Compressed Index for Dictionary Matching[*]

Wing-Kai Hon
National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw

Tak-Wah Lam
University of Hong Kong, Hong Kong
twlam@cs.hku.hk

Rahul Shah
Louisiana State University, LA, USA
rahul@csc.lsu.edu

Siu-Lung Tam
University of Hong Kong, Hong Kong
sltam@cs.hku.hk

Jeffrey Scott Vitter
Purdue University, IN, USA
jsv@cs.purdue.edu

Abstract

The past few years have witnessed several exciting results on compressed representation of a string T that supports efficient pattern matching, and the space complexity has been reduced to $|T|H_k(T) + o(|T| \log \sigma)$ bits [8, 10], where $H_k(T)$ denotes the kth-order empirical entropy of T, and σ is the size of the alphabet. In this paper we study compressed representation for another classical problem of string indexing, which is called dictionary matching in the literature. Precisely, a collection $\mathcal{D}$ of strings (called patterns) of total length n is to be indexed so that given a text T, the occurrences of the patterns in T can be found efficiently. In this paper we show how to exploit a sampling technique to compress the existing $O(n)$-word index to an $(nH_k(\mathcal{D}) + o(n \log \sigma))$-bit index with only a small sacrifice in search time.

1 Introduction

The past few years have witnessed several exciting results on compressed representation of a string (or strings) to allow efficient pattern matching. That is, we want an index of a string T so that given any pattern P, we can locate the occurrences of P in T efficiently. Suppose that characters in T are chosen from an alphabet of size σ. An index of T is said to be *succinct* if it requires space proportional to the worst-case space complexity of T (i.e., $|T| \log \sigma$ bits);[1] furthermore, it is said to be a *compressed* index if it requires space proportional to the space of a compressed representation of T, measured by its zeroth-order entropy or even its kth-order entropy. The two entropy terms are denoted by $H_0(T)$ and $H_k(T)$, respectively, and it follows

[*] This work is supported in part by Taiwan NSC Grant 96-2221-E-007-082-MY3 (W. Hon), Hong Kong RGC Grant HKU 7140/06E (T. Lam), and US NSF Grant CCF–0621457 (R. Shah and J. S. Vitter).

[1]The base of the logarithm is 2 unless specified.

1068-0314/08 $25.00 © 2008 IEEE
DOI 10.1109/DCC.2008.62

that $H_k(T) \leq H_0(T) \leq \log \sigma$. Grossi and Vitter [11] and Ferragina and Manzini [7] are the pioneers in the research of compressed indexes for efficient pattern matching. Since then, their results have been further improved by themselves and others (e.g., [8, 10, 16]; see [15] for a survey). It is now possible to index a string T using $|T|H_k(T) + o(|T| \log \sigma)$ bits, while supporting pattern matching in $O(|P| \log \sigma)$ time. Note that these compressed indexes are also self-indexes in the sense that they can function without the original string T, and they can indeed reproduce T efficiently. It is also worth-mentioning that these indexes are sound practically; when used to index a human genome (abut 3 billion characters), these indexes occupy less than 2 gigabytes [13].

In this paper we consider another classical string matching problem called the dictionary matching problem [1,3,4], which is defined as follows. Let $\mathcal{D} = \{P_1, P_2, \ldots, P_d\}$ be a set of strings (called patterns below) over an alphabet of size σ. We want to build an index for $\mathcal{D}$ so that given any input text T, the occurrences of each pattern of $\mathcal{D}$ in T can be located efficiently. This problem arises naturally in the process of matching with a gene bank or a database of computer virus. Aho and Corasick [1] are the first to give an index using $O(n)$ words that supports the dictionary matching query in $O(|T| + occ)$ time, where n is the total number of characters in $\mathcal{D}$, and occ is the total number of occurrences. Over the past decade, research on dictionary matching has focused on the extension to the dynamic setting [2–4, 17]; the space complexity has not been improved until the recent work of Chan et al. [5], which assumes σ is a constant and gives an $O(n)$-bit index that can search a text T in $O(|T| \log^2 n + occ \log^2 n)$ time.

