View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Online Algorithms for Constructing Linear-Size Suffix Trie

Diptarama Hendrian
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
diptarama@tohoku.ac.jp

Takuya Takagi
Fujitsu Laboratories Ltd., Kawasaki, Japan
takagi.takuya@fujitsu.com

Shunsuke Inenaga
Department of Informatics, Kyushu University, Fukuoka, Japan
inenaga@inf.kyushu-u.ac.jp

Abstract

The suffix trees are fundamental data structures for various kinds of string processing. The suffix tree of a string T of length n has $O(n)$ nodes and edges, and the string label of each edge is encoded by a pair of positions in T. Thus, even after the tree is built, the input text T needs to be kept stored and random access to T is still needed. The *linear-size suffix tries* (*LSTs*), proposed by Crochemore et al. [Linear-size suffix tries, TCS 638:171-178, 2016], are a "stand-alone" alternative to the suffix trees. Namely, the LST of a string T of length n occupies $O(n)$ total space, and supports pattern matching and other tasks in the same efficiency as the suffix tree without the need to store the input text T. Crochemore et al. proposed an *offline* algorithm which transforms the suffix tree of T into the LST of T in $O(n \log \sigma)$ time and $O(n)$ space, where σ is the alphabet size. In this paper, we present two types of *online* algorithms which "directly" construct the LST, from right to left, and from left to right, without constructing the suffix tree as an intermediate structure. Both algorithms construct the LST incrementally when a new symbol is read, and do not access to the previously read symbols. The right-to-left construction algorithm works in $O(n \log \sigma)$ time and $O(n)$ space and the left-to-right construction algorithm works in $O(n(\log \sigma + \log n / \log \log n))$ time and $O(n)$ space. The main feature of our algorithms is that the input text does not need to be stored.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Pattern matching

Keywords and phrases Indexing structure, Linear-size suffix trie, Online algorithm, Pattern Matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.30

Funding *Diptarama Hendrian*: Supported by JSPS KAKENHI Grant Number JP19K20208.
Shunsuke Inenaga: Supported by JSPS KAKENHI Grant Number JP17H01697.

Acknowledgements The authors thank Keisuke Goto and Mitsuru Funakoshi for discussions. The authors are also grateful for the anonymous referees for fruitful suggestions.

1 Introduction

Suffix tries are conceptually important string data structures that are the basis of more efficient data structures. While the suffix trie of a string T supports fast queries and operations such as pattern matching, the size of the suffix trie can be $\Theta(n^2)$ in the worst case, where n is the length of T. By suitably modifying suffix tries, we can obtain linear $O(n)$-size string data structures such as suffix trees [24], suffix arrays [20], directed acyclic word graphs (DAWGs) [4], compact DAWGs (CDAWGs) [5], position heaps [10], and so on. In the case of the integer alphabet of size polynomial in n, all these data structures can be constructed in $O(n)$ time and space in an *offline* manner [8, 9, 11, 13, 16, 18, 21]. In the case of a general

© Diptarama Hendrian, Takuya Takagi, and Shunsuke Inenaga;
licensed under Creative Commons License CC-BY
30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis; Article No. 30; pp. 30:1–30:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ordered alphabet of size σ, there are *left-to-right online* construction algorithms for suffix trees [23], DAWGs [4], CDAWGs [17], and position heaps [19]. Also, there are *right-to-left online* construction algorithms for suffix trees [24] and position heaps [10]. All these online construction algorithms run in $O(n \log \sigma)$ time with $O(n)$ space.

Suffix trees are one of the most extensively studied string data structures, due to their versatility. The main drawback is, however, that each edge label of suffix trees needs to be encoded as a pair of text positions, and thus the input string needs to be kept stored and be accessed even after the tree has been constructed. Crochemore et al. [7] proposed a new suffix-trie based data structure called *linear-size suffix tries* (*LSTs*). The LST of T consists of the nodes of the suffix tree of T, plus a linear-number of auxiliary nodes and suffix links. Each edge label of LSTs is a single character, and hence the input text string can be discarded after the LST has been built. The total size of LSTs is linear in the input text length, yet LSTs support fundamental string processing queries such as pattern matching within the same efficiency as their suffix tree counterpart [7].

