Succinct Index for Dynamic Dictionary Matching*

Wing-Kai Hon[1], Tak-Wah Lam[2], Rahul Shah[3], Siu-Lung Tam[2], and Jeffrey Scott Vitter[4]

[1] National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw
[2] University of Hong Kong, Hong Kong
{twlam,sltam}@cs.hku.hk
[3] Louisiana State University, Louisiana, USA
rahul@csc.lsu.edu
[4] Texas A&M University, Texas, USA
jsv@tamu.edu

Abstract. In this paper we revisit the dynamic dictionary matching problem, which asks for an index for a set of patterns $P_1, P_2, \ldots, P_k$ that can support the following query and update operations efficiently. Given a query text T, we want to find all the occurrences of of these patterns; furthermore, as the set of patterns may change over time, we also want to insert or delete a pattern. The major contribution of this paper is the first succinct index for dynamic dictionary matching. Prior to our work, the most compact index is given by Chan *et al.* (2007), which is based on the compressed suffix arrays (Grossi and Vitter (2005) and Sadakane (2003)) and the FM-index (Ferragina and Manzini (2005)), and it requires $O(n\sigma)$ bits where n is the total length of patterns and σ is the alphabet size. We develop a dynamic succinct index using a different (and simpler) paradigm based on suffix sampling. The new index not only improves the space complexity to $(1 + o(1))n \log \sigma + O(k \log n)$ bits, but also the time complexity of the query and update operations. Specifically, the query and update operations respectively take $O(|T| \log n + occ)$ and $O(|P| \log \sigma + \log n)$ times, where occ is the number of occurrences.

1 Introduction

Given a pattern P and a text T finding all the occurrences of P in T has been a fundamental problem and has developed into a very mature research field. The earliest work involved developing algorithms to solve the problem in $O(|P|+|T|)$ time [14]. When the text remains relatively static but patterns keep changing, one would like to build an index on the text T and treat the patterns as queries. Data structures like suffix trees [16,20] and suffix arrays [15] achieve optimal

* 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).

Y. Dong, D.-Z. Du, and O. Ibarra (Eds.): ISAAC 2009, LNCS 5878, pp. 1034–1043, 2009.

query performance. The space for these structures was considered to be "linear" but this was only when measured in terms of number of words and in asymptotic sense. In the stricter information-theoretic sense (which measures space in bits), this could be $\Theta(\log n)$ times more than the optimal. Furthermore, the hidden constants in the asymptotic notions often make these indexes about 20 to 60 times bigger than the original text.

Recently, Ferragina and Manzini [10] and Grossi and Vitter [11] presented text indexes based on the concept of Burrows-Wheeler transform (BWT) [6] whose space bounds are very close to the size of the compressed text. This has evolved into a thriving research field with many new application-specific compressed/succinct indexes developed.

The *dictionary matching* problem is an orthogonal problem to the text indexing problem. Here, some number of patterns are given beforehand and then a text comes in as the query. We need to find which patterns appear in this query text and at which locations. Hence, the index is built on the set of patterns. More formally, the problem is defined as follows.

Index:	A set of patterns $P_1, P_2, ..., P_k$ with total n characters.
Query:	A text T of size $\|T\|$ characters.
Output:	For each P_i occurring in T, all locations ℓ where P_i matches T beginning at position ℓ.

In the dynamic version of the problem, we support two update operations, namely *insert*(P) and *delete*(P). These operations, respectively, insert a new pattern to the set and delete any of the existing patterns from the set.

Dictionary matching problem has a long history starting with Aho and Corasick in 1975 [1] who solved the problem optimally for the case of static patterns. Amir *et al.* [3] gave a solution for dynamic case where inserts and deletes of patterns are allowed. Their approach consists of constructing a generalized suffix tree of the patterns with suffix links. In particular, suffix links are exploited to avoid repeatedly matching the characters of T when different positions of T are examined for pattern occurrences.