In this paper we make use of a sampling technique to compress an $O(n)$-word, suffix-tree-based index to an $O(n \log \sigma)$-bit index that can can search a text T in $O(|T| \log \log n + occ)$ time, or an $(o(n \log \sigma) + O(d \log n))$-bit index that can search in $O(|T|(\log^\epsilon n + \log d) + occ)$ time, where $\epsilon > 0$ is any constant. Note that the suffix-tree-based index as well as the new indexes all require the presence of the original patterns; thus we need to add $n \log \sigma$ bits to the space complexity in the worst case. Nevertheless, there exist compressed representations of a string which allow constant time retrieval of any of its character (precisely, constant time retrieval of any $O(\log_\sigma n)$ consecutive characters). Ferragina and Venturini [9] have provided one such representation. Thus, if we concatenate the patterns into a string $P_1 P_2 \cdots P_d$ and represent it using such a compressed scheme, the overall space is reduced to $nH_k(\mathcal{D}) + o(n \log \sigma) + O(d \log n)$, where $H_k(\mathcal{D}) = H_k(P_1 P_2 \cdots P_d)$. A summary of the dictionary matching results is shown below.

Space (bits)	Search Time	Reference
$O(n \log n)$	$O(\lvert T\rvert + occ)$	Aho-Corasick [1]
$O(n)$ [assume $\sigma = O(1)$]	$O(\lvert T\rvert \log^2 n + occ \log^2 n)$	Chan et al. [5]
$O(n \log \sigma)$	$O(\lvert T\rvert \log \log n + occ)$	this paper
$n \log \sigma + o(n \log \sigma) + O(d \log n)$	$O(\lvert T\rvert(\log^\epsilon n + \log d) + occ)$	this paper
$nH_k + o(n \log \sigma) + O(d \log n)$	$O(\lvert T\rvert(\log^\epsilon n + \log d) + occ)$	this paper

Organization of the paper: Section 2 gives the preliminaries and defines useful notations. In Section 3, we introduce a crucial problem called *prefix matching*, whose solution helps us obtain an efficient index for the original dictionary matching problem. Section 4 describes our main results. We conclude in Section 5.

2 Preliminaries

2.1 Locus of a String

Let $\Delta = \{S_1, S_2, \ldots, S_r\}$ be a set of r strings over an alphabet Σ of size σ. Let \$ and # be two characters not in Σ, whose alphabetic orders are, respectively, smaller than and larger than any character in Σ. Let $\mathcal{C}$ be a compact trie of the strings $\{S_1\$, S_1\#, S_2\$, S_2\#, \ldots, S_r\$, S_r\#\}$.[2] Then, each string $S_i\$$ or $S_i\#$ corresponds to a distinct leaf in $\mathcal{C}$, and each S_i corresponds to an internal node in $\mathcal{C}$. Also, each edge is labeled by a sequence of characters, such that for each leaf representing some string $S_i\$$ (or $S_i\#$), the concatenation of the edge labels along the root-to-leaf path is exactly $S_i\$$ (or $S_i\#$). For each node v, we use $path(v)$ to denote the concatenation of edge labels along the path from root to v.

Definition 1. *For any string Q, the* locus *of Q in $\mathcal{C}$ is defined to be the lowest node v (i.e., farthest from the root) such that $path(v)$ is a prefix of Q.*

For simplicity, we refer $\mathcal{C}$ to be the compact trie for Δ, despite its constituent strings are constructed by appending a special character to each string in Δ.

2.2 Suffix Tree and Dictionary Matching

The *suffix tree* [14,18] for a set of strings $\{S_1, S_2, \ldots, S_r\}$ is a compact trie storing all suffixes of each S_i. For each internal node v in the suffix tree, it is shown that there exists a unique internal node u in the tree, such that $path(u)$ is equal to the string obtained from removing the first character of $path(v)$. Usually, a pointer is stored from v to such a u; this pointer is known as the *suffix link* of v.

