Full-Fledged Real-Time Indexing for Constant Size Alphabets

Gregory Kucherov1 · Yakov Nekrich2

Received: 8 May 2014 / Accepted: 3 August 2016 / Published online: 17 August 2016
© Springer Science+Business Media New York 2016

Abstract In this paper we describe a data structure that supports pattern matching queries on a dynamically arriving text over an alphabet of constant size. Each new symbol can be prepended to \( T \) in \( O(1) \) worst-case time. At any moment, we can report all occurrences of a pattern \( P \) in the current text in \( O(|P| + k) \) time, where \( |P| \) is the length of \( P \) and \( k \) is the number of occurrences. This resolves, under assumption of constant size alphabet, a long-standing open problem of existence of a real-time indexing method for string matching (see Amir and Nor in Real-time indexing over fixed finite alphabets, pp. 1086–1095, 2008).

Keywords Pattern matching · Data Structures · Text indexing · Real-time indexing

1 Introduction

Two main versions of the string matching problem differ in which of the two components—pattern \( P \) or text \( T \)—is provided first in the input (or is considered as fixed) and can then be preprocessed before processing the other component. The framework when the text has to be preprocessed is usually called indexing, as it can be viewed as constructing a text index supporting matching queries.
Real-time variants of the string matching problem are about as old as string matching itself. In this framework, the text is streamed to the algorithm that has to update the index as fast as possible after each new character is submitted. An algorithm is said to be real-time if the update is done in $O(1)$ time. In the 70s, existence of real-time string matching algorithms was first studied for Turing machines. For example, it has been shown that the language $\{P\#T|P \text{ occurs in } T\}$ can be recognized by a Turing machine, while the language $\{T\#P|P \text{ occurs in } T\}$ cannot [9]. In the realm of the RAM model, the real-time variant of pattern-preprocessing string matching has been extensively studied, leading to very efficient solutions (see e.g. [3] and references therein). The indexing variant, however, still has important unsolved questions.

Back in the 70s, Slisenko [16] claimed a real-time algorithm for recognizing the language $\{T\#P|P \text{ occurs in } T\}$ on the RAM model, but its complex and voluminous full description made it unacknowledged by the scientific community, and the problem remained to be considered open for many years. In 1994, Kosaraju [12] reported another solution to this problem. In our present work, however, we are interested in a more general problem, when matching queries can be made at all moments, rather than after the entire text has been received. Specifically, in our problem, a streaming text should be processed in real time so that at each moment, a matching query $P$ can be made to the portion of the text received so far. We call this the real-time indexing problem. This problem has been considered in 2008 by Amir and Nor [2], who extended Kosaraju’s algorithm to deal with repetitive queries made at any moment of the text scan.

All the three existing real-time indexing solutions [2,12,16] support only existential queries asking whether the pattern occurs in the text, but are unable to report occurrences of the pattern. Designing a real-time text indexing algorithm that would support queries on all occurrences of a pattern is stated in [2] as the most important remaining open problem. The algorithms of [2,12] assume a constant size alphabet and are both based on constructions of “incomplete” suffix trees which can be built in real-time but can only answer existential queries. To output all occurrences of a pattern, a fully-featured suffix tree is needed, however a real-time suffix tree construction, first studied in [1], is in and of itself an open question. The best currently known algorithms spend on each character $O(\log \log n)$ worst-case time in the case of constant-size alphabets [4], or $O(\log \log n + \frac{\log^2 \log |A|}{\log \log \log |A|})$ time for arbitrary alphabets $A$ [7]. A truly real-time suffix tree construction seems unlikely to exist. Therefore, a suffix tree alone seems to be insufficient to solve the real-time indexing problem.

In this paper, we propose the first real-time text indexing solution that supports reporting all pattern occurrences, under the assumption of constant size alphabet. Similar to the previous works on real-time indexing, we assume that the text is read right-to-left, i.e. new characters are prepended at the beginning of the text. Obviously, this does not restrict the generality: if the characters have to be appended at the end of the text, it is sufficient to look for the reversed pattern in order to obtain our framework. The general idea is to maintain several data structures, three in our case, each supporting queries for different pattern lengths. Each of these structures is based on a suffix tree (or suffix-tree-like structure) exteded by some auxiliary data structures. To update a suffix tree, we use an implementation of Weiner’s algorithm which is somewhat
similar to but simpler than that of [4]. The simplification is achieved by using some external algorithmic tools, such as colored predecessor queries [14]. As a result, we can update a suffix tree in $O(\log \log n)$ worst-case time per letter, under the assumption that alphabet size is bounded by $O(\log^{1/4} n)$ and without resorting to a deamortization as in [4]. This is an interesting result in and of itself.

The paper is organized as follows. In Sect. 2, we describe auxiliary data structures and present our method for online update of suffix trees. In Sect. 3, we describe the three data structures for different pattern lengths that constitute the basis of our solution. These data structures, however, do not provide a fully real-time algorithm. Then in Sect. 4, we show how to “fix” the solution of Sect. 3 in order to obtain a fully real-time algorithm.

Note that an earlier version of this work was presented to ICALP’13, but there the $O(1)$ bound on updates held only in expectation. Here we strengthen this result by achieving a worst-case real-time processing. This is done through a simpler suffix tree update algorithm described in Sect. 3.

Throughout the paper, $\Sigma$ is an alphabet of constant size $\sigma$. Since the text $T$ is read right-to-left, it will be convenient for us to enumerate symbols of $T$ from the end, i.e. $T = t_n \ldots t_1$ and substring $t_{i+\ell}t_{i+\ell-1} \ldots t_i$ will be denoted $T[i+\ell..i].T[i..]$ denotes suffix $T[i..1]$. Throughout this paper, we reserve $k$ to denote the number of objects (occurrences of a pattern, elements in a list, etc) in the query answer.