Crochemore et al. [7] showed an algorithm which transforms the *given* suffix tree of string T into the LST of T in $O(n \log \sigma)$ time and $O(n)$ space. This algorithm is *offline*, since it requires the suffix tree to be completely built first. No efficient algorithms which construct LSTs *directly* (i.e. without suffix trees) and in an *online* manner were known.

This paper proposes two online algorithms that construct LSTs directly from the given string. The first algorithm is based on Weiner's suffix tree construction [24], and constructs the LST of T by scanning T *from right to left*. On the other hand, the second algorithm is based on Ukkonen's suffix tree construction [23], and constructs the LST of T by scanning T from *left to right*. Both algorithms construct the LST incrementally when a new symbol is read, and do not access the previously read symbols. This also means that our construction algorithms do not need to store the input text, and the currently processed symbol in the text can be immediately discarded as soon as the symbol at the next position is read. The right-to-left construction algorithm works in $O(n \log \sigma)$ time and $O(n)$ space and the left-to-right construction algorithm works in $O(n(\log \sigma + \frac{\log n}{\log \log n}))$ time and $O(n)$ space.

2 Preliminaries

Let Σ denote an *alphabet* of size σ. An element of Σ^* is called a *string*. For a string $T \in \Sigma^*$, the length of T is denoted by $|T|$. The *empty string*, denoted by ε, is the string of length 0. For a string T of length n, $T[i]$ denotes the i-th symbol of T and $T[i:j] = T[i]T[i+1]\dots T[j]$ denotes the *substring* of T that begins at position i and ends at position j for $1 \le i \le j \le n$. Moreover, let $T[i:j] = \varepsilon$ if $i > j$. For convenience, we abbreviate $T[1:i]$ to $T[:i]$ and $T[i:n]$ to $T[i:]$, which are called *prefix* and *suffix* of T, respectively.

2.1 Linear-size suffix trie

The *suffix trie* $\mathsf{STrie}(T)$ of a string T is a trie that represents all suffixes of T. The *suffix link* of each node U in $\mathsf{STrie}(T)$ is an auxiliary link that points to $V = U[2:|U|]$. The *suffix tree* [24] $\mathsf{STree}(T)$ of T is a path-compressed trie that represents all suffixes of T. We consider the version of suffix trees where the suffixes that occur twice or more in T can be represented by non-branching nodes. The *linear-size suffix trie* $\mathsf{LST}(T)$ of a string T, proposed by Crochemore et al. [7], is another kind of tree that represents all suffixes of T, where each edge is labeled by a single symbol. The nodes of $\mathsf{LST}(T)$ are a subset of the nodes of $\mathsf{STrie}(T)$, consisting of the two following types of nodes:

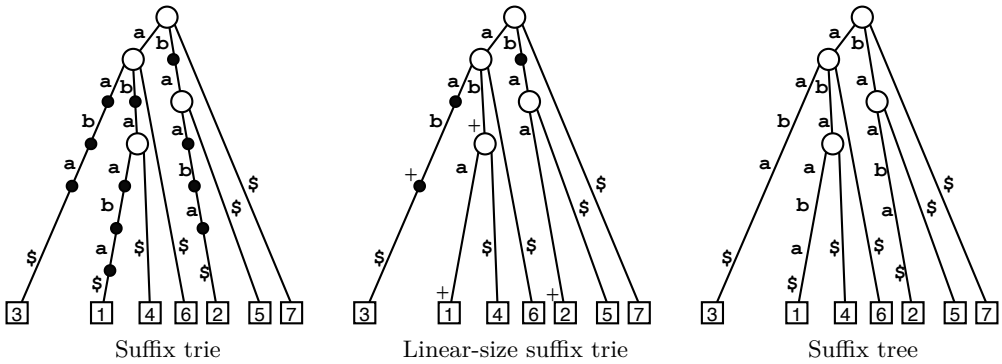

Figure 1 The suffix trie, linear-size suffix trie, and suffix tree of $T = \texttt{abaaba\$}$.

1. Type-1: The nodes of $\mathsf{STrie}(T)$ whose that also nodes of $\mathsf{STree}(T)$.
2. Type-2: The nodes of $\mathsf{STrie}(T)$ that not type-1 nodes and their suffix links point to type-1 nodes.

