.-L., Yiu, S.-M.: Space efficient indexes for string matching with don't cares. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 846–857. Springer, Heidelberg (2007)
19. Lewenstein, M.: Indexing with gaps. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 135–143. Springer, Heidelberg (2011)
20. Lewenstein, M.: Orthogonal range searching for text indexing. In: Brodnik, A., López-Ortiz, A., Raman, V., Viola, A. (eds.) Ianfest-66. LNCS, vol. 8066, pp. 267–302. Springer, Heidelberg (2013)
21. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
22. Rahman, M.S., Iliopoulos, C.S.: Pattern matching algorithms with don't cares. In: SOFSEM (2), pp. 116–126 (2007)
23. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4) (2007)
24. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: SODA, pp. 134–149 (2010)
25. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. 26(3), 362–391 (1983)
26. Tam, A., Wu, E., Lam, T.-W., Yiu, S.-M.: Succinct text indexing with wildcards. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 39–50. Springer, Heidelberg (2009)
27. Thachuk, C.: Compressed indexes for text with wildcards. Theor. Comput. Sci. 483, 22–35 (2013)
28. Weiner, P.: Linear pattern matching algorithms. In: SWAT (FOCS), pp. 1–11 (1973)