2 Preliminaries

In this section, we describe the main algorithmic tools used by our algorithms.

2.1 Range Reporting and Predecessor Queries on Colored Lists

We use data structures from [14] for searching in dynamic colored lists. In this section, we denote by $n$ the standard parameter of the RAM model, that is we assume that a computer word consists of at least $\log n$ bits.

**Colored Range Reporting in a List.** Let elements of a dynamic linked list $L$ be assigned positive integer values called *colors*. A colored range reporting query on a list $L$ consists of two integers $col_1 < col_2$ and two pointers $ptr_1$ and $ptr_2$ that point to elements $e_1$ and $e_2$ of $L$. An answer to a colored range reporting query consists of all elements $e \in L$ occurring between $e_1$ and $e_2$ (including $e_1$ and $e_2$) such that $col_1 \leq col(e) \leq col_2$, where $col(e)$ is the color of $e$. The following result on colored range reporting has been proved by Mortensen [14].

**Lemma 1** ([14], Theorem 4.1) Suppose that $col(e) \leq \log^{f} n$ for all $e \in L$ and some constant $f \leq 1/4$. We can answer color range reporting queries on $L$ in $O(\log \log m + k)$ time using an $O(m)$-space data structure, where $m \leq n$ is the number of elements in $L$. Insertion of a new element into $L$ is supported in $O(\log \log m)$ time.

For completeness, we sketch how the result of Mortensen is obtained. List $L$ is divided into blocks of size $\Theta(\log^{3+\varepsilon} m)$ for an arbitrary small constant $\varepsilon > 0$. Each
block is assigned a positive integer label bounded by \( m \), such that block labels grow monotonously when we traverse the list left-to-right. Mortensen shows how this labeling can be maintained in \( O(1) \) time per update using the algorithm of Willard [18] and Theorem 5 in [6]. That is, his method changes \( O(1) \) block labels when a new element is inserted into the list. For every element of the list, a pointer to the following element of the same color is additionally maintained.

To answer queries that are completely inside one block, we store a balanced binary tree for each block. The leaves contain block elements. In every internal node \( u \), we keep information about the leftmost element of color \( \mu \) in the subtree of \( u \), denoted \( \min(\mu, u) \), for every color \( \mu \); we set \( \min(\mu, u) = \text{NULL} \) if there are no elements of color \( \mu \) below node \( u \). Since every element in a block can be identified by \( O(\log \log n) \) bits, we need \( o(\log n) \) bits to store this information if the number of colors is sufficiently small. Essentially, the constant \( f \) is chosen in such way that the information about leftmost elements of every color can be stored in \( o(\log n) \) bits. We remark that a larger value of \( f \) could also be chosen, but this is not necessary for our algorithm. Using a universal look-up table of size \( o(n) \), we can find for any node \( u \) and for any color range \([\text{col}_1, \text{col}_2]\), all elements \( e \neq \text{NULL} \), such that \( e \) is the leftmost element of color \( \mu \) in the subtree of \( u \) for some \( \mu \in [\text{col}_1, \text{col}_2] \). This query is answered in \( O(1) \) time per color.

Suppose that we have to report all elements \( e \) in an interval \([a, b]\) such that \( \text{col}(e) \in [\text{col}_1, \text{col}_2] \) and \([a, b]\) fits into one block. Since \([a, b]\) fits into one block, it is covered by \( O(\log \log m) \) block tree nodes \( u_1, \ldots, u_t \) for \( t = O(\log \log m) \). We can answer the query by visiting all these nodes. For every visited \( u_i \), we identify all \( e = \min(\text{col}, u_i) \) such that \( e \neq \text{NULL} \) and \( \text{col} \in [\text{col}_1, \text{col}_2] \). By following the links to the next element of the same color, we report all elements of color \( \text{col}(e) \) below node \( u_i \). The total time spent in all nodes is \( O(\log \log m + k) \), where \( k \) is the number of reported elements.

To answer a query that spans multiple blocks and whose boundaries correspond to block boundaries, we maintain a separate data structure \( V \) that contains integer elements associated with colors. For every block \( B_i \) and for each color \( \mu \) that occurs in \( B_i \), \( V \) contains a representative element \( e' \), such that \( \text{col}(e') = \mu \) and the value of \( e' \) is equal to the block label of \( B_i \). Colored reporting queries on \( V \) are answered using a variant of the van Emde Boas data structure [17]; this data structure also uses the fact that certain information about minimal and maximal elements of each color can be obtained in constant time because the number of colors is small. Any query can be split into at most one range that covers multiple blocks and at most two queries that fit into single blocks. A more extensive description of queries and updates can be found in [14].

**Colored Predecessor Problem.** The colored predecessor query on a list \( \mathcal{L} \) consists of an element \( e \in \mathcal{L} \) and a color \( \text{col} \). The answer to a query \((e, \text{col})\) is the closest element \( e' \in \mathcal{L} \) which precedes \( e \) such that \( \text{col}(e) = \text{col} \). The following lemma is also proved in [14]; we also refer to [10], where a similar problem is solved.

**Lemma 2** ([14], Theorem 4.1) Suppose that \( \text{col}(e) \leq \log^f n \) for all \( e \in \mathcal{L} \) and some constant \( f \leq 1/4 \). There exists an \( O(m) \) space data structure that answers colored predecessor queries on \( \mathcal{L} \) in \( O(\log \log m) \) time and supports insertions in \( O(\log \log m) \) time, where \( m \) is the number of elements in \( \mathcal{L} \).
2.2 Online Update of Suffix Trees for Small Alphabets

Classical suffix tree construction algorithms read the input text online and spend an amortized constant time on each text letter, however in the worst-case, they can spend as much as a linear time on an individual letter. Several papers studied the question of reducing the worst-case time spent on a letter, trying to approach the real-time update \([1,4,7,11]\). All of them follow Weiner’s algorithm and process the text right-to-left, as only one new suffix has to be added when a new letter is prepended from left, resulting in a constant amount of modifications. Breslauer and Italiano \([4]\) showed how to deamortize Weiner’s algorithm in the case of constant-size alphabets in order to obtain \(O(\log \log n)\) worst-case time on each new letter. Kopelowitz \([11]\) proposed a solution for an arbitrary alphabet \(A\) spending \(O(\log \log n + \log \log |A|)\) worst-case \(\textit{expected}\) time on each prepended letter. Very recently, Fischer and Gawrychowski \([7]\) showed how to obtain a (deterministic) worst-case time \(O(\log \log n + \frac{\log^2 \log |A|}{\log \log \log |A|})\) for arbitrary alphabets.