A non-suffix type-1 node has two or more children and a type-2 node has only one child. When T ends with a unique terminate symbol \$ that does not occur elsewhere in T, then all type-1 nodes in $\mathsf{LST}(T)$ has two or more children. The nodes of $\mathsf{STrie}(T)$ that are neither type-1 nor type-2 nodes of $\mathsf{LST}(T)$ are called *implicit nodes* in $\mathsf{LST}(T)$.

We identify each node in $\mathsf{LST}(T)$ by the substring of T that is the path label from *root* to the node in $\mathsf{STrie}(T)$. Let U and V be nodes of $\mathsf{LST}(T)$ such that V is a child of U. The edge label of $(U, V) = c$ is the same as the label of the first edge on the path from U to V in $\mathsf{STrie}(T)$. If V is not a child of U in $\mathsf{STrie}(T)$, i.e. the length of the path label from U to V is more than one, we put the $+$ sign on V and we call V a $+$-node. Figure 1 shows an example of a suffix trie, linear-size suffix trie, and suffix tree.

For convenience, we assume that there is an auxiliary node $\perp$ as the parent of the root of $LST(T)$, and that the edge from $\perp$ to the root is labeled by any symbol. This assures that for each symbol appearing in T the root has a non $+$ child. This will be important for the construction of LSTs and pattern matching with LSTs (c.f. Lemma 2).

In the description of our algorithms, we will use the following notations. For any node U, $\mathsf{parent}(U)$ denotes the parent node of U. For any edge (U, V), $\mathsf{label}(U, V)$ denotes the label of the edge connecting U and V, For a node U and symbol c, $\mathsf{child}(U, c)$ denotes the child of U whose incoming edge label is c, if it exists. We denote $+(U) = \mathsf{true}$ if U is a $+$-node, and $+(U) = \mathsf{false}$ otherwise. The suffix link of a node U is defined as $\mathsf{slink}(U) = V$, where $V = U[2 : |U|]$. The reversed suffix link of a node U with a symbol $c \in \Sigma$ is defined as $\mathsf{rlink}(U, c) = V$, if there is a node U such that $cU = V$. It is undefined otherwise. For any type-1 node U, $\mathsf{t1parent}(U)$ denotes the nearest type-1 ancestor of U, and $\mathsf{t1child}(U, c)$ denotes the nearest type-1 descendant of U on c edge. For any type-2 node U, $\mathsf{child}(U)$ is the child of U, and $\mathsf{label}(U)$ is the label of the edge connecting U and its child.

2.2 Pattern matching using linear-size suffix trie

In order to efficiently perform pattern matching on LSTs, Crochemore et al. [7] introduced *fast links* that are a chain of *suffix links of edges*.

▶ **Definition 1.** *For any edge* (U, V)*, let* $\mathsf{fastLink}(U, V) = (\mathsf{slink}^h(U), \mathsf{slink}^h(V))$ *such that* $\mathsf{slink}^h(U) \neq \mathsf{parent}(\mathsf{slink}^h(V))$ *and* $\mathsf{slink}^{h-1}(U) = \mathsf{parent}(\mathsf{slink}^{h-1}(V))$*, where* $\mathsf{slink}^0(U) = U$ *and* $\mathsf{slink}^i(U) = \mathsf{slink}(\mathsf{slink}^{i-1}(U))$.

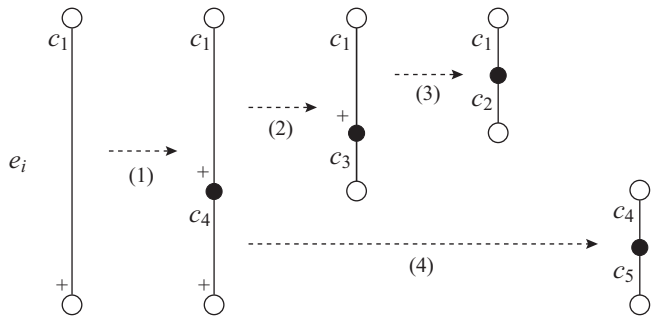

Figure 2 Illustration for our pattern matching algorithm with LST. The dashed arrows represent fast links. The number in parentheses show the orders of applications of fast links when traversing $P_i = c_1c_2c_3c_4c_5$ on the edge e_i.