However, a major problem with all the above solutions was that the index takes too much space. With the advent of the field of compressed data structures, it remained to be shown that a space-efficient index can be designed for dictionary matching. Also, the issue of dynamism was somewhat hard to achieve with some of the earlier compressed indexing solutions. Chan *et al.* [7] were the first to present $O(n\sigma)$ bit index to solve this problem. Their solution mainly relied on Compressed Suffix Arrays (CSA) [11,18] and the subsequent Compressed Suffix Tree (CST) [19] with ingenious extension of suffix link operations. However, this solution remained from optimal in space (it only achieves big-O term) usage.

In this paper, we take a different approach than CSA or BWT based indexes. Our approach is based on directly sparsifying suffix links and using only sampled suffixes. A similar approach, but with a rather different sampling criteria, was considered by Kärkkäinen and Ukkonen [13] to solve the text indexing problem.

1.1 Comparisons with Previous Results

The solution by Aho and Corasick for the static case required $O(n \log n)$ bits and answered the queries in optimal $O(|T| + occ)$ time. When the dynamic case was addressed by Amir *et al.*, they achieved $O(n \log n)$-bit index but their query and update times had an extra multiplicative factor of $O(\log n / \log \log n)$.

The first attempt to reduce index space for the dynamic dictionary matching problem was given by Chan *et al.* [7]. Their approach builds on compressed suffix arrays (CSA) and compressed suffix trees. They extend the compressed suffix tree representation to use suffix links (using LCA queries). However, their approach in a way uses CSA as a black-box tool and hence it not only remains complicated but also that it does not attain the best possible bounds. For the case of constant alphabet size (i.e., $\sigma = O(1)$), they achieve $O(n)$ bits index. However, their search and update times have an extra multiplicative factor of $O(\log^2 n)$ which comes from attempting to dynamize some of more sophisticated data structures underlying the CSA based approach.

Hon *et al.* [12] introduced the suffix sampling technique to obtain stronger bounds for the static version of this problem. Namely, they achieved $O(n \log \sigma)$ bits index with $O(|T| \log \log n + occ)$ time. They also showed that space bounds like $nH_h + o(n \log \sigma) + O(k \log n)$ are achievable if the $\log \log n$ multiplicative factor can be increased to $\log n$; here, H_h denotes the hth-order empirical entropy of the set of patterns.

In this paper, we build on suffix sampling technique of [12] to achieve the best known bounds for the dynamic dictionary matching problem. One of the consequences of suffix sampling is that it results in sparsification of suffix links which in turn results in the space savings. We also show that in our approach we can split the data structure into two parts: one part is where the compressed text can be stored separately and the other part is the indexing overhead. By choosing the sampling rate appropriately, we can arbitrarily reduce the second part of the data structure, achieving space very close to the one required for the compressed representation of the text. Our space requirement and query times are as given in Table 1. We also note that we can achieve the entropy-compressed bound like $nH_h + o(n \log \sigma) + O(k \log n)$ under the assumption that the generative model from which the dynamic pattern statistics are taken remains the same. Not only our method is simpler to understand (and implement) but it may also greatly enhance the understanding of the nature of this problem.

Table 1. Summary of Results

Result	Space (bits)	Query Time	Update Time
[3]	$O(n \log n)$	$O((\|T\| + occ) \log n / \log \log n)$	$O(\|P\| \log n / \log \log n)$
[7]	$O(n\sigma)$	$O((\|T\| + occ) \log^2 n)$	$O(\|P\| \log^2 n)$
this	$O(n \log \sigma)$	$O(\|T\| \log n + occ)$	$O(\|P\| \log \sigma + \log n)$
this	$(1 + o(1))n \log \sigma + O(k \log n)$	$O(\|T\| \log n + occ)$	$O(\|P\| \log \sigma + \log n)$

2 Preliminaries

2.1 Basic Notation

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_r\}$ 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 such that each string $S_i\$$ or $S_i\#$ corresponds to a distinct leaf 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. Note that for each S_i, there must be some internal node v_i such that $path(v_i) = S_i$.

