Algorithmica (2015) 72:515–538
DOI 10.1007/s00453-013-9863-3

Compressing Dictionary Matching Index via Sparsification Technique

Wing-Kai Hon · Tsung-Han Ku · Tak-Wah Lam · Rahul Shah · Siu-Lung Tam · Sharma V. Thankachan · Jeffrey Scott Vitter

Received: 21 January 2013 / Accepted: 26 December 2013 / Published online: 7 January 2014

Abstract Given a set $\mathcal{D}$ of patterns of total length n, the dictionary matching problem is to index $\mathcal{D}$ such that for any query text T, we can locate the occurrences of any pattern within T efficiently. This problem can be solved in optimal $O(|T|+occ)$ time by the classical AC automaton (Aho and Corasick in Commun. ACM 18(6):333–340, 1975), where occ denotes the number of occurrences. The space requirement is $O(n)$ words which is still far from optimal. In this paper, we show that in many cases, sparsification technique can be applied to improve the space requirements of the indexes for the dictionary matching and its related problems. First, we give a compressed index for dictionary matching, and show that such an index can be generalized to handle dynamic updates of $\mathcal{D}$. Also, we give a compressed index for approximate dictionary matching with one error. In each case, the query time is only slowed down by a polylogarithmic factor when compared with that achieved by the best $O(n)$-word counterparts.

This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W.K. Hon), HK RGC Grant HKU 7140/06E (T.W. Lam), and US NSF Grant CCF-1017623 (R. Shah and J.S. Vitter).
A preliminary version of this work appears in the following conference proceedings: the Proceedings of the IEEE Data Compression Conference (DCC), 2008 and 2011, and the Proceedings of the International Symposium on Algorithms and Computation (ISAAC), 2009.

W.-K. Hon (✉) · T.-H. Ku
National Tsing Hua University, Hsinchu City, Hsinchu, Taiwan, Republic of China
e-mail: wkhon@cs.nthu.edu.tw

T.-W. Lam · S.-L. Tam
The University of Hong Kong, Hong Kong, Hong Kong SAR

R. Shah · S.V. Thankachan
Louisiana State University, Baton Rouge, LA, USA

J.S. Vitter
The University of Kansas, Lawrence, KS, USA

Keywords Data compression · Dictionary matching · Text indexing · Sparsification technique

1 Introduction

Given a set $\mathcal{D} = \{P_1, P_2, \ldots, P_d\}$ of d patterns of total length n, the *dictionary matching* problem is to index $\mathcal{D}$ such that for any query text T, we can locate the occurrences of any pattern within T efficiently. This problem has many practical applications, such as locating genes in a genome, or detecting viruses in a computer program segments. The classical AC automaton proposed by Aho and Corasick [1] requires $O(n)$ words of index space in the worst case, and can answer a query in optimal $O(|T| + occ)$ time, where occ denotes the number of occurrences. Over the years, several advances were made to solve the dynamic version of this problem (where patterns can be inserted to or deleted from $\mathcal{D}$) [3–5]. Recently, there has been interests in making the index space compressed or succinct [8, 11, 22, 24]. Suppose that characters in $\mathcal{D}$ are chosen from an alphabet Σ of size σ. Without loss of generality, we shall assume that the alphabet is equal to the set $\{1, 2, \ldots, \sigma\}$ throughout the paper. An index of $\mathcal{D}$ is said to be *succinct* if it requires space at most the worst-case space complexity of $\mathcal{D}$, ignoring lower order terms (i.e., taking space $(1 + o(1))n \log \sigma$ bits in our case); furthermore, it is said to be a *compressed* index if, ignoring lower order terms, it requires space at most the space of a compressed representation of $\mathcal{D}$, measured by its zeroth-order entropy or even its kth-order entropy. The two entropy terms are denoted by H_0 and H_k, respectively, and it follows that $H_k \leq H_0 \leq \log \sigma$. In this paper, we propose a compressed index for the dictionary matching and a succinct index for the dynamic dictionary matching.

We can generalize the dictionary matching problem to the *approximate dictionary matching* problem by relaxing the definition of an occurrence. Precisely, in approximate dictionary matching with k errors, we consider P occurs at position i in T whenever there is some substring $T[i..j]$ such that the edit distance between $T[i..j]$ and P is at most k. The *edit distance* between a string X and a string Y is the minimum number of edit operations (i.e., substitution, insertion, or deletion, of a character) to transform X into Y. In this paper, we focus on the case where $k = 1$, and give a compressed index for approximate dictionary matching with one error.

A common way to solve dictionary matching problem in the uncompressed $O(n)$-word space is to apply suffix tree [27, 31] as a tool to support efficient searching and updates. In this paper, we apply a simple technique called *sparsification* to reduce the space of the suffix tree. Intuitively, a suffix tree attempts to store all possible suffixes of each pattern in $\mathcal{D}$, while sparsification selects only a subset of these suffixes so as to achieve space reduction. Such a technique can be dated back since Kärkkäinen and Ukkonen's paper in [25], and has been successfully applied in obtaining compressed indexes for other string matching problems [12, 23]. The major challenge is to design new querying strategies to overcome the absence of the suffixes, while keeping the searching time as close to the optimal as possible. Other than that, the proposed indexes are all conceptually simple (as we are still working with an uncompressed suffix tree, though sparsified), and thus can be easily implemented in practice.

Table 1 Existing dictionary matching indexes

Index	Space (bits)	Query time
[1]	$O(n \log n)$	$O(\|T\| + occ)$
[11]	$O(n\sigma)$	$O(\|T\| \log^2 n + occ \log^2 n)$
This	$O(n \log \sigma)$	$O(\|T\| \log\log n + occ)$
This	$nH_k + o(n \log \sigma) + O(d \log n)$	$O(\|T\|(\log^{\epsilon} n + \log d) + occ)$
[8]	$nH_0(\mathcal{D}) + O(n)$	$O(\|T\| + occ)$
[24]	$nH_k(\mathcal{D}) + O(n)$	$O(\|T\| + occ)$

The organization of this paper is as follows. Section 1.1 reviews the existing related indexes for the dictionary matching, dynamic dictionary matching, and approximate dictionary matching problems. Section 2 gives the preliminaries and defines useful notation. In Sect. 3, we describe our compressed index for the dictionary matching problem, while in Sect. 4, we give our succinct index for the dynamic version of the problem. In Sect. 5, we show our compressed index for the approximate dictionary matching problem, for the case when $k = 1$. We conclude the paper in Sect. 6.