Here, h is the minimum number of suffix links that we need to traverse so that $\mathsf{slink}^h(U) \neq \mathsf{parent}(\mathsf{slink}^h(V))$. Namely, after taking h suffix links from edge (U, V), there is at least one type-2 node in the path from $\mathsf{slink}^h(U)$ to $\mathsf{slink}^h(V)$. Since type-2 nodes are not branching, we can use the labels of the type-2 nodes in this path to retrieve the label of the edge (U, V) (see Lemma 2 below). Provided that $\mathsf{LST}(T)$ has been constructed, the fast link $\mathsf{fastLink}(U, V)$ for every edge (U, V) can be computed in a total of $O(n)$ time and space [7].

▶ **Lemma 2** ([7]). *The underlying label of a given edge (U, V) of length ℓ can be retrieved in $O(\ell \log \sigma)$ time by using fast links.*

Crochemore et al. [7] claimed that due to Lemma 2 one can perform pattern matching for a given pattern P in $O(|P| \log \sigma)$ time with the LST. However, the proofs provided in [7] for the correctness and time efficiency of their pattern matching algorithm looks unsatisfactory to us, because the algorithm of Crochemore et al. [7] does not seem to guarantee that the label of a given edge is retrieved sequentially from the first symbol to the last one (see also [22]). Still, in the following lemma we present an algorithm which efficiently performs the longest prefix match for a given pattern on the LST with fast links:

▶ **Lemma 3.** *Given $\mathsf{LST}(T)$ and a pattern P, we can find the longest prefix P' of P that occurs in T in $O(|P'| \log \sigma)$ time.*

Proof. Let $P_1P_2 \cdots P_m = P'$ be the factorization of P' such that $P_1 \cdots P_i$ is a node in $\mathsf{LST}(T)$ for $1 \leq i < m$, $P_1 \cdots P_i = \mathsf{parent}(P_1 \cdots P_{i+1})$ for $1 \leq i < m-1$, and $P_1 \cdots P_{m-1}$ is the longest prefix of P' that is a node in $\mathsf{LST}(T)$. If $P_1 \cdots P_{m-1} = P'$, then $P_m = \varepsilon$. In what follows, we consider a general case where $P_m \neq \varepsilon$.

Suppose we have successfully traversed up to $P_1 \cdots P_{i-1}$, and let U be the node representing $P_1 \cdots P_{i-1}$. If U has no out-going edge labeled $c_1 = P_i[1] = P[|P_1 \cdots P_{i-1}| + 1]$, then the traversal terminates on U. Suppose U has an out-going edge labeled c_1 and let V be the child of U with the c_1-edge. We denote this edge by $e_i = (U, V)$. See also Figure 2 for illustration. If V is a not $+$-node, then we have read c_1 and set $U \leftarrow V$ and continue with the next symbol $c_2 = P_i[2] = P[|P_1 \cdots P_{i-1}| + 2]$. Otherwise (if V is a $+$-node), then we apply $\mathsf{fastLink}$ from edge (U, V) recursively, until reaching the edge (U', V') such that V' is not a $+$-node. Then we move onto V'. Note that by the definition of $\mathsf{fastLink}$, V' is always a type-2 node. We then continue the same procedure by setting $U \leftarrow V'$ with the next pattern symbol c_2. This will be continued until we arrive at the first edge (U, V) such that V is a type-1 node. Then, we trace back the chain of $\mathsf{fastLink}$'s from (U, V) until getting back to

the type-2 node V'' whose out-going edge has the next symbol to retrieve. We set $U \leftarrow V''$ and continue with the next symbol. This will be continued until we traverse all symbols c_j in P_i for increasing $j = 1, \ldots, |P_i|$ along the edge e_i, or find the first mismatching symbols.

The correctness of the above algorithm follows from the fact that every symbol in label of the edge e_i is retrieved from a type-2 node that is not branching, except for the first one retrieved from the type-1 node that is the origin of e_i. Since any type-2 node is not branching, we can traverse the edge e_i with P_i iff the underlying label of e_i is equal to P_i for $1 \leq i \leq m-1$. The case of the last edge e_m where the first mismatching symbols are found is analogous.