In this section, we show a simple implementation of Weiner’s algorithm that achieves a worst-case \(O(\log \log n)\) time per letter in the case when the alphabet size is bounded by \(\log^{1/4} n\). Our solution uses Lemma 2 as well as a constant-time solution to the dynamic lowest common ancestor (lca) problem \([5]\). Thus, the solution below can be viewed as a simpler and slightly more general version of the result of \([4]\), extending it from constant-size alphabets to alphabets of size \(\log^{1/4} n\).

We first briefly recall the main idea of Weiner’s algorithm using a description similar to \([4]\). Updating a suffix tree when a new letter \(a\) is prepended to the current text \(T\) is done through maintaining W-links defined as follows. For a suffix tree node labeled \(u\) and a letter \(a \in A\), W-link \(W_a(u)\) points to the locus of string \(au\) in the suffix tree, provided that \(au\) is a substring of \(T\) (i.e. exists in the current suffix tree). Note that the locus of \(au\) can be an explicit or an implicit node, and \(W_a(u)\) is called a hard or soft W-link respectively. The following properties of W-links will be useful in the sequel.

\textbf{Lemma 3} \([7]\)

(i) If for some letter \(a\), a node has a defined W-link \(W_a\), then any of its ancestor nodes has a defined W-link \(W_a\) too.

(ii) If two nodes \(u\) and \(v\) have defined hard W-links \(W_a\), then lca\((u, v)\) has a defined hard W-link \(W_a\) too.

When \(a\) is prepended to a current text \(T\), a new leaf labeled \(aT\) must be created and attached to either an existing node or a new node created by splitting an existing edge. To find the attachment node, the algorithm finds the lowest ancestor \(u\) of the leaf labeled \(T\) for which a (possibly soft) W-link \(W_a(u)\) is defined. Then the target node \(W_a(u)\) is the attachment node. The procedure is illustrated in Fig. 1 (for the case of soft W-link \(W_a(u)\)). The main difficulty of Weiner’s approach is to find the lowest ancestor of a leaf with a defined W-link \(W_a(u)\). Another difficulty is to update (soft) W-links when the attachment node results from an edge split (see \([4]\)).

In our solution, we propose to store only hard W-links and to compute soft W-links “on the fly” rather than storing them. Note that hard W-links, once installed, do not need to be updated for the rest of the algorithm, see \([7]\). To compute W-links, the following lemma from \([7]\) provides a key observation.
Fig. 1 Illustration to Weiner’s algorithm and Lemma 5. Dashed and dotted arrows represent hard and soft \( W_a \)-links respectively. \( u \) is the lowest ancestor of \( T \) with a (soft) \( W_a \)-link. \( v_1 \) is the highest descendant of \( u \) with a hard \( W_a \)-link (cf Lemma 4). b New suffix \( aT \) is attached to a newly created node \( au \). A new hard \( W \)-link \( W_a(u) = au \) is set.

**Lemma 4** [7] Assume that for a node \( u \), \( W_a(u) \) is defined and is a soft link pointing to an implicit node located on an edge \((v, w)\). Then there exists a unique highest descendant \( u' \) of \( u \) for which \( W_a(u') \) is a hard link, and, moreover, \( W_a(u') = w \).

Lemma 4 is illustrated in Fig. 1a.

To find the lowest ancestor \( u \) of a given node \( t \) with a defined (possibly soft) \( W \)-link \( W_a(u) \), consider the Euler tour of the current suffix tree in which each internal node occurs two times corresponding to its first and last visits. Then the following lemma holds.

**Lemma 5** Consider a node \( t \). Assume that \( W_a(t) \) is not defined and \( u \) is the lowest ancestor of \( t \) for which a (possibly soft) link \( W_a(u) \) is defined. Let \( v_1 \) be the closest node preceding \( t \) in the Euler tour of the suffix tree such that \( W_a(v_1) \) is a hard link. Let \( v_2 \) be the closest node following \( u \) in Euler tour of the suffix tree such that \( W_a(v_2) \) is a hard link. Then \( u \) is the lowest node between \( \text{lca}(t, v_1) \) and \( \text{lca}(t, v_2) \). Moreover, if \( \text{lca}(t, v_1) \) is the lower of the two, then \( v_1 \) is the highest descendant of \( u \) with a defined hard \( W \)-link \( W_a \), otherwise \( v_2 \) is such a descendant.

**Proof** By Lemma 3(i), if \( W_a(t) \) is not defined, then \( W_a \) is not defined for any descendant of \( t \). Thus, no node occurring between the first and the second occurrences of \( t \) in the Euler tour has a defined link \( W_a \). Consequently, definitions of nodes \( v_1 \) and \( v_2 \) are unambiguous. The lemma is illustrated in Fig. 1a (with \( t = T \)).

