Compressed Text Indexing With Wildcards *

Wing-Kai Hon[1], Tsung-Han Ku[1], Rahul Shah[2],
Sharma V. Thankachan[2], and Jeffrey Scott Vitter[3]

[1] National Tsing Hua University, Taiwan. {wkhon,thku}@cs.nthu.edu.tw
[2] Louisiana State University, USA. {rahul,thanks}@csc.lsu.edu
[3] The University of Kansas, USA. jsv@ku.edu

Abstract. Let $T = T_1\phi^{k_1}T_2\phi^{k_2}\cdots\phi^{k_d}T_{d+1}$ be a text of total length n, where characters of each T_i are chosen from an alphabet Σ of size σ, and ϕ denotes a wildcard symbol. The text indexing with wildcards problem is to index T such that when we are given a query pattern P, we can locate the occurrences of P in T efficiently. This problem has been applied in indexing genomic sequences that contain single-nucleotide polymorphisms (SNP) because SNP can be modeled as wildcards. Recently Tam et al. (2009) and Thachuk (2011) have proposed succinct indexes for this problem. In this paper, we present the first compressed index for this problem, which takes only $nH_h + o(n\log\sigma) + O(d\log n)$ bits space, where H_h is the hth-order empirical entropy ($h = o(\log_\sigma n)$) of T.

1 Introduction

Text indexing is a fundamental problem in computer science, where the task is to index a given text $T[1..n]$ for locating all the occurrences of an online query pattern $P[1..p]$ within T efficiently. Suffix trees [20, 14] and suffix arrays [13] are the most popular indexes which can answer this query in $O(p + occ)$ and $O(p + \log n + occ)$ times respectively, where occ is the number of occurrences of P in T. Both indexes take $O(n\log n)$ bits space. Note that the query time is (almost) optimal, but the index size can be asymptotically higher than the optimal $n\lceil\log\sigma\rceil$ bits required to store the text in plain form; here, σ denotes the size of the alphabet Σ from which the characters of T and P are chosen. The goal of designing optimal-size indexes was first achieved by Grossi and Vitter (Compressed Suffix Arrays) [8] and Ferragina and Manzini (FM-index) [5]. Their indexes are based on Burrows-Wheeler transform (BWT) [2].

A more general problem of text indexing deals with the case where wildcard characters may appear in the input text T [4, 17, 18]. Let $T = T_1\phi^{k_1}T_2\phi^{k_2}\cdots\phi^{k_d}T_{d+1}$ be a text of total length n, where characters of

* This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W. Hon) and US NSF Grant CCF–1017623 (R. Shah).

each T_i are chosen from an alphabet Σ of size σ, and ϕ represents a wildcard symbol that may match any single character in Σ. The text indexing problem with wildcards is to index T such that when we are given a query pattern P, we can locate the occurrences of P in T efficiently. This problem has been applied in indexing genomic sequences that contain single-nucleotide polymorphisms (SNP) because SNP can be modeled as wildcards.

The problem of text indexing with wildcards was first studied by Cole et al [4]. They proposed an $O(n \log^k n)$-word index with $O(p + \log^k n \log\log n + occ)$ query time, where k is the total number of wildcards. Later, Lam et al. [17] proposed an $O(n)$-word index, but the query time introduces an additional term of $\gamma = \sum_{i,j} \mathit{prefix}(P[i..p], T_j)$, where $\mathit{prefix}(P[i..p], T_j) = 1$ if the text segment T_j is a prefix of $P[i..p]$, else 0. Note that γ is upper bounded by pd. Tam et al. [18] further reduced the space requirement of this index to $(3 + o(1))n \log \sigma + O(d \log n)$ bits. Recently, Thachuk [19] proposed a more space-efficient solution, taking $(2 + o(1))n \log \sigma + O(n) + O(d \log n) + O(k \log k)$ bits and requiring a smaller working space of $O((d + \log n)p)$ bits.

In all the above solutions (which has a γ term in query time), the common approach is to categorize the occurrences into the following 3 types and build separate data structures for reporting each type of occurrences of P in T.

Type-1: P matching a substring of T with no wildcard groups;
Type-2: P matching a substring of T with exactly 1 wildcard group;
Type-3: P matching a substring of T with 2 or more wildcard groups.