To analyze the time complexity, we consider the number of applications of fastLink. For each $1 \leq i \leq m-1$, the number of applications of fastLink is bounded by the length of the underlying label of edge e_i, which is $|P_i|$. This is because each time we follow a fastLink, at least one new symbol is retrieved. Hence we can traverse $P_1 \cdots P_{m-1}$ in $O(|P_1 \cdots P_{m-1}| \log \sigma)$ time. For the last fragment P_m, we consider the number of applications of fastLink until we find the type-2 node X whose out-going edge has the first mismatching symbol. Since the first application of fastLink for P_m begins with an edge whose destination has string depth $|P_1 \cdots P_{m-1}|$ and since each symbol appearing in T is represented by a node as a child of the root, the number of applications of fastLink until finding X is bounded by $|P_1 \cdots P_{m-1}|$. Note that this is independent of the length of the edge e_m which can be much longer than P_m. After finding X, we can traverse P_m as in the same way to previous P_i's. Thus, we can traverse P_m in $O(|P_1 \cdots P_m| \log \sigma)$. Overall, it takes $O(|P_1 \cdots P_m| \log \sigma)$ time to traverse $P' = P_1 \cdots P_m$. This completes the proof. ◀

Algorithm 6 in Appendix shows a pseudo-code of our pattern matching algorithm with the LST in Lemma 3.

3 Right-to-left online algorithm

In this section, we present an online algorithm that constructs $\mathsf{LST}(T)$ by reading T from right to left. Let $\mathcal{T}_i = \mathsf{LST}(T[i:])$ for $1 \leq i \leq n$. Our algorithm constructs $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ incrementally when $c = T[i]$ is read. For simplicity, we assume that T ends with a unique terminal symbol \$ such that $T[i] \neq \$$ for $1 \leq i < n$.

We remark that the algorithm does not construct fast links of the LSTs. The fast links can easily be constructed in $O(n)$ time after $\mathsf{LST}(T)$ has been constructed.

Let us first recall Weiner's suffix tree contraction algorithm on which our right-to-left LST construction algorithm is based. Weiner's algorithm uses the reversed suffix links of the suffix tree called *hard Weiner links*. We in particular consider the version of Weiner's algorithm that also explicitly maintains *soft-Weiner links* [6] of the suffix tree. In the suffix tree of a text T, there is a soft-Weiner link for a node V with a symbol c iff cV is a substring of T but cV is not a node in the suffix tree. It is known that the hard-Weiner links and the soft-Weiner links are respectively equivalent to the primary edges and the secondary edges of the *directed acyclic word graph* (*DAWG*) for the reversal of the input string [4].

Given the suffix tree for $T[i+1:]$, Weiner's algorithm walks up from the leaf representing $T[i+1:]$ and first finds the nearest branching ancestor V such that aV is a substring of $T[i+1:]$, and then finds the nearest branching ancestor V' such that $cV' = U'$ is also a branching node, where $c = T[i]$. Then, Weiner's algorithm finds the insertion point for a new leaf for $T[i:]$ by following the reversed suffix link (i.e. the hard-Weiner link) from V' to U', and then walking down the corresponding out-edge of U' with the difference of the string depths of V and V'. A new branching node U is made at the insertion point if necessary.