Given a set of patterns $\mathcal{D} = \{P_1, P_2, \ldots, P_d\}$, suppose that we store the corresponding suffix tree. Then, inside the suffix tree we mark each node v with $path(v) = P_i$ for some i; after that, each node stores a pointer to its nearest marked ancestor. Let T be any input text, and $T(j)$ be the suffix of T starting at the jth character. Immediately, we have the following:

Lemma 1. *Suppose that the locus of $T(j)$ in the suffix tree of $\mathcal{D}$ is found. Then, we can report all occ patterns which appear at position j in T using $O(1 + occ)$ time.*

Proof. A pattern P_i appears at position j of T if and only if it is a prefix of $T(j)$. Let u be the locus of T in the trie. Then, P_i is a prefix of T if and only if u has a marked ancestor v with $path(v) = P_i$. Thus, reporting all patterns which appear at position

[2] In fact, we can simply use the compact trie for $\{S_1, S_2, \ldots, S_r\}$; the modifications on the strings here are for the ease of later discussion.

j of T is equivalent to reporting all marked ancestors of u. The latter is done by repeatedly tracing pointers of the nearest marked ancestor, starting from u. $\square$

By utilizing the suffix links, Amir et al [4] showed that the locus of $T(j)$ for all j can be found in $O(|T|)$ time, based on a traversal in the suffix tree. Thus, we can apply Lemma 1 to answer the dictionary matching query in $O(|T|+occ)$ time. Nevertheless, the space for storing the suffix tree is $O(n \log n)$ bits.

2.3 String B-tree

String B-tree [6] is an external-memory index for a set of strings that supports various string matching functionalities. It assumes an external memory model that we can read or write a disk page of B words in one I/O operation. By setting $B = \Theta(1)$, string B-tree can be readily applied in the internal memory model.

Let $\{P_1, P_2, \ldots, P_r\}$ be a set of strings over an alphabet of size σ, where $P_1 \leq P_2 \leq \cdots \leq P_r$ lexicographically. Suppose that each string has length at most ℓ, and their total length is n. The following lemma is an immediate result from Theorem 1 in [6]:

Lemma 2. *Assume that the strings are stored separately. We can construct an index of size $O(r \log n)$ bits such that on any input T, we can find the largest i such that $P_i \leq T$ lexicographically, using $O(\ell / \log_\sigma n + \log r)$ time.*

2.4 Computation Model

We assume the standard word RAM with word-size $\Theta(\log n)$ bits as our computation model, where n is the input size of our problem. In this model, standard arithmetic or bitwise boolean operations on word-sized operands, and reading or writing $O(\log n)$ consecutively stored bits, can each be performed in constant time.

3 Prefix Matching for Patterns

Given a set of r patterns $P_1, P_2, \ldots, P_r$ over an alphabet of size σ, with the length of each pattern at most ℓ. Let n be the total length of these r patterns. Without loss of generality, we assume that $P_1 \leq P_2 \leq \cdots \leq P_r$ lexicographically. The *prefix matching* problem is to construct an index for the patterns, so that when we are given an input text T, we can report efficiently all patterns which are a prefix of T. Solving this problem can help us solve the original dictionary matching problem. In the following, we propose two such indexes. The first index works for the general case where ℓ can be arbitrarily large. The second index targets for the case $\ell \leq \log_\sigma n$ with improved matching time.

Later, in Section 4, we will explain how to reduce (part of) the original dictionary matching problem into a prefix matching problem. The first index can be applied to obtain a compressed nH_k-bit index for the dictionary matching problem, while the second index can be applied to speed up the query time when slightly more space ($O(n \log \sigma)$ bits) is allowed.