1.1 Comparisons with Previous Results

The dictionary matching problem can be solved in optimal $O(|T| + occ)$ time by the classical AC automaton [1], where occ denotes the number of occurrences. Alternatively, we may store the (generalized) suffix tree for the patterns in $\mathcal{D}$, and the query time remains $O(|T| + occ)$. Both indexes require $O(n)$ words in the worst case. Chan et al. [11] showed that the index space can be reduced to $O(n\sigma)$ bits, while query is slowed down by a factor of $O(\log^2 n)$.[1] In their scheme, space reduction is achieved by compressing the suffix tree directly, based on various elegant, but complicated, tools for compressed text indexing (e.g., the compressed suffix arrays [20], the FM-index [16], the compressed suffix tree [30]). In contrast, we show that the sparsification technique, despite its simplicity, is capable to give a compressed index for the dictionary matching, which simultaneously improve both the space requirement and query time.

Very recently, Belazzougui [8] successfully compressed the AC automaton directly, using the tools for compressed indexing, and obtained the first nH_0-bit index (ignoring lower-order space term) with optimal $O(|T| + occ)$ query time. Hon et al. [24] later adapted Belazzougui's index to reduce the space to nH_k, while keeping the query time optimal. Although our index does not give as good a performance as the indexes of [8] and [24], we still explain our index in details, due to its simplicity and its applicability in handling dynamic updates (where the latter is not yet possible with the indexes of [8] or [24]). A summary of the above indexes for the dictionary matching problem is shown in Table 1.

For dynamic dictionary matching, Amir et al. gave an $O(n \log n)$-bit index that can answer a query in $O((|T| + occ) \log n / \log\log n)$ time, while updates (insertion or

[1] In fact, Chan et al.'s index can support dynamic updates of $\mathcal{D}$.

Table 2 Existing dynamic dictionary matching indexes

Index	Space (bits)	Query time	Update time
[5]	$O(n \log n)$	$O((\|T\| + occ) \log n / \log\log n)$	$O(\|P\| \log n / \log\log n)$
[11]	$O(n\sigma)$	$O((\|T\| + occ) \log^2 n)$	$O(\|P\| \log^2 n)$
This	$(1 + o(1))n \log\sigma + O(d \log n)$	$O(\|T\| \log n + occ)$	$O(\|P\| \log\sigma + \log^2 n)$

Table 3 Existing approximate dictionary matching indexes (for $k = 1$)

Index	Space (bits)	Query time
[18]	$O(n^{1+\epsilon})$	$O(\|T\| \log\log n + occ)$
[6]	$O(n \log^3 n)$	$O(\|T\| \log^3 n \log\log n + occ)$
[13]	$O(n \log n + d \log d \log n)$	$O(\|T\| \log d \log\log d + occ)$
This	$nH_k + O(n) + o(n \log\sigma)$	$O(\sigma\|T\| \log^3 n \log\log n + occ)$

deletion of a pattern P) can be done in $O(|P| \log n / \log\log n)$ time. 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. Chan et al. [11] were the first to present $O(n\sigma)$-bit index to solve this problem, by extending various compressed text indexing tools for dynamic updates, and combining these tools to simulate the suffix tree. In this paper, we show that we can dynamize our compressed index, and solve the dynamic dictionary matching problem. A summary of the existing results is shown in Table 2.

For approximate dictionary matching, we focus on the case with one error in this paper ($k = 1$). The best-known index, in terms of query performance, is by Ferragina et al. [18], which requires $O(n^{1+\epsilon})$ words of space and answers a query in $O(|T| \log\log n + occ)$ time. Amir et al. [6] gave a suffix-tree-based index with a reduced space of $O(n \log^2 n)$ words, and showed how to reduce the query answering into a three-dimensional range reporting problem, so that the query time became $O(|T| \log^3 n \log\log n + occ)$. Later, Cole et al. [13] reduced the index space further to $O(n + d \log d)$ words, and simultaneously improved the query time to $O(|T| \log d \log\log d + occ)$. In this paper, we adapt Amir et al.'s index with the sparsification technique to obtain a compressed index for this problem. Nevertheless, we reduce the query answering into a *two-dimensional* range reporting problem instead, so as to control the space of the overall index. A summary of the existing approximate dictionary matching indexes for the case $k = 1$ is shown in Table 3.

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 set of strings $\{S_1\$, S_1\#, S_2\$, S_2\#, \ldots, S_r\$, S_r\#\}$. 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* [27, 31] 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. 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 [15].

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 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. □

By utilizing the suffix links, the locus of $T(j)$ for all j can be found based on a traversal in the suffix tree, in a total of $O(|T|(t_{child} + t_{slink}))$ time, where t_{child} denotes the time to access a child node in the suffix tree, and t_{slink} denotes the time to traverse a suffix link. Traversal of a suffix link takes $O(1)$ time if the suffix link is explicitly stored. In case the suffix tree is static (thus no updates in $\mathcal{D}$), $t_{child} = O(1)$ since we can maintain all children pointers of a node by perfect hashing [21], taking space linear to the number of children, and guaranteeing worst-case $O(1)$ time access. In general, when the suffix tree is dynamic, $t_{child} = O(\log \sigma)$, where we maintain the

children pointers by a balanced binary search tree of height $O(\log \sigma)$. We summarize the discussion in the following lemma.

Lemma 2 *The loci of all $T(j)$ in the suffix tree for $\mathcal{D}$ can be found in either $O(|T|)$ time if no updates of $\mathcal{D}$ are allowed, or $O(|T| \log \sigma)$ time if the suffix tree supports updating of $\mathcal{D}$; in the latter case, insertion or deletion of a pattern P in $\mathcal{D}$ can be done in $O(|P| \log \sigma)$ time, based on the algorithm in* [27].

2.3 Suffix Arrays

The *suffix array* $SA[1 \ldots n]$ is an array which stores the starting positions of all the suffixes when the suffixes are sorted in the lexicographical order [26]. In other words, $SA[i]$ is the starting position of the ith lexicographically smallest suffix (among all the n suffixes stored). We also define its inverse, SA^{-1}, to be an array such that $SA[i] = j$ if and only if $SA^{-1}[j] = i$. A suffix array takes $O(n \log n)$ bits of space and supports pattern matching query in $O(|P| + \log n + occ)$ time [26]. For any pattern P, it is known that all suffixes whose prefix matches exactly with P will correspond to a contiguous region in SA. The range of such a region is called the *suffix range* of P. By storing SA and its inverse, we have the following lemma.