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.*

2.2 Suffix Tree

The *suffix tree* [16,20] for a set $\mathcal{S}$ of strings $\{S_1, S_2, \ldots, S_r\}$ is a compact trie storing all suffixes of each $S_i\$$ and each $S_i\#$. It can be stored in $O(m \log m)$-bit space where $m = |\mathcal{S}|$ denote the total number of characters in the strings of $\mathcal{S}$. 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.

By utilizing the suffix links, the suffix tree can be updated according to the insertion or deletion of S_i in the set $\mathcal{S}$ with $O(|S_i| \log \sigma)$ time [9].[1] In addition, we can efficiently find the loci of all suffixes of any text T within the suffix tree in $O(|T| \log \sigma)$ time [3].

2.3 Review: Dictionary Matching with Suffix Trees

Let $\Delta = \{P_1, P_2, \ldots, P_k\}$ be the set of patterns that are currently stored in the collection. Let Σ be the alphabet, and σ be its size. Let $n = \sum |P_i|$ be the total characters of the patterns in Δ. Suppose that we store the suffix tree for Δ; also for each i, we mark the node v_i with $path(v_i) = P_i$. Then we have the following:

Lemma 1. *Let $T(j)$ denote the jth suffix of a text T and let u be the locus of $T(j)$ in the suffix tree of Δ. Then, P_i appears at position j in T if and only if the marked node v_i is an ancestor of u.*

In case the set of patterns is static, we can store a pointer in each node of the suffix tree, pointing to the nearest marked ancestor. Then by the previous lemma, we can answer the dictionary matching query in $O(|T| \log \sigma + occ)$ time,

[1] That is, we insert or delete all suffixes of S_i in the suffix tree.

since finding all loci of all suffixes of T can be done in $O(|T| \log \sigma)$ time. In case the set of patterns is dynamic, the above scheme of storing pointers does not work well, as in the worst case a single pattern update can cause many nodes to change their nearest marked ancestors. Nevertheless, Amir *et al.* [3] showed that with suitable maintenance of the marked ancestors, we can answer the dictionary matching query in $O((|T| + occ) \log n / \log \log n)$ time and we can update a pattern P in $O(|P| \log n / \log \log n)$ time.[2]

3 Towards Succinctness with Sparse Suffix Tree

A major problem with the existing suffix-tree-based solutions is the index space, requiring $O(n \log n)$ bits which can be $\Theta(\log n)$ times more than the storage of the patterns in the plain form. To achieve space reduction, our idea is to *selectively* sample one suffix per every d suffixes, and maintain a compact trie $\mathcal{C}$ from these sampled suffixes. Intuitively, the resulting trie is a suffix tree for the original patterns, when we imagine every d characters of a pattern are merged into a single meta-character.

Our query is answered analogously as in the original suffix tree scheme. Basically, when a text T is given, we shall locate T positions, say π_i for $i = 1$ to $|T|$, in the compact trie $\mathcal{C}$ which respectively represents the locus of $T[i..|T|]$. Because of the similarity of $\mathcal{C}$ and an ordinary suffix tree, finding the loci can be done efficiently by exploiting "suffix links". However, since each meta-character represents d original characters, the computation of loci will be done by d separate traversals in $\mathcal{C}$, where the jth traversal computes the loci of those suffixes $T[i..|T|]$ with $i \pmod d = j$. Afterwards, we report the occurrences by finding the marked ancestors of each locus, using the data structure in Section 4.

3.1 Implementation Details

Various performance tradeoffs can be obtained by varying the sampling factor d. We first consider the simple case where d is set to $0.5 \log_\sigma n$. In this case, the number of suffixes is reduced from n to at most $\lceil n/d \rceil + k$. Consequently, the compact trie $\mathcal{C}$ has $O(n/d + k)$ nodes, so that its space is $O((n/d + k) \times \log n) = O(n \log \sigma + k \log n)$ bits.

Recall that $\mathcal{C}$ is similar to an ordinary suffix tree; indeed, we can analogously define suffix link for each internal node v in $\mathcal{C}$, which is the node u such that $path(u)$ is the same as the string obtained by removal of first d characters of $path(v)$ (i.e., removal of its first meta-character). However, due to the effect of merging characters, the alphabet size has increased from σ to $\sigma^d = \sqrt{n}$.