In this paper, we propose an index which takes near-optimal $nH_h + o(n \log \sigma) + O(d \log n)$ bits space, where H_h is the hth-order empirical entropy of T. The central technique is to make use of the *same* data structure (which is an FM-index of T) for handling all types of occurrences. We need auxiliary structures in locating type-2 and type-3 occurrences, but the space of those structures is bounded by $O(d \log n) + o(n)$ bits. Moreover, the working space requirement is $O((p+\gamma) \log n)$. As γ is upper bounded by dp, therefore our working space will be at most a $\log n$ factor worse than Thachuk's index. However, our index has its advantage when γ is small ($\gamma = o(dp/\log n)$). The table below summarizes the results of our index along with the previously known results supporting matching with wildcard characters. Here k, d and $\hat{d}$ represents the number of wildcards, wildcard groups, distinct wildcard group lengths, respectively; occ_1, occ_2 and occ represents the number of type-1, type-2 and overall occurrences of P in T respectively, and $\epsilon' > 0$ is any fixed constant.

Ref	Index (bits)	Query Time	Working (bits)
[4]	$O(n \log^{k+1} n)$	$O(p + \log^k n \log\log n + occ)$	–
[17]	$O(n \log n)$	$O(p \log n + \gamma + occ)$	$O(n \log n)$
[18]	$(3 + o(1))n \log \sigma + O(d \log n)$	$O(p(\log \sigma + \min(p, \hat{d}) \log d) + occ_1 \log n + occ_2 \log d + \gamma)$	$O(n \log d + p \log n)$
[18]	$(3 + o(1))n \log \sigma + O(d \log n)$	$O(p(\log \sigma + \min(p, \hat{d}) \log d) + occ_1 \log n + occ_2 \log d + \gamma \log_\sigma d)$	$O(n \log \sigma + p \log n)$
[19]	$(2 + o(1))n \log \sigma + O(n + d \log n + k \log k)$	$O(p(\log \sigma + \min(p, \hat{d}) \log k / \log\log k) + occ_1 \log n + occ_2 \log k / \log\log k + \gamma)$	$O(dp + p \log n)$
Ours	$nH_h + o(n \log \sigma) + O(d \log n)$	$O(p(\log^{1+\epsilon'} n + \min(p, \hat{d}) \log d) + occ_1 \log^{1+\epsilon'} n + occ_2 \log^{\epsilon'} d + \gamma \log \gamma)$	$O(\gamma \log n + p \log n)$

2 Preliminaries

2.1 Bit Vectors with Rank/Select

Let B be a bit vector of length n, the rank and select operations are defined as $rank(k) = \sum_{i=1}^{k} B[i]$ and $select(k) = i$ such that $A[i] = 1$ and $rank(i) = k$. Let d be the number of 1s in B, then B can be maintained in $d \log(n/d) + O(d + n \log\log n / \log n)$ bits such that both rank and select operations can be performed in constant time [16].

2.2 Suffix Trees and Suffix Arrays

Suffix trees [20, 14] and suffix arrays [13] are two classic data structures for online pattern matching queries. For a text $T[1..n]$, a 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 any node v, the string formed by concatenating the edge labels from root to v is called $path(v)$. The locus node v of a pattern P is defined as the node closest to the root, such that P is a prefix of $path(v)$.

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 range of a pattern P in SA is defined as the the maximal range $[L, R]$ such that for all $j \in [L, R]$, $SA[j]$ is the starting point of a suffix of T with P as a prefix. Both suffix trees and suffix arrays take $O(n \log n)$ bits space and can perform pattern matching in $O(p + occ)$ and $O(p + \log n + occ)$ time respectively, where $p = |P|$ and occ is the number of occurrences of P within T.

2.3 Compressed Text Indexes

Text indexes which take space close to the size of the text is called compressed/succinct text indexes. There are different compressed text indexes available in the literature, such as [8] and FerMan05. For our purpose, we use the FM-index by Ferragina et al. [6] which takes only $nH_h + o(n \log \sigma)$ bits space, where H_h denotes the hth-order empirical entropy ($h = o(\log_\sigma n)$) of T. For $\sigma = O(poly \log(n))$, this index can count the number of occurrences of P within T in $O(p)$ time, locate each pattern occurrence in $O(\log^{1+\epsilon} n)$ and display a text substring of length ℓ in $O(\ell + \log^{1+\epsilon} n)$ time.