Lemma 3 *Let $[\ell_1, r_1]$ and $[\ell_2, r_2]$ be the suffix ranges of patterns P_1 and P_2. Then the suffix range $[\ell, r]$ of $P_1 P_2$ (concatenation of P_1 and P_2) can be computed in $O(\log n)$ time.*

Proof Note that $\ell_1 \le \ell \le r \le r_1$ and the task is to find the range of i, such that $\ell_1 \le i \le r_1$ and $\ell_2 \le SA^{-1}[SA[i] + |P_1|] \le r_2$. This can be performed in $O(\log n)$ time by doing a binary search on i. □

Lemma 4 *Let T be the input text, and $T(j)$ be the suffix of T starting at the jth character. By preprocessing T initially in $O(|T| \log n)$ time in $O(|T| \log n)$ bits, then for any character c, the locus of $cT(j)$ in a suffix tree for any j can be answered in $O(\log^2 n)$ time, where $cT(j)$ is a string formed by the concatenation of c and $T(j)$.*

Proof For $i = 1, 2, 4, 8, \ldots$, we divide the text T into $|T|/i$ different blocks, each of length i, and find the locus of each block. This takes a total of $O(|T| \log n)$ time by Lemma 2. Now the locus of $cT(j)$ can be obtained as follows: Partition $T(j)$ into $O(\log n)$ maximal intervals, each starting and ending with a blocking boundary. Next, continuously add the partition of $T(j)$ to c, and compute the suffix range by Lemma 3. Once the suffix range is empty, we backtrack to get the longest prefix of $T(j)$ that can be added to c with a non-empty suffix range. Thus, $O(\log n)$ suffix ranges are computed, taking a total of $O(\log^2 n)$ time. The total space for storing the preprocessed data is $O(|T|)$ words, or $O(|T| \log n)$ bits. □

2.4 String B-Tree

String B-tree [15] 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 [15]:

Lemma 5 *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, and the smallest j such that $P_j \geq T$ lexicographically, using $O(\ell/\log_\sigma n + \log r)$ time.*

2.5 Centroid Path and Centroid Path Decomposition

Let Γ be a tree with n nodes. We define the *size* of an internal node v to be the number of leaves in the subtree rooted at v. Then the *centroid path* of the tree Δ is the path starting from the root, so that each node v on the path is the largest-size child of its parent (where ties are broken arbitrarily). The *centroid path decomposition* of the tree Γ is the operation where we decompose each off-path subtree of the centroid path recursively; as a result, the edges in Γ will be partitioned into disjoint centroid paths.

Lemma 6 ([13]) *The path from the root of Γ to v traverses at most $\log n$ centroid paths.*

2.6 Range Minimum Query (RMQ)

Let A be an array of length n. A *range minimum query* (RMQ) on A takes a query interval $[i, j]$ and the task is to return an index k such that $A[k] \leq A[x]$ for all $i \leq x \leq j$.

Lemma 7 ([19]) *By maintaining a structure of size $2n + o(n)$ bits over A, the RMQ query can be answered in constant time.*

2.7 Three-Sided Range Query Structure

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 $n \times n$ grid. Without loss of generality, we assume that $x_i \leq x_{i+1}$. A three-sided query on $\mathcal{R}$ is defined as follows: Given a query range $[x_\ell, x_r] \times [-\infty, y]$, report all points (x_i, y_i) such that $x_\ell \leq x_i \leq x_r$ and $y_i \leq y$.

Lemma 8 *By maintaining a data structure of size $O(n \log n)$ bits over $\mathcal{R}$, the three-sided query can be answered in $O(\log \log n + |output|)$, where $|output|$ denotes the output size.*

Proof For answering this query efficiently, we maintain two arrays X and Y, such that $X[i] = x_i$ and $Y[i] = y_i$. Further, we build a predecessor query structure [32] over X

and an RMQ structure of Lemma 7 over Y. Now, whenever a query comes, we first find the maximal range $X[\ell' \dots r']$ (using predecessor search in X in $O(\log\log n)$ time), such that $x_\ell \le x_{\ell'} \le x_{r'} \le x_r$. Now our task is to retrieve all points $Y[k] \le y$ such that $\ell' \le k \le r'$. This can be done by performing RMQ repeatedly as follows (where we are simulating the corresponding search in a priority search tree [28]). First we obtain the minimum value in this range (say $Y[k]$) and we check if $Y[k] \le y$. If so, we report this as an output and we recursively perform this query in the subrange $Y[\ell' \dots (k-1)]$ and $Y[(k+1) \dots r']$. Whenever RMQ returns a value which is greater than y, we stop recursing in that interval further. The total time can be bounded by $O(\log\log n + |output|)$, where $|output|$ denotes the output size. □

2.8 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 Compressed Index for Dictionary Matching

This section describes our compressed index for the dictionary matching problem. Apart from a sparsified suffix tree, we also make use of an auxiliary index that can answer *prefix matching* queries (see Sect. 3.1 for the definition), where the latter helps in locating the locus more quickly. In the following, we first introduce such an auxiliary index, and then we give the details of our dictionary matching index.

3.1 Prefix Matching for Patterns

Consider a set of r patterns $\{P_1, P_2, \dots, P_r\}$ 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 $P_1 \le P_2 \le \cdots \le P_r$ 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 $\ell \le \log_\sigma n$ with improved matching time.

Later, in Sect. 3.2, 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.

3.1.1 Index for General Patterns

The first index consists of three data structures, namely a compact trie, a string B-tree, and an LCP array (to be defined shortly). We store a compact trie $\mathcal{C}$ comprising the r patterns. Inside the compact trie, for each node, we store the length of its path label in the node. Also, 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 9 *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 10 *Among all r patterns, we can find the lexicographically smallest one and the lexicographically largest one, which share a longest common prefix with T, in $O(\ell/\log_\sigma n + \log r)$ time. The length of such a longest common prefix can also be reported.*

Proof We first compute the length m of the longest common prefix between T and the desired answers. By Lemma 5 we can find the largest j such that P_j is at most T lexicographically, using $O(\ell/\log_\sigma n + \log r)$ time. Then, either P_j or P_{j+1} must be a string which shares the longest common prefix with T. Checking which string is the desired one, and finding the length m, can be done by comparing T with P_j and P_{j+1} in a straightforward manner, with an extra $O(\ell/\log_\sigma n)$ time.

Once m is known, to obtain the lexicographically smallest answer, it is sufficient to apply Lemma 5 again to get the lexicographically smallest pattern that is larger than $T[1..m]\$$. Similarly, we can obtain the lexicographically largest answer by finding the lexicographically largest pattern that is smaller than $T[1..m]\#$. Both steps require an extra $O(\ell/\log_\sigma n + \log r)$ time. □

Let P_x and P_y be, respectively, the lexicographically smallest and largest strings reported in the above lemma which share the longest common prefix with T. Let m be the length of such a longest common prefix. Depending on whether the traversal of $T[1..m]$ in the compact trie ends at a node or in the middle of an edge, the locus of T in the compact trie will be one of the following two cases:

Case 1: The lowest common ancestor z of the node u that corresponds to P_x (i.e., $path(u) = P_x$) and the node v that corresponds to P_y (i.e., $path(v) = P_y$);
Case 2: The parent p of z.

The lowest common ancestor z may not be found directly using our current auxiliary data structures (though if we are willing to store an extra structure for reporting

lowest common ancestor [10], it can be done directly). Nevertheless, it can be seen that the parent p of z must either be w_{x-1} (which corresponds to the longest common prefix of P_{x-1} and P_x) or w_y (which corresponds to the longest common prefix of P_y and P_{y+1}). To distinguish which is the correct parent, we consider the lengths of their path labels $|\pi_{x-1}|$ and $|\pi_y|$, and select the node whose corresponding length is closer to m. Thus, we can find p by a constant number of accesses to the LCP array.