When a query text T is given, our target is to obtain the locus of each suffix of T in $\mathcal{C}$. We may first treat T as a meta-text T' by blocking every d characters. Then, we can utilize the suffix links and find the loci of each suffix of T' in $O((|T|/d+1) \log n)$ time, since there are $O(|T|/d+1)$ meta-characters, each from

[2] When σ is not a constant, an additive $O(|T| \log \sigma)$ and $O(|P| \log \sigma)$ term will be added to the query time and update time, respectively.

an alphabet of $\sqrt{n}$. Note that these loci may not be the same as the loci of those $T[i..|T|]$ with $i(\bmod\ d) = 1$, but they are closely related. For instance, the locus of T can be at most d nodes further from the locus of T'. In general, the locus of each $T[i..|T|]$ with $i(\bmod\ d) = 1$ can be obtained in an extra $O(d \log \sigma) = O(\log n)$ time through traversal in $\mathcal{C}$. As a result, the loci of roughly $1/d$ of all suffixes of T are obtained. To find the other loci, we can repeat the procedure for $d-1$ times, where at the jth time we search $\mathcal{C}$ with the meta-text formed by blocking $T[j+1..|T|]$. This gives the following lemma.

Lemma 2. *When $d = 0.5 \log_\sigma n$, the compact trie $\mathcal{C}$ requires $O(n \log \sigma + k \log n)$ bits of space. On any input text T, the loci of all suffixes of T in $\mathcal{C}$ can be obtained in $O(|T| \log n)$ time.*

Next, we briefly discuss two ideas of further reducing the space terms. The first one is to reduce the $O(k \log n)$ terms, under tha natural assumption that all patterns in the set Δ are distinct. For this case, we shall classify patterns into two groups, one for those longer than d, the other for those with length at most d. The number of patterns, k_1, in the first group is at most n/d, and these patterns will be indexed by a compact trie $\mathcal{C}'$ using Lemma 2. The number of patterns, k_2, in the second group is at most $\Theta(\sqrt{n} \log_\sigma n)$, whose total length is at most $\Theta(\sqrt{n}(\log_\sigma n)^2)$; these patterns will be stored in an ordinary suffix tree $\mathcal{R}$, and requires only $o(n)$ bits of space. Once the loci of all suffixes of T are located in both trees, we can proceed as before to output the marked ancestors of these loci. We summarize the above discussion as follows:

Lemma 3. *Assuming patterns in Δ are distinct. When $d = 0.5 \log_\sigma n$, we can store the compact trie $\mathcal{C}'$ and the $o(n)$-bit suffix tree $\mathcal{R}$, in total $O(n \log \sigma)$ bits of space, such that on any input text T, the loci of all suffixes of T in $\mathcal{C}'$ and in $\mathcal{R}$ can be obtained in $O(|T| \log n)$ time.*

The second idea to reduce space is by raising the sampling factor d. In particular, we set $d = \log n \log_\sigma n$.[3] Then, we can immediately obtain a lemma similar to Lemma 2, such that the space of $\mathcal{C}$ is reduced to $o(n \log \sigma) + O(k \log n)$ bits and finding all loci is done in $O(|T| \log^2 n)$ time. The increased in time to find loci is due to the inefficiency in extending each of the "approximate" locus (obtained from searching T' in $\mathcal{C}$) to the true locus. In fact, each such extension can be reduced to the *prefix matching* problem in [12], which can be solved more efficiently using $O(d/\log_\sigma n + \log k) = O(\log n)$ time (see Lemma 4 of [12]).[4] The extra space required to support the reduction is $O(k \log n)$ bits in total. Thus, we have the following lemma:

[3] Due to the increase in d, we can no longer combine this idea with the first one; as a result, we do not classify short and long patterns, and the $O(k \log n)$ term reappears.

[4] The idea is to maintain an extra data structure, called String B-tree [9], to manage the marked nodes so that once we obtain an approximate locus, we can easily jump to the nearest marked ancestor of the true locus. Due to space limitation, we defer the details to the full paper.