2.4 Compressed Index for Dictionary Matching

The dictionary matching problem is to index a set of (short) text segments $\{T_1, T_2, \ldots, T_{d+1}\}$ of total length n, such that all the occurrences of these text segments within an online (long) query pattern P can be computed efficiently. This is a well-studied problem and many indexes are available in the literature [1, 9, 10]. The recently proposed indexes by Belazzougui [1] and Hon et al. [9] can solve this problem in (almost) optimal space and optimal time. For our purpose, we choose the most space-efficient index (even though the query time is not optimal) by Hon et al. [10]. Their index takes $nH_h + o(n \log \sigma) + O(d \log n)$ bits space and the query time is $O(p(\log^\epsilon n + \log d) + \gamma)$, where H_h denotes the hth-order empirical entropy of the text segments collection and γ denotes number of occurrences of text segments in P, and $\epsilon > 0$ is any fixed constant. In [10], it is assumed that the text segments are stored using Ferragina-Venturini scheme, where the storage space is $nH_h + o(n \log \sigma)$ bits and displaying a text substring of length ℓ takes $O(\ell / \log_\sigma n + 1)$ time [7] .

2.5 Orthogonal Range Reporting

Let $\mathcal{R} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ be a set of n points in the two-dimensional space. An orthogonal range reporting query on $\mathcal{R}$ is defined as follows: Given a query range $[x_\ell, x_r] \times [y_\ell, y_r]$, report all points (x_i, y_i) such that $x_\ell \leq x_i \leq x_r$ and $y_\ell \leq y_i \leq y_r$. For our purpose, we use the $O(n \log n)$ bits structure by Nekrich [15] which can perform an orthogonal range reporting of t output points in $O(\log n + t \log^\epsilon n)$ time.

2.6 Sparse Suffix Trees

Sparse suffix tree of a text is a compact trie, which consists of only selected suffixes of the text [3, 11, 12]. We shall define the sparse suffix tree for a

collection $\{T_1, T_2, \ldots, T_{d+1}\}$ of $d+1$ text segments of total length n as follows: Let α be a *sampling factor*, and for each text segment $T_j[1..|T_j|]$, the suffix $T[i..|T_j|]$ such that $i \; mod(\alpha) = 1$ is called an α-sampled suffix of T_j. A trie of all α-sampled suffixes of all text segments is called a (forward) sparse suffix tree Δ_f. Similarly a trie of all α-sampled suffixes of $\overleftarrow{T_j}$ for $j = 1, 2, \ldots, d+1$ is called (reverse) sparse suffix tree Δ_r, where $\overleftarrow{T_j}$ is the reverse of T_j. The number of nodes in Δ_f (Δ_r) can be bounded by $O(n/\alpha + d)$. For each internal node $u \neq root$, there exists a unique node v such that $path(v)$ can be obtained by deleting the first α characters of $path(u)$. Then we maintain a pointer from node u to node v, which we called an α-sampled suffix link. The contiguous range of all α-sampled suffixes in the subtree of u is called the suffix range of a pattern P, where u is the locus node (node closest to root such that P is a prefix of $path(u)$) of P. Note that neither Δ_f, nor Δ_r is a self-index, hence we maintain the original text $T = T_1\phi^{k_1}T_2\phi^{k_2}\cdots\phi^{k_d}T_{d+1}$ in the form of FM-index [6] in $nH_h + o(n\log\sigma)$ bits, which is capable of retrieving any substring of T of length ℓ in $O(\ell + \log^{1+\epsilon} n)$ time ($\sigma = \text{polylog}(n)$). We store the starting character of every edges explicitly and from every node ($\neq root$), we maintain a pointer to its ancestor. By choosing $\alpha = \log^{1+\epsilon} n, \epsilon > 0$, the size of Δ_f and Δ_r, together with the encoding of T, can be bounded by $nH_h + o(n\log\sigma) + O(d\log n)$ bits.