Once p, and the length m' of its path label, are known, we can find z by locating the p's child whose edge label starts with $T[m'+1]$. This can be done in $t_{child} = O(1)$ time by following the child pointer in the compact trie. Finally, to decide whether the locus of T belongs to Case 1 or Case 2, it is sufficient to check if the length of z's path label is exactly m. If so, it will be Case 1; otherwise, it will be Case 2.

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

Lemma 11 *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.*

3.1.2 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 smallest P_x which is at least T (and the largest pattern P_y 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. Here, we assume implicitly that $\log \sigma$, $\log_\sigma n$ and $\log n$ are integers. (This assumption can be removed by using $\lceil \log \sigma \rceil$, $\lfloor (\lceil \log n \rceil / \lceil \log \sigma \rceil) \rfloor$, and $\lceil \log n \rceil$, respectively, in the analysis. The query time, and the space of the lower-order terms, will only be affected by a constant factor, so that the asymptotic query time bound, and the overall space, will still hold.) In this way, we have converted the r patterns into r integers.

We maintain the set of r integers by Willard's y-fast trie data structure [32] which takes $O(r \log n)$ bits of space and supports $O(\log \log n)$-time *predecessor* and *suc-*

cessor queries.[2] To search for the desired P_x and P_y, we follow the same steps as before. We first find one of the patterns that shares the longest common prefix with T. This can be done by extracting the first $\log_\sigma n$ characters of T (padding \$ if T is too short) and consider it as an integer t. Then, either the pattern that corresponds to the predecessor of t, or the pattern that corresponds to the successor of t, will be one sharing the longest common prefix. After that, we can check which is the case, as well as the length m of such a longest common prefix. As all integers are $\log n$-bit patterns, the checking, and the reporting of m, can be done in $O(1)$ time by the standard table-lookup method, using an auxiliary data structure of size $o(n)$ bits.

Next, to get P_x, we pad $T[1..m]$ with \$ to get an integer t', and search for the successor of t' among the r integers to find its corresponding pattern. Similarly, to get P_y, we pad $T[1..m]$ with # to get an integer t'', and search for its predecessor in the set to find its corresponding pattern. In summary, the overall process requires constant number of predecessor and successor queries, and an extra $O(1)$ time to compute the length m. In addition, it requires constant number of word RAM operations to prepare the integers t, t', and t'', taking $O(1)$ time. Thus, the overall process can be done in a total of $O(\log\log n)$ time. Combining this with the subsequent searching 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.*

3.2 Details of 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, and we also call this suffix tree *sparse suffix tree*. 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 $0 \le i\alpha < s$) 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. We collect all α-sampled suffixes of each pattern. Then, for each such suffix, we block every α characters (starting from the beginning) into a single meta-character, until the number of remaining characters is between 1 and α. We call the remaining characters the *residue* of the suffix, which will be ignored temporarily. The core of our compressed index for $\mathcal{D}$ is a *sparse suffix tree*, which is a compact trie $\mathcal{C}$ storing all the blocked suffixes. Note that in this compact trie, there may be some degree-1 internal nodes, in case the path label of such a node is exactly a blocked suffix. Also, the length of the path label of each node is exactly a multiple of α. In addition, we define the following for this compact trie:

[2] A predecessor query reports the largest integer in the set that is at most the query integer, while a successor query reports the smallest integer in the set that is at least the query integer.

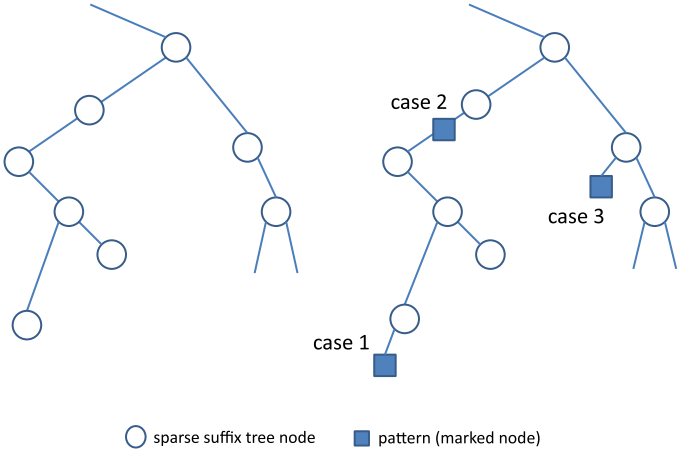

Fig. 1 An example of the sparse suffix tree. The *left side* shows the tree before adding the marked nodes. The *right side* shows the three possible cases of adding a marked node

- For each pattern P_i, a marked node is created in the trie so that the path label of the marked node is exactly P_i. Let v be the locus of P_i with its residue removed, which must exist as some node in the trie. There are three possible cases (see Fig. 1 for an illustration):

  Case 1: v is a leaf. In this case, we add the marked node as a child of v;
  Case 2: v is an internal node, and the residue of P_i is a prefix of the edge label of some child u of v. In this case, we add the marked node as a child of v, and the marked node is lying in the middle of the edge (v, u), with edge label equal to the residue of P_i.
  Case 3: v is an internal node, and the residue of P_i is not a prefix of edge label of any child of v. In this case, we add the marked node as a child of v.

  The marked node is said to be associated with the node v.
- Each node u in $\mathcal{C}$, whether marked or unmarked, stores a pointer to the nearest marked ancestor (i.e., the nearest marked node in the middle of the edges when we trace from u (inclusive) back to the root).
- For each unmarked 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 meta-character (i.e., the first α characters) of $path(v)$. We store a pointer from v to u, called the *sparse suffix link* of v.

For any string Q, we define the *unmarked locus* of Q in the compact trie $\mathcal{C}$ to be the lowest unmarked node v such that $path(v)$ is a prefix of Q. Based on the above definitions, we have:

Lemma 12 *Let v be the unmarked 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*:

1. *the locus of P_i is a marked ancestor of v*;
2. *v is the unmarked locus of P_i (i.e., v is associated with the marked node of P_i), and the residue of P_i is a prefix of $T(j+\phi)$.*

Based on the above lemma, the occurrence of all patterns appearing at position j can be found as follows:

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

Consider $\alpha = \log_\sigma n$. For Step 2, 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 3, it can be solved by storing a separate data structure of Theorem 2 for each unmarked node that has associated marked nodes. It remains to show how to find the unmarked locus of $T(j)$ in Step 1 efficiently.