By Lemma 4, \( u \) has a unique highest descendant, say \( v \), with a defined hard link \( W_a(v) \). If \( v \) occurs before \( t \) in the Euler tour, then \( v \) is the closest node preceding \( t \) in the Euler tour with defined \( W_a(v) \). To show this, assume there is a closer such node \( v' \). Observe that \( v' \) is also a descendant of \( u \) and \( v' \) is not a descendant of \( v \). By Lemma 3(ii), \( \text{lca}(v, v') \) is a node with a defined hard link \( W_a \). On the other hand, \( \text{lca}(v, v') \) is a proper ancestor of \( v \) which is a contradiction. Therefore, \( v \) is node \( v_1 \) from the lemma.

Symmetrically, if \( v \) occurs after \( t \) in the Euler tour, then \( v \) is node \( v_2 \) from the lemma. Clearly, to compute \( u \), it is sufficient to pick the lowest between \( \text{lca}(t, v_1) \) and \( \text{lca}(t, v_2) \). \( \square \)
Based on the above, we implement Weiner’s algorithm by maintaining the Euler tour of the current suffix tree in a colored list \( L_W \). If a node \( u \) has a defined hard W-link \( W_u(a) \), then both occurrences of \( u \) in \( L_W \) are colored with \( a \). Note that a node can have up to \( |A| \) hard W-links and therefore have up to \( 2|A| \) occurrences in \( L_W \). However, the total number of hard W-links is limited by the number of tree nodes, as a node has at most one incoming hard W-link.

By Lemma 2, we can answer colored predecessor and successor queries on \( L_W \) in \( O(\log \log |L_W|) \) time. Therefore, nodes \( v_1 \) and \( v_2 \) defined in Lemma 5 can be found in \( O(\log \log |L_W|) \) time. Using lowest common ancestor queries on a dynamic tree [5], \( lca(t, v_1) \) and \( lca(t, v_2) \) can be computed in \( O(1) \) time. Therefore, updating the suffix tree after prepending a new symbol is done in \( O(\log \log |L_W|) = O(\log \log |T|) \) time. As an update can introduce two new hard W-links, we also need to update the colored list \( L_w \). This is easily done in \( O(1) \) time. (Details are left out and can be found e.g. in [13].)

We conclude with the main result of this section.

**Theorem 1** Consider a text over an alphabet \( A \), \( |A| \leq \log^{1/4} n \), arriving online right-to-left. After prepending a new letter to the current text \( T \), the suffix tree of \( T \) can be updated in time \( O(\log \log |T|) \) using an auxiliary data structure of size \( O(|T|) \).

The construction used in Theorem 1 can be modified to maintain a truncated suffix tree instead of the regular suffix tree. A truncated suffix tree is a compacted trie of all substrings, i.e., truncated suffixes, of a fixed length \( s \) which occur in the text. We will use the truncated suffix tree later in Sect. 3.2. Consider the problem of maintaining the truncated suffix tree in online mode. Let \( v \) be the prefix of the current text \( T \) with \( |v| = s \). The locus of \( v \) is the “active leaf” of the current truncated suffix tree. Let \( v = v'b \) for \( b \in A \), and assume that a letter \( a \) is prepended to \( T \). We then have to insert string \( av' \) into the tree, or navigate to the corresponding leaf if this string already exists in the tree. The case when \( av' \) does not exist in the tree is processed exactly in the same way as described earlier in this section, using the same data structures as those introduced for the regular suffix tree. The difference is only in the case when \( av' \) already exists in the tree, and we have to jump from leaf \( v'b \) to leaf \( av' \).

To do this, we maintain additional W-links as follows. For every truncated suffix \( cw, |cw| = s, c \in A \), we always set a W-link \( W_c(w) = cw \). For this, we always make node \( w \) explicit even if it is not a branching node. These W-links are easy to maintain: right before the node \( cw \) is created (i.e. before \( cw \) occurs for the first time), the current node is \( wc' \) for some \( c' \in A \). It is then trivial to locate node \( w \), or create it if necessary, and to set W-link \( W_c(w) = cw \). The number of additional nodes does not exceed the number of leaves in the tree, therefore the asymptotic space usage remains unchanged.

Now, when a new letter \( a \) is processed after processing the truncated suffix \( v = v'b \), we first check if \( W_a(v') \) is defined. If this is the case, we directly jump to \( av' \). Otherwise, \( av' \) has not occurred before and we insert it using the technique of this section. We summarize our result for truncated suffix tree in the following statement.

**Corollary 1** The truncated suffix tree for a text \( T \) over an alphabet \( A \), \( |A| \leq \log^{1/4} n \), can be updated in time \( O(\log \log |T|) \) after prepending a new letter; using an auxiliary data structure of size \( O(|T|) \).
3 Fast Off-Line Solution

In this section, we describe the main part of our algorithm of real-time text indexing. Based on the suffix tree construction from the previous section, the algorithm updates the index by reading the text in the right-to-left order. However, the algorithm we describe in this section will not be online, as it will have to access some “forthcoming” characters to the left of the currently processed character. Another “flaw” of the algorithm is that it will only support pattern matching queries with an additional exception: it will be able to report all occurrences of a pattern except for those that start within a small number of most recently processed symbols of \( T \). In the next section, we will show how to fix these issues and turn our algorithm into a fully real-time indexing solution that reports all occurrences of a pattern.

The algorithm distinguishes between three types of query patterns depending on their length: long patterns contain at least \((\log \log n)^2\) symbols, medium-size patterns contain between \((\log(3) n)^2\) and \((\log \log n)^2\) symbols, and short patterns contain less than \((\log(3) n)^2\) symbols\(^1\). For each of the three types of patterns, the algorithm will maintain a separate data structure supporting queries in \(O(|P| + k)\) time for matching patterns of the corresponding type.