Lemma 4. *When* $d = \log n \log_\sigma n$, *the compact trie* $\mathcal{C}$ *requires* $o(n \log \sigma) + O(k \log n)$ *bits of space. On any input text* T, *the loci of all suffixes of* T *in* $\mathcal{C}$ *can be obtained in* $O(|T| \log n)$ *time.*

4 New Approach for Dynamic Marked Ancestors

Let $\mathcal{C}$ be a rooted tree with m nodes, where some k nodes are marked. The dynamic marked ancestor problem is to index $\mathcal{C}$ so that on given any node v, we can report all the ancestors of v which are marked; in addition, the tree can be updated by insertion or deletion of nodes, and by marking or unmarking nodes. Existing solutions [3,2] are achieved by the reduction to parentheses maintenance problem. In the following, we use an alternative approach where we solve the problem via management of one-dimensional intervals.

4.1 Reduction for Semi-static Case: Intervals Management

When the structure of the tree is static, and the set of marked nodes is fixed, the marked ancestor problem can be easily and optimally solved, simply by maintaining a pointer in each node to its nearest marked ancestor. Nevertheless, we shall show a non-optimal solution, which acts as a stepping stone towards an efficient solution for the dynamic case.

First, we perform a pre-order traversal of the tree. Each node is assigned the order in which it is first visited as its label. For instance, the root has label 1 and its leftmost child has label 2. For each marked node v, let v' denote the last node visited in the subtree rooted at v; also, let L_v and $L_{v'}$ be their labels, respectively. It is easy to check that v is a marked ancestor of a node u if and only if the label of u falls in the interval $[L_v, L_{v'}]$.

Using the *interval tree*, we can maintain the k intervals corresponding to the k marked nodes in $O(k \log m)$ bits, such that for any node u with label L_u, we can report all *occ* intervals containing L_u in $O(\log k + occ)$ time; that is, we can find all marked ancestors of u in $O(\log k + occ)$ time.

In fact, if the tree structure is static, the above scheme can also handle marking or unmarking of a tree node. Each such operation simply corresponds to inserting or deleting an interval in the interval tree. For this semi-static case, we can apply the dynamic interval tree by Arge and Vitter [4], where each update can be done in $O(\log k)$ time, while the query time and the space requirement remain unchanged.

4.2 Reduction for Dynamic Case: Elastic Intervals Management

Note that the interval tree scheme cannot be directly used to handle the fully dynamic case. In particular, when a node is inserted or deleted in the tree,[5] it

[5] Here, node insertion includes the case where a node is inserted into the middle of an existing edge, thus splitting one edge into two edges. On the other hand, when a degree-1 internal node is deleted, we reverse the process so that its parent edge and its child edge will be merged to a single edge.

can cause the pre-order label of many nodes to change, which in turn can cause the intervals of many marked nodes to change.

However, observe that the relative order of the pre-order label of the existing nodes, before and after the updates, are not changed. This motivates us to represent each marked node v by an "elastic" interval (instead of a fixed interval when v is marked), where endpoints are represented by pointers to v and v', so that its interval can be flexibly changed according to the current ranks of v and v' in the tree.

Now, suppose that the *relative* rank of two nodes can be compared online in $f(m)$ time, where m is the number of nodes in the tree. Then the dynamic interval tree of Arge and Vitter can easily be adapted to support each update in $O(f(m) \log k)$ time and each query in $O(f(m)(\log k + occ))$ time. One simple solution is to overlay a balanced binary tree for the nodes so that the exact rank of any node can be computed in $O(\log m)$ time, thus comparison can be made in $O(\log m)$ time. A more complicated solution is by Dietz and Sleator [8] or by Bender *el.* [5], which is an $O(m \log m)$-bit data structure for maintaining order in a list of items. In this order-maintenance data structure, an item can be inserted into the list in $O(1)$ time when either its predecessor or its successor is given, while it can be deleted (freely) in $O(1)$ time; given two items, we can compare their rank in the list in $O(1)$ time. Thus, we can obtain a solution of dynamic marked ancestor by interval tree without any sacrifice in query efficiency.