3.1 Index for General Patterns

The first index consists of three data structures, namely a compact trie, a string B-tree, and an LCP array. We store a compact trie $\mathcal{C}$ comprising the r patterns. Inside the compact trie, we mark each node v with $path(v) = P_i$ for some i; after that, each node stores a pointer to its nearest marked ancestor. Based on the same argument as we prove Lemma 1, we have the following:

Lemma 3. *Suppose that the locus of a string T in the compact trie $\mathcal{C}$ is found. Then, we can report all occ patterns which are prefix of T using $O(1 + occ)$ time.*

To facilitate finding the locus of T in the compact trie, we store a string B-tree for the r patterns. In addition, we store an LCP array which is defined as follows: Let π_i denote the longest common prefix of P_i and P_{i+1}, and let w_i be the node in the compact trie with $path(w_i) = \pi_i$. The LCP array is an array L such that $L[i]$ stores the length of the longest common prefix $|\pi_i| = |path(w_i)|$ and a pointer to w_i. By using the string B-tree only, we obtain the following result.

Lemma 4. *Among all r patterns, we can find one which shares the longest common prefix with T, and report the length of such a longest common prefix, using $O(\ell/\log_\sigma n + \log r)$ time.*

Proof. By Lemma 2 we can find the largest i such that P_i is at most T lexicographically, using $O(\ell/\log_\sigma n + \log r)$ time. Then, either P_i or P_{i+1} must be a string which shares the longest common prefix with T. Checking which string is the desired answer, and finding the length of the longest common prefix, can be done by comparing T with P_i and P_{i+1} in a straightforward manner. This requires an extra $O(\ell/\log_\sigma n)$ time. $\square$

Let P_i be the string reported in the above lemma which shares the longest common prefix with T. Let m be the length of such a longest common prefix. Then, the locus of T in the compact trie must be one of the following three cases:

Case 1: The node v that corresponds to P_i (i.e., $path(v) = P_i$);

Case 2: The node w_{i-1}, which corresponds to the longest common prefix of P_{i-1} and P_i;

Case 3: The node w_i, which corresponds to the longest common prefix of P_i and P_{i+1}.

To see why the above case division is correct, let u be the locus of T in the compact trie. By the choice of P_i, either one of the following cases happen: `(i)` $P_i \le T < P_{i+1}$, or `(ii)` $P_{i-1} \le T < P_i$, lexicographically. If it is the former case, then u must either be v or w_i. If it is the latter case, then u must either be v or w_{i-1}.

Now, to distinguish which is the case, we compare m with $|P_i|$, $|\pi_{i-1}|$, and $|\pi_i|$. If $m = |P_i|$, then it is Case 1. Else, we consider $|\pi_{i-1}|$ and $|\pi_i|$, and select one which is at most m (if both are at most m, select the larger one). If $|\pi_{i-1}|$ is selected, then it is Case 2. Otherwise, it is Case 3.

Thus, by using the string B-tree and the LCP array, we have:

Lemma 5. *We can find the locus of T in the compact trie in $O(\ell/\log_\sigma n + \log r)$ time.*

Suppose that the patterns are stored separately so that we can retrieve any consecutive t characters of any pattern in $O(1 + t/\log_\sigma n)$ time, for any t. Then, the space of the compact trie, the string B-tree, and the LCP array each takes $O(r \log n)$ bits. This gives the following theorem.

Theorem 1. *Given r patterns of total length n, with the length of each pattern at most ℓ. Suppose the patterns are stored separately. We can construct an $O(r \log n)$-bit index such that we can report every pattern which is a prefix of any input T in $O(\ell/\log_\sigma n + \log r + occ)$ time.*

3.2 Index for Very Short Patterns

When ℓ is at most $\log_\sigma n$, Theorem 1 implies that prefix matching can be done in $O(1 + \log r + occ)$ time. Here, we give an alternative index so that the time becomes $O(\log \log n + occ)$. The time is better when r is moderately large (say, $r = \sqrt{n}$).

Firstly, we observe that the bottleneck $O(\log r)$-term in the previous time bound comes from searching the string B-tree. The main purpose of this searching is to find out the largest pattern P_i which is at most T lexicographically. Now, by padding each P_i with sufficient \$ characters to make its length $\log_\sigma n$, we can consider each padded pattern as a bit string of length $\log_\sigma n \times \log \sigma = \log n$, which can in turn be considered as an integer of $\log n$ bits.[3] In this way, we have converted the r patterns into r integers. To search for the desired P_i, we extract the first $\log_\sigma n$ characters of T and consider it as an integer. (If $|T| < \log_\sigma n$, we pad sufficient \$ characters to make it $\log_\sigma n$-char long.) Then, the desired P_i is exactly the largest of the r integers whose value is at most T (this P_i is known as the *predecessor* of T).