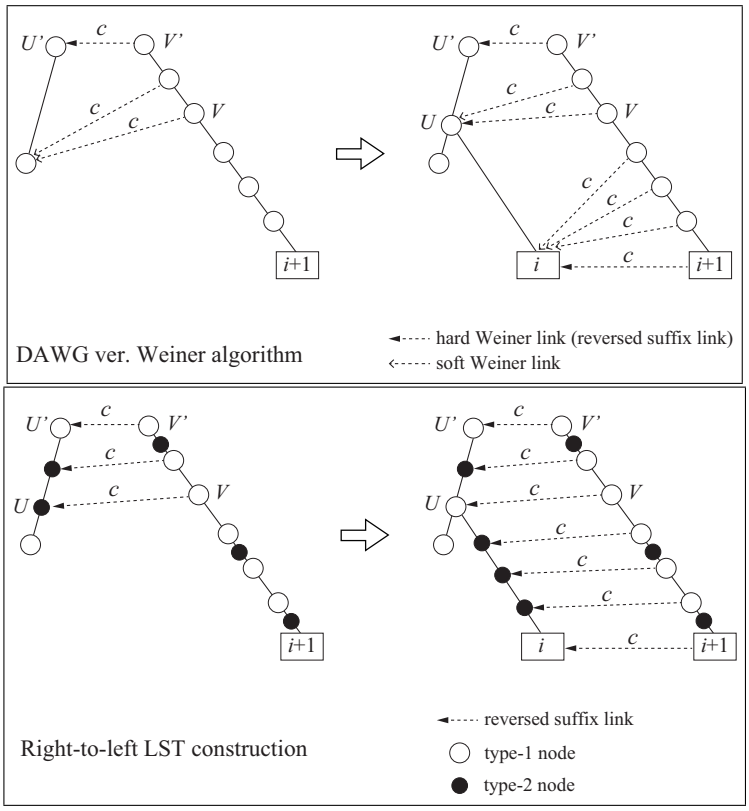

Figure 3 Upper: The DAWG version of Weiner's algorithm when updating the suffix tree for $T[i+1:]$ to the suffix tree for $T[i:]$. Lower: Our right-to-left LST construction when updating $\mathcal{T}_{i+1} = \mathsf{LST}(T[i+1:])$ to $\mathcal{T}_i = \mathsf{LST}(T[i:])$.

New soft-Weiner links are created from the nodes between the leaf for $T[i+1:]$ and V to the new leaf for $T[i:]$.

Now we consider our right-to-left LST construction. See the lower diagram of Figure 3 for illustration. The major difference between the DAWG version of Weiner's algorithm and our LST construction is that in our LST we explicitly create type-2 nodes which are the destinations of the soft-Weiner links. Hence, in our linear-size suffix trie construction, for every type-1 node between V and the leaf for $T[i+1:]$, we explicitly create a unique new type-2 node on the path from the insertion point to the new leaf for $T[i:]$, and connect them by the reversed suffix link labeled with c. Also, we can directly access the insertion point U by following the reversed suffix link of V, since U is already a type-2 node before the update.

The above observation also gives rise to the number of type-2 nodes in the LST. Blumer et al. [4] proved that the number of secondary edges in the DAWG of any string of length n is at most $n-1$. Hence we have:

▶ **Lemma 4.** *The number of type-2 nodes in the LST of any string of length n is at most $n-1$.*

The original version of Weiner's suffix tree construction algorithm only maintains a Boolean value indicating whether there is a soft-Weiner link from each node with each symbol. We note also that the number of pairs of nodes and symbols for which the indicators are true is the same as the number of soft-Weiner links (and hence the DAWG secondary edges).

We have seen that LSTs can be seen as a representation of Weiner's suffix trees or the DAWGs for the reversed strings. Another crucial point is that Weiner's algorithm only needs to read the first symbols of edge labels. This enables us to easily extend Weiner's suffix tree algorithm to our right-to-left LST construction. Below, we will give more detailed properties of LSTs and our right-to-left construction algorithm.

Let us first observe relations between $\mathcal{T}_i$ and $\mathcal{T}_{i+1}$.

▶ **Lemma 5.** *Any non-leaf type-1 node U in $\mathcal{T}_i$ exists in $\mathcal{T}_{i+1}$ as a type-1 or type-2 node.*

Proof. If there exist two distinct symbols $a, b \in \Sigma$ such that Ua, Ub are substrings of $T[i+1:]$, then clearly U is a type-1 node in $\mathcal{T}_{i+1}$. Otherwise, then let b be a unique symbol such that Ub is a substring of $T[i+1:]$. This symbol b exists since U is not a leaf in $\mathcal{T}_i$. Also, since U is a type-1 node in $\mathcal{T}_i$, there is a symbol $a \neq b$ such that Ua is a substring of $T[i:]$. Note that in this case Ua is a prefix of $T[i:]$ and this is the unique occurrence of Ua in $T[i:]$. Now, let $U' = U[2:]$. Then, $U'a$ is a prefix of $T[i+1:]$. Since $U'b$ is a substring of $T[i+1:]$, U' is a type-1 node in $\mathcal{T}_{i+1}$ and hence U is a type-2 node in $\mathcal{T}_{i+1}$. ◀

As was described above, only a single leaf is added to the tree when updating $\mathcal{T}_{i+1}$ to $\mathcal{T}_i$. The type-2 node of $\mathcal{T}_i$ that becomes type-1 in $\mathcal{T}_i$ is the *insertion point* of this new leaf.