Lemma 1 *Given a pattern $P[1..p]$, the suffix ranges of all its suffixes ($P[i..p]$ for $i = 1, 2, 3, \ldots, p$) in Δ_f can be computed in $O(p\log^{1+\epsilon'} n)$ time. Similarly the suffix ranges of the reverse of all its prefixes ($\overleftarrow{P[1..i]}$, for $i = 1, 2, 3, \ldots, p$) in Δ_r can be computed in $O(p\log^{1+\epsilon'} n)$ time, where $\epsilon' > 0$.*

Proof : Firstly we show how to compute the suffix ranges of all suffixes of P in Δ_f. The procedure works in α stages. At stage k (for $k = 1, 2, 3, \ldots, \alpha$), we compute the suffix ranges of all α-sampled suffixes ($P[k..p]$, $P[(k+\alpha)..p]$, $P[(k+2\alpha)..p]$, $\ldots$) of $P[k..p]$. The main challenge comes from the fact that the time for retrieving a substring of T of length ℓ is $O(\ell + \log^{1+\epsilon} n)$, which means $O(\log^{1+\epsilon} n)$ time is needed for retrieving even a single character in T. We handle this situation carefully as follows: Firstly, we do a blind matching of first α characters of $P[k..p]$ in Δ_f by only matching the starting characters of the edges. Next, we verify if this matching is correct by retrieving α characters from the FM-index and matching them in $O(\alpha\log\sigma)$ time. If the first α characters are matched, we continue matching the next α characters, and so on. We stop when

all characters in $T[k..p]$ are matched or encounter the first mismatch. Let x be the number of characters matched, then the matching time can be bounded by $O(x \log \sigma + \log^{1+\epsilon} n)$ ($O(x + \log^{1+\epsilon} n)$ time for retrieving x characters from FM-index of T and matching those x characters with a prefix of $P[k..p]$ takes $O(x \log \sigma)$ time).

If $x = p-k+1$, then $P[k..p]$ is fully matched with prefixes of some text segments and the first node obtained by traversing further down in Δ_f will be the locus node u_k of $P[k..p]$. Now the locus node of $P[(k+\alpha)..p]$ can be computed as follows: First reach the node u'_k by chasing the α-sampled suffix link from u_k, then the locus node $u_{k+\alpha}$ of $P[(k+\alpha)..p]$ is given by the node closest to the root in the path from u'_k to root, which is at least $x - \alpha$ distance away from root. The locus node $u_{k+2\alpha}$ of $P[(k+2\alpha)..p]$ can be obtained similarly by further chasing the α-sampled suffix link of $u_{k+\alpha}$ and so on. The total number of α-sampled suffix links chased will be $O(p/\alpha+1)$ and given the locus node of a pattern, its suffix range can be obtained in constant time. Hence, the total time is bounded by $O(p \log \sigma + \log^{1+\epsilon} n)$.

If $x < p - k + 1$, then $P[k..p]$ is not fully matched. Let v_k be the node closest to the root such that x is the length of the longest common prefix of $P[k..p]$ and $path(v_k)$. Then, we chase the α-sampled suffix link from v_k and reach a node v'_k, and continue matching from the position $x - \alpha$ distance away from root in $path(v'_k)$. Thus, we continue the matching of $P[k..p]$, $P[(k+\alpha)..p]$, $P[(k+2\alpha)..p]$, and and so on by chasing the α-sampled suffix links. Each time we match a small portion of P (that portion will not be matched again) of length say ℓ' in $O(\ell' + \log^{1+\epsilon} n)$ time and the number of such portions (number of α-sampled suffix links chased) is $O(p/\alpha + 1)$. Hence, the time for computing the locus node (if exists) and the corresponding suffix ranges of all α-sampled suffixes of $P[k..p]$ is given by $O(p \log \sigma + \log^{1+\epsilon} n(p/\alpha + 1))$. Thus, the total time for computing the suffix ranges of all suffixes of $P[1..p]$ (i.e. α-sampled suffixes of $P[k..p]$, for $k = 1, 2, 3, \ldots, \alpha$) is given by $O((p \log \sigma + \log^{1+\epsilon} n) \log^{1+\epsilon} n) = O(p \log^{1+\epsilon} n \log \sigma)$, when $p > \log_\sigma n \log^\epsilon n$.

For computing the suffix range of short patterns ($p \leq \log_\sigma n \log^\epsilon n$), we maintain an $o(n)$ bits additional information on Δ_f as follows: Along the path of every $(\log^2 n)$th α-sampled suffix in Δ_f, we write the first $\log^{1+\epsilon} n$ characters explicitly, which takes only $O((n/\alpha + d)/\log^2 n) \log^{1+\epsilon} n) = o(n)$ bits extra space. Thus, to find the suffix range of a pattern P, we first compute the suffix range $[L, R]$ by considering only those suffixes whose first $\log^{1+\epsilon} n$ characters are explicitly written. This traversal will take only $O(p \log \sigma)$ time. After this, to find the exact suffix range, we