Willard [19] has devised a y-fast trie data structure which takes $O(r \log n)$ bits of space and supports $O(\log \log n)$-time predecessor query. Combining this with the compact trie and LCP array in the previous subsection, we obtain the following theorem.

Theorem 2. *Given r patterns of total length n, with the length of each pattern at most $\log_\sigma n$. Suppose the patterns are stored separately. We can construct an $O(r \log n)$-bit index such that we can report every pattern which is a prefix of any input T in $O(\log \log n + occ)$ time.*

4 Compressed Indexes for Dictionary Matching

Now we show how to make use of the prefix matching index to build a compressed dictionary matching index. Let α be a *sampling factor* to be fixed later. We intend to build a suffix tree with only one node per α suffixes so that we can save space.

[3] Here, we assume implicitly that $\log \sigma$, $\log_\sigma n$ and $\log n$ are integers. This assumption can be removed easily with minor modifications, without affecting the main results.

The missing suffixes will be covered by more intensive searching with the help of Theorems 1 and 2.

For a string $S[1..s]$, we call every substring $S[1+i\alpha\,..\,s]$ (where $0 \leq i\alpha < s$) an α-sampled suffix of S. Let $\mathcal{D} = \{P_1, P_2, \ldots, P_d\}$ be the set of patterns in the dictionary matching problem. The core of our compressed index for $\mathcal{D}$ is the compact trie $\mathcal{C}$ storing all α-sampled suffixes of each pattern. In addition, we define the following for the compact trie:

- For an internal node v, if the length $|path(v)|$ is a multiple of α, then v is called a *regular* node. Otherwise, v is called an *irregular* node.
- Each node is associated with the nearest ancestor which is regular.
- Nodes which correspond to occurrence of a pattern (d nodes in total) are marked. Each node in $\mathcal{C}$ stores a pointer to its nearest marked ancestor.
- For each regular node v, it is easy to show that there is a unique node u with $path(u)$ equal to the string obtained from removing the first α characters of $path(v)$. We store a pointer from v to u, called the *regular link* of v.

For any string Q, we define the *regular locus* of Q in the compact trie $\mathcal{C}$ to be the lowest regular node v such that $path(v)$ is a prefix of Q. For each pattern P_i with $|P_i| = x \pmod{\alpha}$ for some x between 1 and $\alpha - 1$, the *residue* of P_i is defined to be its last x characters. Based on the above definitions, we have:

Lemma 6. *Let v be the regular locus of $T(j)$ in $\mathcal{C}$. Let ϕ be the length of $path(v)$. Then, a pattern P_i appears at position j of T if and only if one of the following cases occurs:*

- *the locus of P_i (i.e., the node u with $path(u) = P_i$) is v;*
- *the locus of P_i is a marked ancestor of v;*
- *v is the regular locus of P_i, and the residue of P_i is a prefix of $T(j + \phi)$.*

Note that in the third case, the locus of P_i in $\mathcal{C}$ must be an irregular marked node associated with u. Thus, the occurrence of all patterns appearing at position j can be found as follows:

1. Find the regular locus u of $T(j)$ in $\mathcal{C}$.
2. Report u if it is marked.
3. Report all marked ancestors of u by tracing pointers.
4. Report all irregular marked nodes associated with u, whose corresponding residue is a prefix of $T(j + \phi)$.

Consider $\alpha = \log_\sigma n$. For Step 2 and Step 3, reporting the occurrences can be done in $O(1 + occ_j)$ time, where occ_j denotes the number of patterns which appear at position j of T. For Step 4, it can be solved by storing a separate data structure of Theorem 2 for each regular node that is associated with irregular marked nodes. It remains to show how to find the regular locus of $T(j)$ efficiently.