Essentially, if we consider only the unmarked nodes, the compact trie $\mathcal{C}$ is equivalent to a suffix tree for blocked patterns, and the sparse suffix links are equivalent to the suffix links of such a suffix tree. As a result, we can utilize the sparse suffix links and apply Amir et al.'s [5] traversal algorithm to find the unmarked loci of $T(j)$ for all $j \equiv x \pmod{\alpha}$, for a particular x, in $O(|T|/\alpha+1)$ time. Thus, the unmarked loci of $T(j)$ for all j can be found applying the traversal algorithm α times, taking a total of $O(|T|+\alpha) = O(|T|)$ time (Here, we assume $|T| \geq \alpha$, as otherwise, the dictionary matching problem can be trivially be solved in $O(1+occ)$ time with the standard table-lookup method, requiring an extra $o(n)$-bit index space). In summary, we obtain the following lemma.

Lemma 13 *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 unmarked 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 $O(\sum_{i=1}^{d}(|P_i|/\alpha+1)\log n)$ bits, which is $O(n\log\sigma+d\log n)$ bits. For the data structures of Theorem 2 in the unmarked nodes (which have associated 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, and it supports 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 13, 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 by a data structure of Theorem 1 for each unmarked node that has associated marked nodes. In addition, one minor technical point arises: In the previous case where $\alpha = 0.5 \log_\sigma n$, we have implicitly assumed that the child pointers of each node in the trie are maintained by perfect hashing [21], so that $t_{child} = O(1)$. However, when α is set to $\log^{1+\epsilon} n / \log \sigma$, we can no longer use perfect hashing to maintain the child pointers, since each branching character is too long ($\omega(\log n)$ bits). Instead, we replace the hashing table by a modified Patricia trie (Lemma 9 from Sect. 4.1 in [20]). As a result, t_{child} becomes $O(\log^\epsilon n)$ time, and the required space of the modified Patricia trie is still linear (in words) to the number of children. Consequently, we can modify Lemma 13 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 $O(|T|(\log^\epsilon n + \log d) + occ)$ time.*

Proof Finding the locus of all $T(j)$ is done in $O(|T| \log^\epsilon n + \alpha) = O(|T| \log^\epsilon n)$ time (Here, we assume $|T| \geq 0.5 \log_\sigma n$, as otherwise the dictionary matching problem can be solved by table-lookup, as mentioned before). Reporting occurrences is done in $O(|T|(\log^\epsilon n + \log d) + occ)$ time. For the space complexity, the compact trie takes $O(\sum_{i=1}^{d} (|P_i|/\alpha + 1) \log n)$ 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 unmarked nodes (which have associated 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 [17], without affecting the time of retrieving characters from any pattern. This gives the following corollary.

Corollary 1 *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.*

4 Succinct Index for Dynamic Dictionary Matching

This section describes our succinct index for the dynamic dictionary matching problem. The design of the index is the same as that in Sect. 3, except that we replace the static component data structures into their dynamic counterparts. In the following, we first describe a new index for solving the dynamic marked ancestor problem, and after that, we give the details of our dynamic dictionary matching index.

4.1 New Approach for Dynamic Marked Ancestors

Let $\mathcal{I}$ be a rooted tree with m nodes, where some κ nodes are marked. The dynamic marked ancestor problem is to index $\mathcal{I}$ 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 [2, 5] 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. We first discuss the *semi-static* case, where the tree is static but we are allowed to mark or unmark an existing node in the tree. After that, we extend the idea to handle the *dynamic* case, where the tree structure can further be changed by node insertion or deletion.

4.1.1 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 κ intervals corresponding to the k marked nodes in $O(\kappa \log m)$ bits, such that for any node u with label L_u, we can report all occ intervals containing L_u in $O(\log \kappa + occ)$ time; that is, we can find all marked ancestors of u in $O(\log \kappa + occ)$ time. The space of the data structure is $O(\kappa \log m)$ bits.

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 [7], where each update can be done in $O(\log \kappa)$ time, while the query time and the space requirement remain unchanged.

4.1.2 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,[3] it 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 $O(f(m)\log\kappa)$ time and each query in $O(f(m)(\log\kappa + occ))$ 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 [14] or by Bender et al. [9], 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 ranks 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.

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 node will always be well-defined.

4.2 Details of Succinct Index for Dynamic Dictionary Matching

Dynamic dictionary matching problem can insert or delete a pattern, so we solve this problem by applying the indexes described in Sect. 3 without the perfect hashing

[3] 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.

tables. Recall that in the Sect. 3.2, we can use an ordinary suffix tree for short patterns (length shorter than $\alpha = 0.5 \log_\sigma n$), with space $o(n)$ bits, such that the loci of all suffixes of T can be obtained in $O(|T| \log \sigma)$ time. Also, we use a sparse suffix tree for the long patterns, with space $O(n \log \sigma)$ bits, such that the unmarked loci of all suffixes of T can be obtained in $O(|T| \log n)$ time. Here, we use these two indexes combined with the dynamic marked ancestor data structures (Sect. 4.1) to solve the dynamic dictionary matching.

When $\alpha = 0.5 \log_\sigma n$ and assuming the patterns are distinct, we can solve the dictionary matching query as follows.

1. We locate the unmarked loci of all suffixes of T in the sparse suffix tree in $O(|T| \log n)$ time.
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 denotes the number of occurrences of long patterns.
3. Next, we traverse the ordinary suffix tree to locate the $|T|$ loci of all suffixes of T and report all marked ancestors in total $O(|T| \log \sigma + occ_s)$, where occ_s denotes the number of occurrences of short patterns.

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 α, we add P and its suffixes into the ordinary suffix tree, 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 sparse suffix tree and the dynamic marked ancestor data structures as follows:

1. We first insert the $\lceil |P|/\alpha \rceil$ suffixes of P (with the residue removed) into the sparse suffix tree, using $O((|P|/\alpha + 1) \log n)$ time, by exploiting the suffix links. In addition, we will ensure that for each node inserted to the sparse suffix tree, 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 $O(|P|/\alpha + 1)$ time.
3. Next, we add the marked node v with $path(v) = P$ in the sparse suffix tree. 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 the sparse suffix tree 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 the dummy node $\hat{v}$ takes $O(|P|)$ time.
4. After that, we add the elastic interval corresponding to the marked node v (i.e., pointers to v and $\hat{v}$) to the dynamic interval tree. This step takes $O(\log d)$ time, where d is the current number of patterns in $\mathcal{D}$.