need to perform a binary search only among $\log^2 n$ α-sampled suffixes on either side of the $[L, R]$ and check if the pattern P is matching with its prefix. Thus we need to retrieve only $O(\log(\log^2 n))$ substrings of T (each of length $p \leq \log_\sigma n \log^\epsilon n$). Hence, the time for computing the suffix range of P is $O(p \log \sigma + \log^{1+\epsilon} n \log\log n) = O(\log^{1+\epsilon'} n)$, where $\epsilon' \geq 2\epsilon$. Now the suffix range of each suffix of P can be computed independently and the total time is $O(p \log^{1+\epsilon'} n)$.

The query time for finding the suffix ranges of the reverse of all prefixes of P ($\overleftarrow{P[1..i]}$, for $i = 1, 2, 3, \ldots, p$) in Δ_r can be analyzed in the same fashion. □

3 Matching with Wildcards in Compressed Text

A wildcard is a character which can match with any character in an alpahbet Σ, and it is denoted by ϕ in this paper. Given a text $T = T_1\phi^{k_1}T_2\phi^{k_2}\cdots\phi^{k_d}T_{d+1}$ of total length n, where each T_i are strings drawn over the alphabet Σ of size σ (assume $\sigma = O(poly\log(n))$), and ϕ^j denotes a string of j consecutive wildcards, our objective is to construct an index for T for locating all the occurrences of an online pattern P of length p efficiently. The general way to approach this problem is to categorize the occurrences of P in T into the following 3 types [17–19] and build a dedicated data structure for handling each type.

Type-1: P matching a substring of T with no wildcard groups;
Type-2: P matching a substring of T with exactly 1 wildcard group;
Type-3: P matching a substring of T with 2 or more wildcard groups.

We also follow the same approach, however we reduce the index space by making use of the *same* data structure (which is an FM-index of T) for handling all the three types of occurrences. We need auxiliary structures in type-2 and type-3 occurrences, but the space of those structures is bounded by $O(d \log n) + o(n)$ bits.

3.1 Type-1 Matching

In type-1 matching, we are looking for exact matches of P within T (without matching any wildcards). This case is the same as the basic text indexing problem, and by using FM-index [6], we have the following lemma.

Lemma 2 *By maintaining an $nH_h + o(n \log \sigma)$ bits index, all the type-1 occurrences can be reported in $O(p + occ_1 \log^{1+\epsilon} n)$ time, where occ_1 is the number of type-1 occurrences.*

3.2 Type-2 Matching

In type-2 matching, we are looking for each substring S within T which matches with P, such that S contains exactly one wildcard group (could be partial). We further divide such an occurrence S into the following 3 subcases:

1. S is a concatenation of a suffix of T_j , ϕ^{k_j} and a prefix of T_{j+1}. For this, we need to find all those text segments T_j such that $\overleftarrow{P[1..i]}$ is a prefix of $\overleftarrow{T_j}$ and $P[i+k_j+1..p]$ is a prefix of T_{j+1} for $1 \leq i < (i+k_j+1) \leq p$. If these conditions are satisfied, then P matches at position $z-i$ in T, where z is the starting position of jth wildcard group within T.
2. S is a concatenation of a portion of ϕ^{k_j} and a prefix of T_{j+1}. For this, we need to find all those text segments T_{j+1} such that $P[i+1..p]$ is a prefix of T_{j+1} and $1 \leq i \leq k_j$. Then, the matching position of P within T is $z+k_j-i$.
3. S is a concatenation of a suffix of T_j and a portion of ϕ^{k_j}. For this, we need to find all those text segments T_j such that $\overleftarrow{P[1..i]}$ is a prefix of $\overleftarrow{T_j}$ and $1 \leq p-i \leq k_j$. Then, the matching position of P within T is $z-i$.

The above conditions can be verified easily by maintaining different 2-dimensional orthogonal range reporting structures, where each structure corresponds to a distinct wildcard group lengths. Let RS_β represents the orthogonal range searching structure which links two text segments on either side of all wildcard groups of length β, then RS_β contains all those points (x_j, y_j) such that the (lexicographically) x_jth (α-sampled) suffix in Δ_r is $\overleftarrow{T_j}$, the (lexicographically) y_jth (α-sampled) suffix in Δ_f is T_{j+1}, and $k_j = \beta$ (i.e., there are exactly β wildcard symbols between T_j and T_{j+1} in T). The number of 2-dimensional orthogonal range reporting structures is $\hat{d}$ (the number distinct wildcard group lengths) and the total number of points among all those $\hat{d}$ structures is $O(d)$.