Yet, there are two important points to note for using the final scheme. First, the insertion of a node v in a tree will require the knowledge of which node is v's predecessor or successor. This can be immediately done when v is the first child of its parent (so that its predecessor is known), or v is inserted in the middle of an existing edge (whose successor is known). However, it will be time-consuming in case v is the *last* child of its parent, in which case we may need to find its successor by traversing to the root and finding the first branch to the right. Thus, the position of where a node is inserted will greatly affect the time in updating.

Second, as the endpoints of the interval for a marked node v is now replaced by pointers to v and v', it will cause a serious problem if v' can be deleted while v is marked (in that case, the endpoint becomes undefined). To avoid this problem, whenever we mark a node v, we will create a *dummy* node $\hat{v}$ and insert it as the rightmost child of v; on the other hand, $\hat{v}$ will be deleted only when v becomes unmarked. As $\hat{v}$ will always be the last node visited in the subtree rooted at v, $\hat{v} = v'$ by definition, so that the interval of each marked nodes will always be well-defined.

5 All in a Nutshell

We are now ready to combine the sparse suffix tree (Section 3) and the dynamic marked ancestor data structures (Section 4) to see their overall performance.

When $d = 0.5 \log_\sigma n$ and assuming the patterns are distinct, we can solve the dictionary matching query as follows. Recall that we maintain a compact trie $\mathcal{C}'$ for long patterns (length longer than d) and a suffix tree $\mathcal{R}$ for short patterns (length at most d).

1. We locate the loci of all suffixes of T in $\mathcal{C}'$ in $O(|T| \log n)$ time. (Lemma 3)
2. Then, we apply the dynamic interval tree to report all marked ancestors of these $|T|$ loci in a total of $O(|T| \log n + occ_\ell)$ time, where occ_ℓ denote the number of occurrences of long patterns.
3. Next, we traverse the suffix tree in $O(|T| \log \sigma)$ time to locate the $|T|$ loci of all suffixes of T in $\mathcal{R}$.
4. Then, we use a brute-force method to report all marked ancestors of these $|T|$ loci in a total of $O(|T| \times d) = O(|T| \log n)$ time.

Thus, in total, $O(|T| \log n + occ)$ time is required.

To support the update when a pattern P is inserted, we perform the following. Firstly, when P is shorter than d, we add P and its suffixes into the suffix tree $\mathcal{R}$, using $O(|P| \log \sigma)$ time. After that, we mark the node v with $path(v) = P$, using $O(1)$ time. Otherwise, when P is long, we shall update the compact trie $\mathcal{C}'$ and the dynamic marked ancestor data structures as follows:

1. We first insert the $\lceil |P|/d \rceil$ suffixes of P into $\mathcal{C}'$, using $O((|P|/d + 1) \log n)$ time, by exploiting the suffix links. In addition, we will ensure that for each node inserted to the tree $\mathcal{C}$, if it is not inserted into the middle of some existing edge, then it will be inserted as the *first* child of its parent.
2. Then, for each node inserted, we find either its predecessor or its successor in the pre-order traversal in $O(1)$ time. Then, we make the corresponding change in the Dietz-Sleator order-maintenance data structure, using an extra $O(1)$ time per node. In total, this takes $O(|P|/d + 1)$ time.
3. Next, we mark the node v with $path(v) = P$ in $\mathcal{C}'$. This involves adding a dummy node $\hat{v}$ as the rightmost child of v. For this step, we find the successor of $\hat{v}$ in $\mathcal{C}'$ by traversing from $\hat{v}$ to the root, and finding the first branch to the right. This takes $O(|P|)$ time. After that, we update the order-maintenance data structure in $O(1)$ time. In total, adding $\hat{v}$ takes $O(|P|)$ time.
4. After that, we add the elastic interval corresponding to the marked node v to the dynamic interval tree. This step takes $O(\log k)$ time.

As the most time-consuming step is Step 1, pattern insertion can be supported in $O((|P|/d + 1) \log n) = O(|P| \log \sigma + \log n)$ time. To support pattern deletion, it can be done similarly (and more easily) with the above steps, using the same time bound. This gives the following theorem.