▶ **Lemma 6.** *Let U be the longest prefix of $T[i:]$ such that U is a prefix of $T[j:]$ for some $j > i$. U is a node in $\mathcal{T}_{i+1}$.*

Proof. If $U = \varepsilon$ then U is the root. Otherwise, since U occurs twice or more in $T[i:]$ and $T[i : i+|U|] \neq T[j : j+|U|]$, U is a type-1 node in $\mathcal{T}_i$. By Lemma 5, U is a node in $\mathcal{T}_{i+1}$. ◀

By Lemma 6, we can construct $\mathcal{T}_i$ by adding a branch on node U, where U is the longest prefix of $T[i:]$ such that U is a prefix of $T[j:]$ for some $j > i$. This node U is the insertion point for $\mathcal{T}_i$. The insertion point U can be found by following the reversed suffix link labeled by c from the node $U[2:]$ i.e. $U = \mathsf{rlink}(U[2:], c)$. Since U is the longest prefix of $T[i:]$ where $U[2:]$ occurs at least twice in $T[i+1:]$, $U[2:]$ is the deepest ancestor of the leaf $T[i+1:]$ that has the reversed suffix link labeled by c. Therefore, we can find U by checking the reversed suffix links of the ancestors of $T[i+1:]$ walking up from the leaf. We call this leaf representing $T[i+1:]$ as the *last leaf* of $\mathcal{T}_{i+1}$.

After we find the insertion point, we add some new nodes. First, we consider the addition of new type-1 nodes.

▶ **Lemma 7.** *There is at most one type-1 node U in $\mathcal{T}_i$ such that U is a type-2 node in $\mathcal{T}_{i+1}$. If such a node U exists, then U is the insertion point of $\mathcal{T}_i$.*

Proof. Assume there is a type-1 node U in $\mathcal{T}_i$ such that U is a type-2 node in $\mathcal{T}_{i+1}$. There are suffixes UV and UW such that $|V| > |W|$ and $V[1] \neq W[1]$. Since U is a type-2 node in $\mathcal{T}_{i+1}$, $UV = T[i:]$ and $UW = T[j:]$ for some $j > i$. Clearly, such a node is the only one which is the branching node. ◀

From Lemma 7, we know that new type-1 node is added at the insertion point if it is a type-2 node. The only other new type-1 node is the new leaf representing $T[i:]$.

Next, we consider the addition of the new branch from the insertion point. By Lemma 7, there are no type-1 nodes between the insertion point and the leaf for $T[i:]$ in $\mathcal{T}_i$. Thus, any node V in the new branch is a type-2 node and this node is added if $V[2:]$ is a type-1 node. This can be checked by ascending from leaf $T[i+1:]$ to $U[2:]$, where U is the insertion point. Regarding the labels of the new branch, for any new node V and its parent W, the label of

Figure 4 Illustration of (a) new branch addition and (b) type-2 nodes addition. The new nodes, edges, and reverse suffix link are colored red.

(W, V) edge is the same as the label of the first edge between $W[2:]$ and $V[2:]$. The node V is a $+$-node if $V[2:]$ is a $+$-node or there is a node between $W[2:]$ and $V[2:]$. Figure 4 (a) shows an illustration of the branch addition: V can be found by traversing the ancestors of $i+1$ leaf. After we find the insertion point $U = \mathsf{rlink}(V, c)$, we add a new leaf i and type-2 nodes for each type-1 node between $i+1$ leaf and V.

Last, consider the addition of type-2 nodes when updating the insertion point U to a type-1 node. In this case, we add a type-2 node dU for any $d \in \Sigma$ such that dU occurs in $T[i:]$.

▶ **Lemma 8.** *Let U be the insertion point of $\mathcal{T}_i$. Consider the case where U is a type-2 node in $\mathcal{T}_{i+1}$. Let Z be the nearest type-1 descendant of U and Y be the nearest type-1 ancestor of U in $\mathcal{T}_{i+1}$. For any node Q such that $Q = \mathsf{rlink}(Z, d)$ for some $d \in \Sigma$, $P = \mathsf{rlink}(Y, d)$ is the parent of Q in $\mathcal{T}_{i+1}$ and there is a type-2 node R between P and Q in $\mathcal{T}_i$.*

Proof. First, we prove that P is the parent of Q in $\mathcal{T}_{i+1}$. Assume on the contrary that P is not the parent of Q. Then, there is a node $Q[:j] = dZ[:j-1]$ for some $|P| < j < |Q|$. Thus, $Z[:j-1]$ is a type-1 ancestor of Z and a type-1 descendant of Y, however this contradicts the definition of Z or Y.