Now, type-2 matching can be performed as follows: first we obtain the suffix ranges $[L_i^f, R_i^f]$ of $P[i..p]$ for $i = 1, 2, \ldots, p$ in Δ_f and the suffix ranges $[L_i^r, R_i^r]$ of $\overleftarrow{P[1..i]}$ for $i = 1, 2, \ldots, p$ in Δ_r in $O(p \log^{1+\epsilon'} n)$ (using Lemma 1). Then, we have:

- All case-1 occurrences can be obtained by reporting all those points in RS_β with query range $[L_i^r, R_i^r] \times [L^f_{(i+\beta+1)}, R^f_{(i+\beta+1)}]$ for all $\beta < p$ and $1 \leq i < (i+\beta+1) \leq p$.

– All case-2 occurrences can be obtained by reporting all those points in RS_β with query range $[-\infty, \infty] \times [L^f_{(i+1)}, R^f_{(i+1)}]$ for all $\beta < p$ and $1 \leq i < \beta$.
– All case-3 occurrences can be obtained by reporting all those points in RS_β with query range $[L^r_i, R^r_i] \times [-\infty, \infty]$ for all $\beta < p$ and $1 \leq p - i < \beta$.

The number of orthogonal range searching queries is $O(p \min(p, \hat{d}))$ and the total number of points among all the orthogonal range searching structures is $O(d)$. Therefore, by using an $O(d \log n)$-bit orthogonal range searching structure by [15], the query time can be bounded by $O(p \log^{1+\epsilon'} n + p \min(p, \hat{d}) \log d + occ_2 \log^{\epsilon'} d)$, where occ_2 is the number of type-2 occurrences. Moreover, P can trivially be matched at all $(k_j - p + 1)$ positions within a wildcard group of length $k_j \geq p$.

Lemma 3 *By maintaining an $nH_h + o(n \log \sigma) + O(d \log n)$ bits index, all the type-2 occurrences of P in T can be reported in $O(p(\log^{1+\epsilon'} n + min(p, \hat{d}) \log d) + occ_2 \log^{\epsilon'} d)$ time, where $\epsilon' > 0$.*

3.3 Type-3 Matching

In type-3 matching, we are looking for each substring of T which matches with P and contains more than one wildcard groups. Therefore, P contains at least a whole text segment T_j. We follow a similar approach proposed by Lam et al. [17] for handling this case, where we first retrieve all those text segments T_js which are completely contained in P, and check if each such T_j can be extended for a type-3 matching. This is equivalent to the dictionary matching problem as described in Section 2.4.

The dictionary matching can be performed in $O(p(\log^{\epsilon} n + \log d) + \gamma)$ time using an $o(n \log \sigma) + O(d \log n)$ bits index, where γ is the number of dictionary matching outputs (Section 2.4) [10]. In [10], it is assumed that the text segments are stored using Ferragina-Venturini scheme [7], where the storage space is $nH_h + o(n \log \sigma)$ bits and a text substring of length ℓ can be displayed in $O(\ell / \log_\sigma n + 1)$ time. However, we cannot afford to store the text segments using this scheme as it will double the index space. Since FM-index is already stored for the type-1 and type-2 matchings, we will use the same FM-index as the storage scheme for the text collection (even though retrieval time is slower). It remains to check how the dictionary matching time will be affected by using FM-index (as a storage scheme) instead of Ferragina-Venturini scheme. For this, we consider the following two facts: (i) The time for displaying a substring of

length $\ell < \log^{1+\epsilon} n$ using FM-index is $O(\log^{1+\epsilon} n)$ and that of Ferragina-Venturini scheme is $\Omega(1)$. Hence, FM-index is worse by at most a factor of $O(\log^{1+\epsilon} n)$. (ii) When $\ell \geq \log^{1+\epsilon} n$, FM-index takes $O(\ell)$ time, where as Ferragina-Venturini scheme takes $O(\ell/\log_\sigma n)$ time. Hence, FM-index is worse by a factor of $\Theta(\log_\sigma n)$. Therefore, by using FM-index as a storage scheme, the dictionary matching time can get worse by a factor $O(max(\log^{1+\epsilon} n, \log_\sigma n)) = O(\log^{1+\epsilon} n)$. We remark [4] that only the term $p \log^\epsilon n$ will get multiplied by $\log^{1+\epsilon} n$. Choosing $\epsilon' \geq 2\epsilon > 0$, we have the following lemma.