Theorem 1. *Suppose that the patterns in Δ are distinct. Then we can maintain an $O(n \log \sigma)$-bit index for Δ, such that on any given text T, a dictionary matching query can be answered in $O(|T| \log n + occ)$ time. Also, the index supports insertion or deletion of a pattern in Δ in $O(|P| \log \sigma + \log n)$ time.*

When $d = \log n \log_\sigma n$, answering a dictionary matching query will only involve the search in the compact trie $\mathcal{C}$ for $|T|$ loci, and subsequently finding the marked ancestors of each locus using the data structures of Section 4. In total, this can be done in $O(|T| \log n + occ)$ time. For updates due to pattern insertion or deletion, it can be done in similar time as the above.[6] This gives the following theorem.

[6] Though we will need to handle a single update in the String B-tree data structure, this can easily be done in $O(|P|)$ time. Details are deferred in the full paper.

Theorem 2. *Suppose that patterns in Δ are stored separately in $n \log \sigma$ bits. Then we can maintain an $o(n \log \sigma) + O(k \log n)$-bit index for Δ, such that dictionary matching query can be answered in $O(|T| \log n + occ)$ time. The index supports insertion or deletion of a pattern in $O(|P| \log \sigma + \log n)$ time.*

References

1. Aho, A., Corasick, M.: Efficient String Matching: An Aid to Bibliographic Search. Communications of the ACM 18(6), 333–340 (1975)
2. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked Ancestor Problems. In: Proceedings of Symposium on Foundations of Computer Science, pp. 534–544 (1998)
3. Amir, A., Farach, M., Idury, R., La Poutre, A., Schaffer, A.: Improved Dynamic Dictionary Matching. Information and Computation 119(2), 258–282 (1995)
4. Arge, L., Vitter, J.S.: Optimal External Memory Interval Management. SIAM Journal on Computing, 1488–1508 (2003)
5. Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two Simplified Algorithms for Maintaining Order in a List. In: Proceedings of European Symposium on Algorithms, pp. 152–164 (2002)
6. Burrows, M., Wheeler, D.J.: A Block-sorting Lossless Data Compression Algorithm. Tech Report 124, Digital Equipment Corporation, CA, USA (1994)
7. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed Indexes for Dynamic Text Collections. ACM Transactions on Algorithms 3(2) (2007)
8. Dietz, P.F., Sleator, D.D.: Two Algorithms for Maintaining Order in a List. In: Proceedings of Symposium on Theory of Computing, pp. 365–372 (1987)
9. Ferragina, P., Grossi, R.: The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. Journal of the ACM 46(2), 236–280 (1999)
10. Ferragina, P., Manzini, G.: Indexing Compressed Text. Journal of the ACM 52(4), 552–581 (2005)
11. Grossi, R., Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. SIAM Journal on Computing 35(2), 378–407 (2005)
12. Hon, W.-K., Lam, T.-W., Shah, R., Tam, S.-L., Vitter, J.S.: Compressed Index for Dictionary Matching. In: DCC 2008, pp. 23–32 (2008)
13. Kärkkäinen, J., Ukkonen, E.: Sparse Suffix Trees. In: Proceedings of International Conference on Computing and Combinatorics, pp. 219–230 (1996)
14. Knuth, D.E., Morris, J.H., Pratt, V.B.: Fast Pattern Matching in Strings. SIAM Journal on Computing 6(2), 323–350 (1977)
15. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing 22(5), 935–948 (1993)
16. McCreight, E.M.: A Space-economical Suffix Tree Construction Algorithm. Journal of the ACM 23(2), 262–272 (1976)
17. Overmars, M.H.: The Design of Dynamic Data Structures. LNCS, vol. 156. Springer, Heidelberg (1983)
18. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48(2), 294–313 (2003)
19. Sadakane, K.: Compressed Suffix Trees with Full Functionality. Theory of Computing Systems, 589–607 (2007)
20. Weiner, P.: Linear Pattern Matching Algorithms. In: Proceedings of Symposium on Switching and Automata Theory, pp. 1–11 (1973)