3.1 Long Patterns

To match long patterns, we maintain a sparse suffix tree \( T_L \) storing only suffixes that start at positions \( q \cdot d \) for \( q \geq 1 \) and \( d = \log \log n / (4 \log \sigma) \). Suffixes stored in \( T_L \) are regarded as strings over a meta-alphabet of size \( \sigma^d = \log^{1/4} n \). That is, each successive block of \( d \) characters of \( \Sigma \) is packed into a single meta-character. This allows us to use the method of Sect. 2.2 to update \( T_L \), spending \( O(\log \log n) \) time on each meta-character. Since each meta-character encodes \( O(\log \log n) \) regular characters and \( \sigma = O(1) \), the update takes \( O(1) \) amortized time per character. On the other hand, we show in this section that matching a long pattern can be done with time overhead \( O((\log \log n)^2) \) which is absorbed by \( O(|P|) \) as \( |P| \geq (\log \log n)^2 \).

As the tree \( T_L \) only stores suffixes starting at positions \( qd \), \( q \geq 1 \), it is well-suited for searching for pattern occurrences starting at positions \( qd \) as well. We first describe how those occurrences are computed. Then, we will show how this can be extended to report all occurrences of a pattern.

Computing occurrences of a pattern \( P \) starting at positions \( qd \), \( q \geq 1 \) is done as in the usual suffix tree. Starting from the root, we navigate into \( T_L \) by “spelling out” \( P \) viewed as a string over the alphabet of meta-characters. That is, we process \( P \) by successive blocks of \( d \) characters until we are left with a suffix of less than \( d \) characters. If the navigation fails at some step, then \( P \) has no occurrences. Since the meta-alphabet size is only \( \log^{1/4} n \), navigation in \( T_L \) from a node to a child can be supported in \( O(1) \) time as follows. Observe that the children of any internal node \( v \in T_L \) are naturally ordered by the lexicographic order of edge labels. We store the children of \( v \) in a data structure \( P_v \) which allows us to find, in time \( O(1) \), the child whose edge label starts

---

\(^1\) Henceforth, \( \log(3) n = \log \log \log n \).
with a string (meta-character) \( S = s_1 \ldots s_d \). Moreover, we can also compute in time \( O(1) \) the “smallest” and the “largest” child of \( v \) whose edge label starts with a string \( S = s_1 \ldots s_j \) with \( j \leq d \). \( P_v \) will also support adding a new edge to \( P_v \) in \( O(1) \) time. Data structure \( P_v \) can be implemented using e.g. atomic heaps \([8]\); since all elements in \( P_v \) are bounded by \( \log^{1/4} n \), we can also implement \( P_v \) as described in \([15]\). Once we are left with a suffix \( s_1 \ldots s_j \) of \( P \), \( j < d \), we have to retrieve those children whose label (first meta-character) starts with \( s_1 \ldots s_j \). According to the above, these children form an interval in the order of all children and this interval can be retrieved in \( O(1) \) time. The occurrences of \( P \) are then retrieved by enumerating the leaves of the corresponding subtrees.

Let us now focus on all occurrences of \( P \). An occurrence of \( P \) is said to be a \( \delta \)-occurrence, \( 0 \leq \delta < d \), if it starts in \( T \) at a position \( j = qd + \delta \), for some \( q \). To be able to find all occurrences, we need an additional data structure. We maintain a list \( L_E \) defined similarly to list \( L_W \) from Sect. 2.2. The list \( L_E \) contains copies of all nodes of \( T_L \) as they occur during the Euler tour of \( T_L \). \( L_E \) contains one element for each leaf and two elements for each internal node of \( T_L \). If a node of \( L_E \) is a leaf that corresponds to a suffix \( T[i..] \), we mark the corresponding list element with the meta-character \( \overrightarrow{T}[i, d] = t_{i+1}t_{i+2} \ldots t_{i+d} \) which is interpreted as the color of this element. Colors are ordered by lexicographic order of underlying strings. If \( S = s_1 \ldots s_j \) is a string with \( j < d \), then \( S \) defines an interval of colors, denoted \([\minc(S), \maxc(S)]\), corresponding to all character strings of length \( d \) with prefix \( S \). Recall that there are \( \log^{1/4} n \) different colors. On list \( L_E \), we maintain the data structure of Lemma 1 for colored range reporting queries.

Before explaining the matching procedure, we first explain how \( T_L \) and \( L_E \) are updated. After reading character \( t_i \) for \( i = qd, q \geq 1 \), we add suffix \( T[i..] \), viewed as a string over the meta-alphabet of cardinality \( \log^{1/4} n \), to \( T_L \) following the algorithm described in Sect. 2.2. In addition, we have to update \( L_E \), i.e. to insert the new leaf holding the suffix \( T[i..] \) colored with \( t_{i+1}t_{i+2} \ldots t_{i+d} \) into \( L_E \). Note that this requires “looking ahead” at the forthcoming \( d \) characters \( t_{i+1}, t_{i+2}, \ldots, t_{i+d} \) of \( T \). If a new internal node has been inserted into \( T_L \), we also update the list \( L_E \) accordingly.

We now consider a query pattern \( P = p_1 \ldots p_m \) and show how the occurrences of \( P \) are computed. For each \( \delta, 0 \leq \delta < d \), we find all \( \delta \)-occurrences as follows. First we search for \( P_\delta = p_{\delta+1} \ldots p_m \) in \( T_L \) as it was described earlier. If this step does not fail, we then retrieve the closest explicit descendant node \( v_\delta \), or a range of descendant nodes \( v_\delta^l, v_\delta^{l+1}, \ldots, v_\delta^r \) in the case when \( P_\delta \) leads to an explicit node. This step takes time \( O(|P|/d + 1) \) for a specific \( \delta \).