As the most time-consuming step is Step 1, pattern insertion can be supported in $O((|P|/\alpha + 1) \log n) = O(|P| \log \sigma + \log n)$ 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 5 *Suppose that the patterns in $\mathcal{D}$ 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 $\mathcal{D}$ in $O(|P| \log \sigma + \log n)$ time.*

We can increase α to $\log n \log_\sigma n$, so that the space of sparse suffix tree is further reduced to $o(n \log \sigma) + O(d \log n)$ bits. Finding all loci can be done in $O(|T|(t_{child} + t_{extend}))$ time, where t_{extend} denotes the time to extend the unmarked locus v of a suffix $T(j)$ to its true locus in the sparse suffix tree. Both t_{child} and t_{extend} can be bounded by $O(\alpha / \log_\sigma n + \log d) = O(\log n)$, if we maintain the child pointers, and the marked nodes of an unmarked node, by a String B-tree [15]. The total space required is, respectively, $o(n \log \sigma)$ bits and $O(d \log n)$ bits. For updates due to pattern insertion or deletion, it can be done in similar time as the above, though we will need to handle extra updates in the String B-tree data structures, which can be done in $O((|P|/\alpha + 1) \log^2 n) + O(|P|) = O(|P| \log \sigma + \log^2 n)$ time. This gives the following theorem.

Theorem 6 *Suppose that patterns in $\mathcal{D}$ are stored separately in $n \log \sigma$ bits. Then we can maintain an $o(n \log \sigma) + O(r \log n)$-bit index for $\mathcal{D}$, such that on any given text T, dictionary matching query can be answered in $O(|T| \log n + occ)$ time. Also, the index supports insertion or deletion of a pattern in $O(|P| \log \sigma + \log^2 n)$ time.*

5 Compressed Index for Approximate Dictionary Matching

This section describes our compressed index for the approximate dictionary matching problem. Our approach is to compress the index by Amir et al. [6]. In the following, we first introduce Amir et al.'s index and its query algorithm, and a simple trick to trade index space with query time. After that, we show how the compression is made to achieve our result.

5.1 Amir et al.'s Index

Amir et al. based on the following simple idea to design their index for the approximate matching problem with one error. Let S and P be two strings of length p. To decide whether S matches $P[1..p]$ with exactly one error, suppose $S[i] \neq P[i]$ for some $1 \leq i \leq p$. Then, we can check whether $S[1..i-1]$ is the same as $P[1..i-1]$ and $S[i+1..s]$ is the same as $P[i+1..p]$. If this is the case, S matches P with one error.

We now describe Amir et al.'s index. For a string S, we use S^R to denote the reverse of S. Given a set $\mathcal{D} = \{P_1, P_2, \ldots, P_d\}$ of d patterns, let $\mathcal{D}^R = \{P_1^R, P_2^R, \ldots, P_d^R\}$ be the set of patterns such that P_i^R is the reverse of P_i. Amir et al.'s index consists of a suffix tree $ST_{\mathcal{D}}$ for all the patterns in $\mathcal{D}$, and a suffix tree $ST_{\mathcal{D}^R}$ for all patterns in $\mathcal{D}^R$. In addition, all the nodes in $ST_{\mathcal{D}}$ and $ST_{\mathcal{D}^R}$ are renamed according to the centroid path decomposition, such that the nodes in the same centroid path have contiguous id values. They also maintain a three-dimensional $n \times n \times \sigma$ range

searching index which links the nodes $v \in ST_{\mathcal{D}}$, $u \in ST_{\mathcal{D}^R}$, and a character $c \in \Sigma$, whenever the concatenation of (i) the reverse of $path(u)$, (ii) c, and (iii) $path(v)$ is equal to some pattern $p_i \in \mathcal{D}$. Note that u and v are represented by the node id values which are renamed according to the centroid path decomposition. The approximate dictionary matching algorithm for the replacement error is given as follows, while the other types of edit errors (i.e., insertion or deletion) can be handled analogously.

Let $T = t_1 t_2 \dots t_{|T|}$. For $i = 1, \dots, |T|$ do

1. Find the locus node v of the longest prefix of $t_{i+1} \dots t_{|T|}$ in $ST_{\mathcal{D}}$.
2. Find the locus node u of the longest prefix of $t_{i-1} \dots t_1$ in $ST_{\mathcal{D}^R}$.
3. Report all patterns that match a substring of T with exactly one error, where such an error occurs at location t_i.

Steps 1 and 2 can be answered by Lemma 2 in $O(|T|)$ time. For Step 3, it can be modeled as a range query on the linking structure between all the ancestors of u, all the ancestors of v, and character $c \neq t_i$. Based on the property of centroid path decomposition and the naming convention for nodes in the suffix trees, all the ancestors of u, and all the ancestors of v, will be partitioned into $O(\log n)$ ranges. For those characters not equal to t_i, they will fall into the two ranges $[1, t_i - 1]$ and $[t_i + 1, \sigma]$, or equivalently the two ranges $[-\infty, t_i - 1]$ and $[t_i + 1, \infty]$. Thus, the query in Step 3 can be decomposed into $O(\log n) \times O(\log n) \times 2 = O(\log^2 n)$ 5-sided range queries, which can be answered in $O(|T| \log^3 n \log\log n + occ)$ time if we use the range searching index of Overmars [29]. For the space, the bottleneck comes from Overmars's index, which requires $O(n \log^2 n)$ words.

5.1.1 Trading Index Space with Query Time

We show how to reduce the above three-dimensional range searching problem into a series of two-dimensional range searching problems, so that we can achieve an index with smaller space. In particular, we use a two-dimensional range searching structure, such that the nodes $v \in ST_{\mathcal{D}}$ and $u \in ST_{\mathcal{D}^R}$ are linked if and only if the concatenation of (i) the reverse of $path(u)$ and (ii) $path(v)$ is equal to a pattern $p_i \in \mathcal{D}$. The query algorithm is modified as follows.

For $i = 1, \dots, |T|$ and for each character $c \in \Sigma$ and $c \neq t_i$ do

1. Find the locus node v of the longest prefix of $t_{i+1} \dots t_{|T|}$ in $ST_{\mathcal{D}}$.
2. Find the locus node u of the longest prefix of $ct_{i-1} \dots t_1$ in $ST_{\mathcal{D}^R}$.
3. Report all patterns that match a substring of T with exactly one error, where such an error occurs at location t_i, and the match occurs when t_i is replaced by c.

For Step 3 in the modified algorithm, we can model the reporting query as a range query on the linking structure for all the ancestors of u against all the ancestors of v. With similar arguments as before, the query can be split into $O(\log^2 n)$ four-sided range queries. In fact, a closer look reveals that the ancestors of v in each centroid path must appear as a prefix in the centroid path. Thus, if we partition the linking structure into separate linking structures, one for each centroid path in $ST_{\mathcal{D}}$, the range of the ancestors of v in a certain centroid path will correspond to a contiguous range starting from the beginning. Consequently, the $O(\log^2 n)$ four-sided queries on

the original linking structure will become $O(\log^2 n)$ three-sided queries on $O(\log n)$ different linking structures, where each query can be answered efficiently using the structures of Lemma 8 described in Sect. 2.7. This gives the following theorem.