Lemma 4 *Dictionary matching can be performed in $O(p \log^{1+\epsilon'} n + \gamma)$ time by maintaining an $nH_h + o(n \log \sigma) + O(d \log n)$ bits index.*

To perform type-3 matching, we first use the above lemma to find all the γ occurrences of text segments within P. Corresponding to each such occurrence, we compute a pair (s, e) as follows: Let the text segment T_j has a match within P at position i (i.e. $T_j = P[i..(i + |T_j|] - 1)]$) and T_j starts at position j' in T. This match will be a part of a valid occurrence of P in T if and only if P starts at position $(j' - i + 1)$ within T. Then (s, e) for this match is given by $s = j' - i + 1$ and $e = j'$. Further, we obtain a sorted sequence of all γ pairs $(s_1, e_1), (s_2, e_2), (s_3, e_3), \ldots, (s_\gamma, e_\gamma)$ such that $s_k < s_{k+1}$ or $s_k = s_{k+1}$ and $e_k < e_{k+1}$ in $O(\gamma \log \gamma)$ time. Along with this γ pairs of values, we also maintain the suffix ranges corresponding to all suffixes of P and $\overleftarrow{P}$ in Δ_f and Δ_r respectively (using Lemma 1). Therefore, our query working space is $O((p + \gamma) \log n)$ bits.

We maintain two bit vectors $B_s[1..n]$ and $B_e[1..n]$ for marking the starting and ending positions of text segments within T, such that $B_s[i] = 1$ if $T[i] \neq \phi$ and $T[i-1] = \phi$ or $i = 1$, else 0, and $B_e[i] = 1$ if $T[i] \neq \phi$ and $T[i+1] = \phi$ or $i = n$, else 0. Using these two bit vectors, the staring position and ending position ($select_{B_s}(j)$ and $select_{B_e}(j)$, respectively) of any given text segment T_j can be computed in constant time (Section 2.1). Similarly the text segment T_j or wildcard group ϕ^{k_j} corresponding to a given position i in T can be computed in constant time ($j = rank_{B_s}(i)$ and $T[i]$ will be within a text segment if $i \leq select_{B_e}(j)$, else $T[i]$ will be a part of jth wildcard group). We also maintain two arrays $A_f[1..d]$ and $A_r[1..d]$, such that A_f stores the lexicographic ordering of T_j in Δ_f ($A_f[j] = k$ if T_j is the kth lexicographically smallest α-sampled suffix in

[4] In the paper by Hon et al. [10], the computation of the locus of all suffixes of P in their dictionary matching index takes $O(p \log^\epsilon n)$ time and reporting all the occurrences takes $O(p(\log^\epsilon n + \log d) + \gamma)$ time. Note that the pattern matching is needed only in the first step.

Δ_f) and A_r stores the lexicographic ordering of $\overleftarrow{T}_j$ in Δ_r ($A_r[j] = k$ if $\overleftarrow{T}_j$ is the kth lexicographically smallest α-sampled suffix in Δ_r) . Thus, given the suffix range of any pattern in Δ_f, in constant time we can check if this is a prefix of given text segment using A_f. Similarly, given the suffix range of the reverse of any pattern in Δ_r, in constant time we can check if this is a prefix of given text segment using A_r. Now for a pattern P to match at position s in T (i.e. $P = T[s..(s+p-1)]$), the following conditions should be satisfied:

1. Corresponding to each text segment T_j which is completely contained in $T[s..(s+p-1)]$, there should be a pair (s, e) with e being the staring position of the T_j within T, and $e = select_{B_s}(j)$.
2. The longest prefix of $T[s..(s+p-1)]$ without any wildcard should be a prefix of P. In other words, if $T[s] \neq \phi$ and $T[s-1] \neq \phi$ (i.e., a prefix $P[1..x]$ of P has a match with a suffix ($\neq T_{j'}$) of a text segment $T_{j'}$), $\overleftarrow{P[1..x]}$ should be a prefix of $\overleftarrow{T}_{j'}$ (i.e., $A_r[j']$ should be within the suffix range of $\overleftarrow{T}_{j'}$ in Δ_r) , where $j' = rank_{B_s}(s)$ and $x = select_{B_e}(j') - s + 1$.
3. The longest suffix of $T[s..(s+p-1)]$ without any wildcard should be a suffix of P. In other words, if $T[s+p-1] \neq \phi$ and $T[s+p] \neq \phi$ (i.e., a suffix $P[y..p]$ of P has a match with a prefix ($\neq T_{j''}$) of a text segment $T_{j''}$), $P[y..p]$ should be a prefix of $T_{j''}$ (i.e. $A_f[j'']$ should be within the suffix range of $T_{j''}$ in Δ_f), where $j'' = rank_{B_s}(s+p-1)$ and $y = select_{B_s}(j'') - s + 1$.