Then we jump to the list \( L_E \) and retrieve the first occurrence of \( v_\delta \) (or \( v_\delta^l \)) and the second occurrence of \( v_\delta \) (or \( v_\delta^r \)) in \( L_E \). A leaf \( u \) of \( T_L \) corresponds to a \( \delta \)-occurrence of \( P \) if and only if \( u \) occurs in the subtree of \( v_\delta \) (or the subtrees of \( v_\delta^l, \ldots, v_\delta^r \)) and the color of \( u \) belongs to \([\minc(p_\delta \ldots p_1), \maxc(p_\delta \ldots p_1)]\). In the list \( L_E \), these leaves occur precisely within the interval we computed. Therefore, all \( \delta \)-occurrences of \( P \) can be retrieved in time \( O(\log \log n + k_\delta) \) by a colored range reporting query (Lemma 1), where \( k_\delta \) is the number of \( \delta \)-occurrences. Summing up over all \( \delta \), all occurrences of a long pattern \( P \) can be reported in time \( O(d(|P|/d + \log \log n) + k) = \)
\( O(\mid P \mid + d \log \log n + k) = O(\mid P \mid + k) \), as \( d = \log \log n/(4 \log \sigma) \), \( \sigma = O(1) \) and \( \mid P \mid \geq (\log \log n)^2 \).

### 3.2 Medium-Size Patterns

Now we show how to answer matching queries for patterns \( P \) where \((\log^{(3)} n)^2\leq \mid P \mid< (\log \log n)^2\). In a nutshell, we apply the same method as in Sect. 3.1 with the main difference that the sparse suffix tree will store only truncated suffixes of length \((\log \log n)^2\), i.e. prefixes of suffixes bounded by \((\log \log n)^2\) characters. We store truncated suffixes starting at positions spaced by \((\log^{(3)} n) = \log \log n\) characters. The total number of different truncated suffixes is at most \(\sigma(\log \log n)^2\). This small number of suffixes will allow us to search and update the data structures faster compared to Sect. 3.1. This will result in an \( O(\log^{(3)} n) \) time for an update for a block of \((\log^{(3)} n) \) regular characters, i.e. amortized \( O(1) \) time per character. Like with long patterns, a search for a medium-size pattern will be shown to take \( O((\log^{(3)} n)^2) = O(\mid P \mid) \) time. We now describe the details of the construction.

We store all truncated suffixes that start at positions \(qd', \) for \( q \geq 1 \) and \(d' = \log^{(3)} n\), in a tree \( \mathcal{T}_M\). \( \mathcal{T}_M \) is a compacted trie for substrings \(T[qd'..qd'−(\log \log n)^2+1] \), where these substrings are regarded as strings over the meta-alphabet \( \Sigma^{d'} \). Thus, \( \mathcal{T}_M \) is a truncated suffix tree of \( T \) (see Sect. 2.2) considered over the meta-alphabet \( \Sigma^{d'} \). Observe that the same truncated suffix can occur several times. Therefore, we augment each leaf \( v \) with a list of colors \( \text{Col}(v) \) corresponding to left contexts of the corresponding truncated suffix \( S \). More precisely, if \( S = T[qd'..qd'−(\log \log n)^2+1] \) for some \( q \geq 1 \), then \( T[qd', d'] \) is added to \( \text{Col}(v) \). Note that the number of colors is bounded by \( \sigma^{\log^{(3)} n}\). Furthermore, for each color \( \text{col} \) in \( \text{Col}(v) \), we store all positions \( i = qd' \) of \( T \) such that \( S \) occurs at \( i \) and \( \mathcal{T}[i, d'] = \text{col} \). Similar to Sect. 3.1, we maintain a colored list \( \mathcal{L}_M \) that stores the Euler tour traversal of \( \mathcal{T}_M \). For each internal node, \( \mathcal{L}_M \) contains two elements. For every leaf \( v \) and for each value \( \text{col} \) in its color list \( \text{Col}(v) \), \( \mathcal{L}_M \) contains a separate element colored with \( \text{col} \). Observe that since the size of \( \mathcal{L}_M \) is bounded by \( O(\sigma(\log \log n)^2 + \log^{(3)} n) \), updates of \( \mathcal{L}_M \) can be supported in \( O(\log \log (\sigma(\log \log n)^2)) = O(\log^{(3)} n) \) time and colored reporting queries on \( \mathcal{L}_M \) can be answered in \( O(\log^{(3)} n + k) \) time (Lemma 1).

Truncated suffixes are added to \( \mathcal{T}_M \) according to Corollary 1. After reading a symbol \( t_{qd'} \) for some \( q \geq 1 \), we insert \( S_{\text{new}} = T[qd'..qd'−(\log \log n)^2+1] \) colored with \( T[qd', d'] \) into the tree \( \mathcal{T}_M \) and update the list \( \mathcal{L}_M \) accordingly. If \( \mathcal{L}_M \) already contains a leaf with string value \( S_{\text{new}} \) and color \( T[qd', d'] \), we add \( qd' \) to the list of its occurrences, otherwise we insert a new element into \( \mathcal{L}_M \) and initialize its location list to \( qd' \). Altogether, using Corollary 1, the addition of a new truncated suffix \( S_{\text{new}} \) requires \( O(\log \log \mid \mathcal{T}_M \mid) = O(\log^{(3)} n) \) time.

\footnote{For simplicity we assume that \( \log^{(3)} n \) and \( \log \log n \) are integers and \( \log^{(3)} n \) divides \( \log \log n \). If this is not the case, we can find \( d' \) and \( d \) that satisfy these requirements such that \( \log \log n \leq d' \leq 2\log \log n \) and \( \log^{(3)} n \leq d' \leq 2\log^{(3)} n \).}
A query for a pattern \( P = p_1 \ldots p_m \), such that \((\log^3 n)^2 \leq m < (\log \log n)^2\), is answered in the same way as in Sect. 3.1. For each \( \rho = 0, \ldots, \log^3 n - 1 \), we find locus nodes \( v_\rho^l, \ldots, v_\rho^r \) (possibly with \( v_\rho^l = v_\rho^r \)) of \( P_\rho = p_\rho + 1 \ldots p_m \). Then, we find all elements in \( L_M \) occurring between the first occurrence of \( v_\rho^l \) and the second occurrence of \( v_\rho^r \) and colored with a color \( \text{col} \) that belongs to \([\min(p_\rho \ldots p_1), \max(p_\rho \ldots p_1)]\). For every such element, we traverse the associated list of occurrences: if a position \( i \) is in the list, then \( P \) occurs at position \( (i + \rho) \). The total time needed to find all occurrences of a medium-size pattern \( P \) is \( O(d'(|P|/d' + \log^3 n) + k) = O(|P| + \log^3 n^2 + k) = O(|P| + k) \) since \(|P| \geq (\log^3 n)^2\).