Theorem 7 *Approximate dictionary matching query with one error can be performed in $O(\sigma|T|\log^2 n \log\log n + occ)$ by maintaining an $O(n)$-word index.*

5.2 Compressed Approximate Dictionary Matching with One Error

We now describe our index for compressed approximate dictionary matching with one error. Our approach is to answer a query with the idea in Sect. 5.1.1, and construct a corresponding compressed index with sparsification technique. We handle long patterns ($|P_i| \geq \log n$) and short patterns ($|P_i| < \log n$) separately using two different indexes.

5.2.1 Handling Long Patterns

In the following, we shall adopt the definitions of residue and sparse suffix tree, and the related notion, from Sect. 3.2. Given a sampling factor α and a pattern P, we use $trunc(P)$ to denote the pattern formed by removing the residue of P. Let $trunc(\mathcal{D})$ denote the set of patterns $\{trunc(P_1), \ldots, trunc(P_d)\}$, and $trunc(\mathcal{D})^R$ denote the corresponding set of reverse patterns.

We shall set $\alpha = \log n$, and maintain two sparsified suffix trees. The first one is the sparse suffix tree Δ_F for $\mathcal{D}$, while the second one is an ordinary suffix tree Δ_R for the 'blocked' patterns in $trunc(\mathcal{D})^R$ (every α characters in each pattern is blocked into a single meta-character). Intuitively, these two suffix trees correspond to the two suffix trees used in Amir et al.'s index. To facilitate the query algorithm, all pattern residues associated to an unmarked node in the sparse suffix tree for $\mathcal{D}$ are maintained by a Patricia trie, and we maintain a linear-space index [10] to support constant-time lowest common ancestor (LCA) query.

To find all 1-error pattern matches within an input text T, we classify the matches into two groups. The first one contains those matches with error occurring in the 'blocked' part of the pattern, while the second one contains those matches with error occurring in the residue part of the pattern. To obtain the matches in the first group, our method is based on the following observation:

Observation 1 *For a string S, let $\zeta(S)$ denote the maximal suffix of S whose length is a multiple of α, and $\rho(S)$ denote the maximal prefix of S whose length is a multiple of α. Suppose that a pattern P matches a substring $T[j..j+|P|-1]$, with exactly one replacement error inside the ith block of P. Then* (i) *the prefix of P of length $(i-1)\alpha$ is a suffix of $\zeta(T[1..j+(i-1)\alpha-1])$,* (ii) *the ith block of P has one mismatch with $T[j+(i-1)\alpha..j+i\alpha-1]$, and* (iii) *the remaining suffix of P is a prefix of $\rho(T[j+i\alpha..|T|])$.*

We shall use the framework, as shown in Algorithm 1, to find all the matches in the first group. Now, we describe the details of each step in Algorithm 1.

Algorithm 1 Finding all matches in the first group

For $j = 1, \ldots, \alpha$ do
For $i = 1, \ldots, |T|/\alpha$ do
Change each character in $t_{j+(i-1)\alpha} \ldots t_{j+i\alpha-1}$ for σ times. For each change, let β be the meta-character obtained, and we do the following: `/* executed with` $\sigma\alpha$ `different` β`s for each combination of` i `and` j `*/`

1. Find the locus node v of $t_{j+i\alpha} \ldots t_{|T|}$ in Δ_F.
2. Find the locus node u of $\beta^R t_{j+(i-1)\alpha-1} \ldots t_1$ in Δ_R.
3. Report all patterns that match a substring of T with exactly one error, where such an error occurs at the non-residue of the pattern, and the match occurs when the substring $t_{j+(i-1)\alpha} \ldots t_{j+i\alpha-1}$ is replaced by β.

- For each round j, Step 1 can be performed in a total of $O((|T|/\alpha + 1)\log(\sigma^\alpha)) = O((|T| + \alpha)\log\sigma)$ time, based on the result of Lemma 2. Thus, the overall time spent in Step 1 is $O((|T| + \log n)\log n \log\sigma)$ time.
- For each particular β, Step 2 can be performed by first finding the suffix range of β^R in Δ^R, using $O(\alpha \log n) = O(\log^2 n)$ time, and then apply the result of Lemma 4 to obtain the desired locus, in $O(\log^2 n)$ time. Since we need to preprocess T before applying Lemma 4, each round j requires an extra $O(|T|\log n)$ preprocessing time, and $O(|T|/\alpha + 1)$ words of working space. Thus, Step 2 can be performed in an overall of $\alpha \times |T|/\alpha \times O(\sigma\alpha) \times O(\log^2 n) + O(|T|\log n)$ time, which is bounded by $O(\sigma|T|\log^3 n)$.
- For Step 3, we use the same scheme as in Sect. 5.1,which takes $O(\log^2 n \log\log n + |output|)$ time for each β in each subround i. Thus, Step 3 can be performed in an overall of $\alpha \times |T|/\alpha \times O(\sigma\alpha) \times \log^2 n \log\log n + O(occ)$ time, which is bounded by $O(\sigma|T|\log^3 n \log\log n + occ)$.

For the index space, the sparse suffix trees require $O(\sum_{i=1}^{d}(|P_i|/\alpha + 1)\log n) = O(n)$ bits, while the range searching structure for Step 3 also needs $O(\sum_{i=1}^{d}(|P_i|/\alpha + 1)\log n) = O(n)$ bits.

To find all the matches in the second group, we shall use the framework as shown in Algorithm 2. This algorithm is very similar to Algorithm 1, and it is easy to show that they can be implemented within the same time complexity. The index space and working space complexity is also the same as we use the same structures. This gives the following theorem.

Theorem 8 *Let $\mathcal{D}_1 \subseteq \mathcal{D}$ be the subset of all long patterns in $\mathcal{D}$, where each pattern $P \in \mathcal{D}_1$ has length at least α. Let $\sum_{P\in\mathcal{D}_1} |P| = n_1 \leq n$. Then $\mathcal{D}_1$ can be indexed in $n_1 H_k(\mathcal{D}_1) + o(n\log\sigma) + O(n)$ bits of space, such that all the occurrences of patterns $p \in \mathcal{D}_1$ which appear as a substring of an online text T with one error can be reported in $O(\sigma|T|\log^3 n \log\log n + occ)$ time.*

5.2.2 Handling Small Patterns

In order to handle short patterns, we maintain an index for exact dictionary matching for the set of all short patterns ($|P_i| \leq \alpha = \log n$). Here we choose the $nH_k(\mathcal{D}) +$

Algorithm 2 Finding all matches in the second group

For $j = 1, \ldots, \alpha$ do
For $i = 1, \ldots, |T|/\alpha$ do
Change each character in $t_{j+i\alpha} \ldots t_{j+(i+1)\alpha-1}$ for σ times. For each change, let β be the meta-character obtained, and we do the following: `/* executed with` $\sigma\alpha$ `different` β`s for each combination of` i `and` j `*/`

1. Find the locus node v of β in Δ_F.
2. Find the locus node u of $t_{j+i\alpha-1} \ldots t_1$ in Δ_R.
3. Report all patterns that match a substring of T with exactly one error, where such an error occurs at the residue of the pattern, and the match occurs when the substring $t_{j+i\alpha} \ldots t_{j+(i+1)\alpha-1}$ is replaced by β.