Let us consider the rooted tree formed by linking each regular node in $\mathcal{C}$ to its associated regular nodes. Essentially, the tree formed is equivalent to a suffix tree for the original patterns, except that every α characters in the pattern are 'merged' into one big character. Then, finding regular locus of $T(j)$ in $\mathcal{C}$ can be directly reduced to finding locus of $T(j)$ in this suffix tree.

In addition, the regular links in $\mathcal{C}$ are equivalent to the suffix links in the suffix tree. As a result, we can utilize the regular links and apply Amir et al [4] traversal algorithm to find the regular locus of $T(j)$ for all $j = x \pmod{\alpha}$, for a particular x, together in $O(|T|/\alpha)$ time.[4] Thus, the regular locus of $T(j)$ for all j can be found applying the traversal algorithm α times, taking a total of $O(|T|)$ time.

In summary, we obtain the following lemma.

Lemma 7. *Let $\{P_1, P_2, \ldots, P_d\}$ be d patterns over an alphabet of size σ, with total length n. Suppose the patterns are stored separately in $n \log \sigma$ bits. We can construct an index taking $O(n \log \sigma) + O(d \log n)$ bits such that we can answer the dictionary matching query for any input T in $O(|T| \log\log n + occ)$ time.*

Proof. The searching of the regular locus u_j of $T(j)$ takes $O(|T|)$ time in total, for all j. For a particular j, we report part of the occurrences of patterns (that appear at position j of T) by tracing pointers, starting from u_j. Also, we report the remaining occurrences from the data structure of Theorem 2 for u_j. The total time for reporting is $O(|T| \log\log n + occ)$.

For the space complexity, the compact trie takes $O\left(\sum_{i=1}^{d} (|P_i|/\alpha + 1) \log n\right)$ bits, which is $O(n \log \sigma + d \log n)$ bits. For the data structures of Theorem 2 in the regular nodes (which have associated irregular marked nodes), they require $O(d \log n)$ bits in total. Thus, the total space is $O(n \log \sigma) + O(d \log n)$ bits. □

Notice that for patterns whose length is at most $0.5 \log_\sigma n$, we can just store them together using an ordinary suffix tree. The number of such (distinct) patterns is at most $O(\sqrt{n} \log n)$, and their total length is at most $O(\sqrt{n} \log^2 n)$. Thus, the suffix tree occupies $O(\sqrt{n} \log^3 n) = o(n)$ bits of space, while support dictionary matching of T in $O(|T| + occ)$ time. For the remaining patterns, there are at most $d' = O(n/\log_\sigma n)$ of them; these patterns can be stored using our core index in Lemma 7, taking $O(n \log \sigma) + O(d' \log n) = O(n \log \sigma)$ bits. Thus, we can restate the above lemma as follows:

Theorem 3. *Let $\{P_1, P_2, \ldots, P_d\}$ be d patterns over an alphabet of size σ, with total length n. Suppose the patterns are stored separately in $n \log \sigma$ bits. We can construct an index taking $O(n \log \sigma)$ bits such that we can answer the dictionary matching query for any input T in $O(|T| \log\log n + occ)$ time.*

Now, let us increase the sampling factor α from $\log_\sigma n$ to $\log^{1+\epsilon} n / \log \sigma$. We store similar data structures as before, except we replace each data structure of Theorem 2

[4]One minor technical point arises: Amir et al's traversal algorithm requires that when we have traversed to some node in the suffix tree, we can select in constant time which child node to be traversed next. For our case, we can do so by storing a perfect hashing table [12] in each regular node when $\alpha = \log_\sigma n$. Later, when α is set to $\log^{1+\epsilon} n / \log \sigma$, we replace the hashing table by a modified Patricia tree (adapted from Section 4.1 in [11]), where the length of all edges are a multiple of $\log_\sigma n$. As a result, the selection takes $O(\log^\epsilon n)$ time instead. Both schemes do not affect the overall space. Details are deferred to the full paper.

by a data structure of Theorem 1 for each regular node (with associated irregular marked nodes). Consequently, we can modify Lemma 7 and obtain the following theorem:

Theorem 4. *Let $\{P_1, P_2, \ldots, P_d\}$ be d patterns over an alphabet of size σ, with total length n. Suppose the patterns are stored separately in $n \log \sigma$ bits. We can construct an index taking $o(n \log \sigma) + O(d \log n)$ bits such that we can answer the dictionary matching query for any input T in $O\big(|T|(\log^{\epsilon} n + \log d) + occ\big)$ time.*

Proof. Finding the locus of all $T(j)$ is done in $O(|T| \log^{\epsilon} n)$ time. Reporting occurrences is done in $O\big(|T|(\log^{\epsilon} n + \log d) + occ\big)$ time. For the space, the compact trie takes $O\big(\sum_{i=1}^{d}(|P_i|/\alpha + 1) \log n\big)$ bits, which is $o(n \log \sigma) + O(d \log n)$ bits for $\alpha = \log^{1+\epsilon} n / \log \sigma$. For the data structures of Theorem 1 in the regular nodes (which have associated irregular marked nodes), they require $O(d \log n)$ bits in total. Thus, the total space is $o(n \log \sigma) + O(d \log n)$ bits. □

Finally, for the patterns which are originally stored separately in its raw form (i.e., using $n \log \sigma$ bits), it can be stored in $nH_k + o(n \log \sigma)$ bits for $k = o(\log_{\sigma} n)$ using the scheme proposed by Ferragina and Venturini [9], without affecting the time of retrieving characters from any pattern. This gives the following corollary.

Corollary 5. *For $k = o(\log_{\sigma} n)$, the space occupied by the patterns and the index in Theorem 4 is $nH_k + o(n \log \sigma) + O(d \log n)$ bits.*

5 Concluding Remarks

We have applied a simple sampling technique to compress the existing suffix-tree-based index for static dictionary matching, giving the first index whose space is measured in terms of the kth-order entropy of the indexed patterns. An interesting open problem will be: Can we extend the sampling technique to obtain a compressed index for the *dynamic* dictionary matching problem? Specifically, can the space of such index be measured in terms of $nH_k(\mathcal{D})$?

References

[1] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked Ancestor Problems. Technical Report RS-98-16, Basic Research in Computer Science (BRICS), 1998. Extended abstract appears in *FOCS'98*, pages 534–543.

[3] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic Dictionary Matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994.

[4] A. Amir, M. Farach, R. Idury, A. La Poutre, and A. Schaffer. Improved Dynamic Dictionary Matching. *Information and Computation*, 119(2):258–282, 1995.

[5] H. L. Chan, W. K. Hon, T. W. Lam, and K. Sadakane. Compressed Indexes for Dynamic Text Collections. *ACM Transactions on Algorithms*, 3(2), 2007.

[6] P. Ferragina and R. Grossi. Optimal On-Line Search and Sublinear Time Update in String Matching. *SIAM Journal on Computing*, 27(3):713–736, 1998.

[7] P. Ferragina and G. Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005. A preliminary version appears in FOCS'00.

[8] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed Representations of Sequences and Full-Text Indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[9] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proceedings of Symposium on Discrete Algorithms*, pages 690–696, 2007.

[10] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of Symposium on Discrete Algorithms*, pages 841–850, 2003.

[11] 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. A preliminary version appears in STOC'00.

[12] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic Dictionaries. *Journal on Algorithms*, 41(1):69–85, 2001.

[13] W. K. Hon, T. W. Lam, W. K. Sung, W. L. Tse, C. K. Wong, and S. M. Yiu. Practical Aspects of Compressed Suffix Arrays and FM-index in Searching DNA Sequences. In *Proceedings of Workshop on Algorithm Engineering and Experiments*, pages 31–38, 2004.

[14] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[15] G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys*, 39(1), 2007.

[16] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003. A preliminary version appears in ISAAC 2000.

[17] S. C. Sahinalp and U. Vishkin. Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm. In *Proceedings of Symposium on Foundations of Computer Science*, pages 320–328, 1996.

[18] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[19] D. E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.