### 3.3 Short Patterns

Finally, we describe our indexing data structure for patterns \( P \) with \(|P| < (\log^3 n)^2\). Here we maintain a compact tree \( T_S \) of all distinct substrings of length \( \Delta = (\log^3 n)^2 \) seen so far in the text. Since \( \Delta \) is very small, we can afford encoding all possible trees and tabulating, for each of them, all possible updates. Furthermore, we explicitly maintain, for each substring of length \( \Delta \), a list of its occurrences in the text. This allows us to easily match a pattern in time \( O(|P|) \). A formal description of the construction follows.

Define \( T_S \) to be the compacted trie storing all distinct substrings \( T[i..i - \Delta + 1] \) for positions \( i \) of \( T \). Each internal node of a tree stores pointers to its leftmost and rightmost leaves, the leaves of a tree are organized in a list, and each leaf stores the encoding of the corresponding string \( Q \). Observe that there are \( O(2^{\sigma^\Delta}) \) different trees, and \( O(\sigma^\Delta) \) different update queries can be made on each tree.

The update table \( T_u \) stores, for each tree \( T_S \) and for any string \( Q, |Q| = \Delta \), a pointer to the tree \( T'_S \) (possibly the same) obtained after inserting \( Q \) to \( T_S \). Table \( T_u \) uses \( O(2^{\sigma^\Delta} \sigma^\Delta) = o(n) \) space. The output table \( T_o \) stores, for every string \( Q \) of length \( \Delta \), the list of positions in the current text \( T \) where \( Q \) occurs. \( T_o \) has \( \sigma^\Delta = o(n) \) entries and all lists of occurrences take \( o(n) \) space altogether.

When scanning the text, we maintain the encoding of the string \( Q \) of \( \Delta \) most recently read symbols of \( T \). The encoding is updated after reading each new symbol using bit operations. Then, the current tree \( T_S \) is updated using table \( T_u \) and the current position is added to the entry \( T_o[Q] \). Both updates take \( O(1) \) time.

To answer a query \( P, |P| < \Delta \), we find the locus \( u \) of \( P \) in the current tree \( T_S \), retrieve the leftmost and rightmost leaves and traverse the leaves in the subtree of \( u \). For each traversed leaf \( v_l \) with label \( Q \), we report the occurrences stored in \( T_o[Q] \). The query takes time \( O(|P| + k)^3 \).

### 4 Real-Time Indexing

The indexes for long and medium-size patterns, described in Sects. 3.1 and 3.2 respectively, do not provide real-time indexing solutions for several reasons. The index for

---

3 In fact, the query time is even slightly better.
long patterns, for example, requires to look ahead for the forthcoming \(d\) symbols when processing symbols \(t_i\) for \(i = qd, q \geq 1\). Furthermore, for such \(i\), we are unable to find occurrences of query patterns \(P\) starting at positions \(t_{i-1} \ldots t_{i-d+1}\) before processing \(t_i\). A similar situation holds for medium-size patterns. Another issue is that in our previous development we assumed the length \(n\) of \(T\) to be known, whereas this may of course not be the case in the real-time setting. In this section, we show how to fix these issues in order to turn the indexes into real-time. First, we show how the data structures of Sects. 3.1 and 3.2 can be updated in a real-time mode. Then, we describe how to search for patterns that start among most recently processed symbols. We describe our solutions to these issues for the case of long patterns, as a simple change of parameters provides a solution for medium-size patterns too. Finally, we will show how we can circumvent the fact that the length of \(T\) is not known in advance.

In the algorithm of Sect. 3.1, the text is partitioned into blocks of length \(d\), and the insertion of a new suffix \(T[i..]\) is triggered only when the leftmost symbol \(t_i\) of a block is reached. The insertion takes time \(O(d)\) and assumes the knowledge of the forthcoming block \(t_{i+d} \ldots t_{i+1}\). To turn this algorithm into real-time, we apply a standard deamortization technique. We distribute the cost of the insertion of suffix \(T[i-d..]\) over \(d\) symbols of the block \(t_{i+d} \ldots t_{i+1}\). This is correct, as by the time we start reading the block \(t_{i+d} \ldots t_{i+1}\), we have read the block \(t_i \ldots t_{i-d+1}\) and therefore have all necessary information to insert suffix \(T[i-d..]\). In this way, we spend \(O(1)\) time per symbol to update all involved data structures.

Now assume we are reading a block \(t_{i+d} \ldots t_{i+1}\), i.e. we are processing some symbol \(t_{i+\delta}\) for \(1 \leq \delta < i\). At this point, we are unable to find occurrences of a query pattern \(P\) starting at \(t_{i+\delta} \ldots t_{i+1}\) as well as within the two previous blocks, as they have not been indexed yet. This concerns up to \((3d - 1)\) most recent symbols. We then introduce a separate procedure to search for occurrences that start in the \(3d\) leftmost positions of the already processed text. This can be done by simply storing \(T\) in a compact form \(T_c\) where every \(\log n\) consecutive symbols are packed into one computer word\(^4\). Thus, \(T_c\) uses \(O(|T|/\log n)\) words of space. Using \(T_c\), we can test whether \(T[j..j - |P| + 1] = P\), for any pattern \(P\) and any position \(j\), in \(O(|P|/\log n) = o(|P|/d) + O(1)\) time. Therefore, checking \(3d\) positions takes time \(o(|P|) + O(d) = O(|P|)\) for a long pattern \(P\).