In each contiguous sublist of (s, e) with same s value, the first condition can be verified by scanning the corresponding e values and the remaining conditions can be verified in constant time. Thus, the time for filtering all type-3 occurrences from the sorted list of (s, e) values can be bounded by $O(\gamma)$. Thus, we have the following lemma.

Lemma 5 *By maintaining an $nH_h + o(n \log \sigma) + O(d \log n)$ bits index, all the type-3 occurrences of P in T can be reported in $O(p \log^{1+\epsilon'} n + \gamma \log \gamma)$ time, where $\epsilon' > 0$. The working space required is $O((p+\gamma) \log n)$ bits.*

Combining the results of type-1 (Lemma 2), type-2 (Lemma 3), and type-3 (Lemma 5) matchings, we have the following theorem.

Theorem 1 *There exists an index of size $nH_h + o(n \log \sigma) + O(d \log n)$ bits for wildcard matching, which can answer each online query for a pattern P of length p in $O(p(\log^{1+\epsilon'} n + \min(p, \hat{d}) \log d) + occ_1 \log^{1+\epsilon'} n + occ_2 \log^{\epsilon'} d + \gamma \log \gamma)$ time, for any fixed $\epsilon' > 0$. The working space required is $O((p+\gamma) \log n)$ bits.*

References

1. D. Belazzougui. Succinct Dictionary Matching With No Slowdown. In *CPM*, pages 88–100, 2010.
2. M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA, 1994.
3. Y. F. Chien, W. K. Hon, R. Shah, and J. S. Vitter. Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In *DCC*, pages 252–261, 2008.
4. R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary Matching and Indexing with Errors and Don't Cares. In *STOC*, pages 91–100, 2004.
5. P. Ferragina and G. Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.
6. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed Representations of Sequences and Full-Text Indexes. *ACM Transactions on Algorithms*, 3(2), 2007.
7. P. Ferragina and R. Venturini. A Simple Storage Scheme for Strings Achieving Entropy Bounds. *Theoretical Computer Science*, 372(1):115–121, 2007.
8. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
9. W. K. Hon, T. H. Ku, R. Shah, S. V. Thankachan, and J. S. Vitter. Faster Compressed Dictionary Matching. In *SPIRE*, pages 191–200, 2010.
10. W. K. Hon, T. W. Lam, R. Shah, S. L. Tam, and J. S. Vitter. Compressed Index for Dictionary Matching. In *DCC*, pages 23–32, 2008.
11. W. K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. On Entropy-Compressed Text Indexing in External Memory. In *SPIRE*, pages 75–89, 2009.
12. J. Kärkkäinen and E. Ukkonen. Sparse Suffix Trees. In *COCOON*, pages 219–230, 1996.
13. U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
14. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
15. Y. Nekrich. Orthogonal Range Searching in Linear and Almost-Linear Space. *Computational Geometry*, 42(4):342–351, 2009.
16. R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. *ACM Transactions on Algorithms*, 3(4), 2007.
17. T. W. Lam, W. K. Sung, S. L. Tam, and S. M. Yiu. Space-Efficient Indexes for String Matching With Don't Cares. In *ISAAC*, pages 846–857, 2007.
18. A. Tam, E. Wu, T. W. Lam, and S. M. Yiu. Succinct Text Indexing With Wildcards. In *SPIRE*, pages 39–50, 2009.
19. C. Thachuk. Succincter Text Indexing with Wildcards. In *CPM*, pages 27–49, 2011.
20. P. Weiner. Linear Pattern Matching Algorithms. In *FOCS*, pages 1–11, 1973.
21. J. Ziv and A. Lempel. Compression of Individual Sequences via Variable Length Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.