   `/* The reporting is done by the same scheme as in Sect. 5.1. However, to avoid reporting the same occurrence repeatedly, we ensure that only the marked ancestors of` v `in` Δ_F`, whose path label is long enough to contain the changed character, are considered. As` v `has` $O(\alpha)$ `ancestors, the desired marked nodes can be found in` $O(\alpha)$ `time directly. */`

$O(n)$-bit index by Hon et al. [24] which can perform dictionary matching in optimal $O(|T| + occ)$ time. Our approach is again by substitution. First we block the text T into overlapping intervals $T_1, T_2, \ldots$ of length 2α, where $T_i = T[1 + (i-1)\alpha, \ldots, (i+1)\alpha]$. Then we obtain a set of new blocks by introducing an error in each position of each of these blocks. The total number of such new blocks (with one error) is $O(\sigma\alpha(|T|/\alpha)) = O(\sigma|T|)$. Now all these new blocks can be checked separately to report the matches. However, this approach creates a small problem as all the reported occurrences need not be an approximate match (some can be exact matches as the introduced error need not be within an occurrence). However the number of occurrences within a block is bounded by $\binom{2\alpha}{2} = O(\alpha^2)$, hence the total number of outputs can be bounded by $O(\sigma|T|\alpha^3) = O(\sigma|T|\log^3 n)$. Among all outputs, those which match with an error position can be reported as the actual outputs.

Theorem 9 *Let $\mathcal{D}_2 \subseteq \mathcal{D}$ be the subset of all short patterns in $\mathcal{D}$, where each pattern $P \in \mathcal{D}_2$ has length less than α. Let $\sum_{P\in\mathcal{D}_2} |P| = n_2 \leq n$. Then $\mathcal{D}_2$ can be indexed in $n_2 H_k(\mathcal{D}_2) + O(n_2)$ bits space, such that all the occurrences of patterns $P \in \mathcal{D}_2$ which appear as a substring of an online text T with one error can be reported in $O(\sigma|T|\log^3 n)$ time.*

By combining Theorems 8 and 9, we obtain the following result. Note that H_k is the kth order empirical entropy of the complete set of patterns.

Theorem 10 *Approximate dictionary matching with one error can be solved in $nH_k + O(n) + o(n\log\sigma)$ bits space and $O(\sigma|T|\log^3 n \log\log n + occ)$ query time.*

For the discussion so far, we have assumed that the error in a match comes as a replacement error. It is straightforward to modify the above query algorithms, using the *same* index and *same* query time, to report matches with one insertion or one deletion error. We omit the details for brevity.

6 Conclusion

We have shown that the sparsification technique can be applied to improve the space requirements of the indexes, to nearly optimal, for the dictionary matching and two of its related problems. The indexes are mainly based on the sparsified version of the suffix tree, which is conceptually very simple; tools that are technically involved are mainly used in maintaining the child pointers to speed up the access time, and in maintaining the residues of the patterns. For practical consideration, these tools may be replaced with the slower but simpler alternatives (such as non-perfect hashing or balanced binary search tree) to simplify the programming tasks, while keeping the index space nearly optimal.

References

1. Aho, A., Corasick, M.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS'98), pp. 534–544 (1998)
3. Amir, A., Farach, M.: Adaptive dictionary matching. In: Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS'91), pp. 760–766 (1991)
4. Amir, A., Farach, M., Galil, Z., Giancarlo, R., Park, K.: Dynamic dictionary matching. J. Comput. Syst. Sci. **49**(2), 208–222 (1994)
5. Amir, A., Farach, M., Idury, R., Poutre, A.L., Schaffer, A.: Improved dynamic dictionary matching. Inf. Comput. **119**(2), 258–282 (1995)
6. Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Text indexing and dictionary matching with one error. J. Algorithms **37**(2), 309–325 (2000)
7. Arge, L., Vitter, J.S.: Optimal external memory interval management. SIAM J. Comput. **32**(6), 1488–1508 (2003)
8. Belazzougui, D.: Succinct dictionary matching with no slowdown. In: Proceedings of Symposium on Combinatorial Pattern Matching (CPM'10), pp. 88–100 (2010)
9. 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 (ESA'02), pp. 152–164 (2002)
10. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms **57**(2), 75–94 (2005)
11. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Trans. Algorithms **3**, 2 (2007)
12. Chien, Y.F., Hon, W.K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler transform: linking range searching and text indexing. In: Proceedings of IEEE Data Compression Conference (DCC'08), pp. 252–261 (2008)
13. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proceedings of ACM Symposium on Theory of Computing (STOC'04), pp. 91–100 (2004)
14. Dietz, P.F., Sleator, D.D.: Two algorithms for maintaining order in a list. In: Proceedings of ACM Symposium on Theory of Computing (STOC'87), pp. 365–372 (1987)
15. Ferragina, P., Grossi, R.: The string B-tree: a new data structure for string search in external memory and its applications. J. ACM **46**(2), 236–280 (1999)

16. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005)
17. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theor. Comput. Sci. **372**(1), 115–121 (2007)
18. Ferragina, P., Muthukrishnan, S., de Berg, M.: Multi-method dispatching: a geometric approach with applications to string matching problems. In: Proceedings of ACM Symposium on Theory of Computing (STOC'99), pp. 483–491 (1999)
19. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011)
20. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2005)
21. Hagerup, T., Miltersen, P.B., Pagh, R.: Deterministic dictionaries. J. Algorithms **41**(1), 69–85 (2001)
22. Hon, W.K., Lam, T.W., Shah, R., Tam, S.L., Vitter, J.S.: Compressed index for dictionary matching. In: Proceedings of IEEE Data Compression Conference (DCC'08), pp. 23–32 (2008)
23. Hon, W.K., Shah, R., Thankachan, S.V., Vitter, J.S.: On entropy-compressed text indexing in external memory. In: Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE'09), pp. 75–89 (2009)
24. Hon, W.K., Ku, T.H., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster compressed dictionary matching. In: Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE'10), pp. 191–200 (2010)
25. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: Proceedings of International Conference on Computing and Combinatorics (COCOON'96), pp. 219–230 (1996)
26. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
27. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–272 (1976)
28. McCreight, E.M.: Priority search trees. SIAM J. Comput. **14**(2), 257–276 (1985)
29. Overmars, M.H.: Efficient data structures for range searching on a grid. J. Algorithms **9**(2), 254–275 (1988)
30. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. **41**(4), 589–607 (2007)
31. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of Symposium on Switching and Automata Theory, pp. 1–11 (1973)
32. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. Inf. Process. Lett. **17**(2), 81–84 (1983)