We now describe how we can apply our algorithm in the case when the text length is not known beforehand. In this case, we assume \(|T|\) to take increasing values \(n_0 < n_1 < \ldots\), as long as the text \(T\) keeps growing. Here, \(n_0\) is some appropriate initial value and \(n_i = 2n_{i-1}\) for \(i \geq 1\).

Suppose now that \(n_i\) is the currently assumed value of \(|T|\). After we reach character \(t_{ni/2}\), during the processing of the next \(ni/2\) symbols, we keep building the index for \(|T| = n_i\) and, in parallel, rebuild all the data structures under assumption that \(|T| = n_{i+1} = 2n_i\). In particular, if \(\log \log (2n_i) \neq \log \log n_i\), we build a new index for long patterns, and if \(\log^3 (2n_i) \neq \log^3 n_i\), we build a new index for medium-size and short patterns. If \(\log_2 (2n_i) \neq \log_2 n_i\), we also construct a new compact representation \(T_c\) introduced earlier in this section. Altogether, we distribute the construction cost of

\(^4\) In fact, it would suffice to store \(3d - 1\) most recently read symbols in compact form.
the data structures for \( T[n_i..1] \) under assumption \(|T| = 2n_i\) over the processing of \( t_{n_i/2+1} \ldots t_{n_i} \). Since \( O(n_i) = O(n_i/2) \), processing these \( n_i/2 \) symbols remains real-time. By the time \( t_{n_i} \) has been read, all data structures for \(|T| = 2n_i\) have been built. The algorithm proceeds then with the new value \(|T| = n_{i+1}\). Finally, observe that the intervals \( [n_i/2 + 1, n_i] \) are all disjoint, therefore the overhead per letter incurred by the procedure remains constant. In conclusion, the whole algorithm remains real-time. We finish with our main result.

**Theorem 2** There exists a data structure storing a text \( T \) over a constant-size alphabet that can be updated in \( O(1) \) worst-case time after prepending a new symbol to \( T \). This data structure supports reporting all occurrences of a pattern \( P \) in the current text \( T \) in \( O(|P| + k) \) time, where \( k \) is the number of occurrences.

5 Conclusions

In this paper we presented the first real-time indexing data structure that supports reporting all pattern occurrences in optimal time \( O(|P| + k) \). As in the previous works on this topic [2,4,12], we assume that the input text is over an alphabet of constant size. It may be possible to extend our result to alphabets of poly-logarithmic size.

Acknowledgements  GK has been supported by the Labex Bézout program funded by the French government. This work was done during the visit of YN to the Laboratoire d’Informatique Gaspard Monge, supported by Université Paris-Est Marne-la-Vallée and CNRS. We thank the anonymous reviewers for helpful comments.

References

1. Amir, A., Kopelowitz, T., Lewenstein, M., Lewenstein, N.: Towards real-time suffix tree construction. In: Consens, M., Navarro, G. (eds.) Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE), volume 3772 of Lecture Notes in Computer Science, pp. 67–78. Springer, Berlin (2005)
2. Amir, A. and Nor, I.: Real-time indexing over fixed finite alphabets. In: Proceedings of 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2008), pp. 1086–1095 (2008)
3. Breslauer, D., Grossi, R., Mignosi, F.: Simple real-time constant-space string matching. In: Giancarlo, R., Manzini, G. (eds.) Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 6661, pp. 173–183. Springer, Berlin (2011)
4. Breslauer, D., Italiano, G.F.: Near real-time suffix tree construction via the fringe marked ancestor problem. In: Proceedings of 18th International Symposium on String Processing and Information Retrieval (SPIRE 2011), pp. 156–167 (2011)
5. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J. Comput. 34(4), 894–923 (2005)
6. Dietz, P.F., Sleator, D.D.: Two algorithms for maintaining order in a list. In: Proceedings of 19th Annual ACM Symposium on Theory of Computing (STOC 1987), pp. 365–372 (1987)
7. Fischer, J., Gawrychowski, P.: Alphabet-dependent string searching with wexponential search trees. CoRR, abs/1302.3347 (2013)
8. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J. Comput. Syst. Sci. 48(3), 533–551 (1994)
9. Galil, Z.: String matching in real time. J. ACM 28(1), 134–149 (1981)
10. Giora, Y., Kaplan, H.: Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. ACM Trans. Algorithms (2009). doi:10.1145/1541885.1541889
11. Kopelowitz, T.: On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In: Proceedings of 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012), pp. 283–292 (2012)

12. Kosaraju, S.R.: Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In: Proceedings of 26th Annual ACM Symposium on Theory of Computing (STOC 1994), pp. 310–316. ACM (1994)

13. Kucherov, G., Nekrich, Y., Starikovskaya, T.: Cross-document pattern matching. In: Kärkkäinen, J., Stoye, J. (eds) Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM), July 3–5, 2012, Helsinki (Finland), volume 7354 of Lecture Notes in Computer Science, pp. 196–207. Springer (2012)

14. Mortensen, C.W.: Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In: Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003), pp. 618–627 (2003)

15. Navarro, G., Nekrich, Y.: Top-$k$ document retrieval in optimal time and linear space. In: Proceedings of 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012), pp. 1066–1077 (2012)

16. Slisenko, A.: String-matching in real time: some properties of the data structure. In: Mathematical Foundations of Computer Science 1978, Proceedings, 7th Symposium, Zakopane, Poland, September 4–8, 1978, volume 64 of Lecture Notes in Computer Science, pp. 493–496. Springer (1978)

17. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Math. Syst. Theory 10, 99–127 (1977)

18. Willard, D.E.: A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. Inf. Comput. 97(2), 150–204 (1992)