Second, we prove that there is a type-2 node between P and Q in $\mathcal{T}_i$. Since U is a type-2 node in $\mathcal{T}_{i+1}$ and $Q = dZ$ is a node in $\mathcal{T}_{i+1}$, dU occurs in $T[i+1:]$ but is not a node in $\mathcal{T}_{i+1}$. Since U is a type-1 node in $\mathcal{T}_i$, dU is a type-2 node $\mathcal{T}_i$. ◀

See Figure 4 (b) for an illustration of type-2 nodes addition. It follows from Lemma 8 that we can find the position of new type-2 nodes by first following the reversed suffix link of the nearest type-1 descendant Z of U in $\mathcal{T}_{i+1}$. Then, we obtain the parent P of Z, and obtain Y by following the suffix link of P. The string depth of a new type-2 node R equal to the string depth of U plus one. We can determine whether R is a $+$-node using the difference of the string depths of Y and U. By Lemma 5, the total number of type-2 nodes added this way for all positions $1 \leq i \leq n$ is bounded by the number of type-1 and type-2 nodes in $\mathcal{T}_n$ for the whole text T.

Algorithm 1 in Appendix shows a pseudo-code of our right-to-left linear-size suffix trie construction algorithm. For each symbol $c = T[i]$ read, the algorithm finds the deepest node U in the path from the root to the last leaf for $T[i+1:]$ for which $\mathsf{rlink}(U, c)$ is defined, by walking up from the last leaf (line 5). If the insertion point $\mathit{insertPoint} = \mathsf{rlink}(U, c)$ is a type-1 node, the algorithm creates a new branch. Otherwise (if $\mathit{insertPoint}$ is a type-2 node), then the algorithm updates $\mathit{insertPoint}$ to type-1 and adds a new branch. The branch addition is done in lines 10–21.

Also, the algorithm adds nodes R such that $R = \mathsf{rlink}(\mathit{insertPoint}, d)$ for some $d \in \Sigma$ in $\mathcal{T}_i$. The algorithm finds the locations of these nodes by checking the reversed suffix links of the nearest type-1 ancestor and descendant of $\mathit{insertPoint}$ by using $\mathsf{createType2}(\mathit{insertPoint})$.

Let Y be the nearest type-1 ancestor of *insertPoint* and Z be the nearest type-1 descendant of *insertPoint*. For a symbol d such that $\mathsf{rlink}(Z, d)$ is defined, let $P = \mathsf{rlink}(Y, d)$ and $Q = \mathsf{rlink}(Z, d)$: the algorithm creates type-2 node R and connects it to P and Q.

A snapshot of right-to-left LST construction is shown in Figure 8 of Appendix.

We discuss the time complexity of our right-to-left online LST construction algorithm. Basically, the analysis follows the amortization argument for Weiner's suffix tree construction algorithm. First, consider the cost for finding the insertion point for each i.

▶ **Lemma 9.** *Our algorithm finds the insertion point of $\mathcal{T}_i$ in $O(\log \sigma)$ amortized time.*

Proof. For each iteration, the number of type-1 and type-2 nodes we visit from the last leaf to find the insertion point is at most $\mathsf{depth}(L_{i+1}) - \mathsf{depth}(U_i) + 1$, where L_{i+1} is the leaf representing $T[i+1:]$ and U_i is the insertion point for the new leaf representing $T[i:]$ in $\mathcal{T}_i$, respectively, and $\mathsf{depth}(X)$ denotes the depth of any node X in $\mathcal{T}_i$. See also the lower diagram of Figure 3 for illustration. Therefore, the total number of nodes visited is $\sum_{1 \le i < n} \mathsf{depth}(L_{i+1}) - \mathsf{depth}(U_i) + 1 \le 2n$. Since finding each reversed suffix link takes $O(\log \sigma)$ time, the total cost for finding the insertion points for all $1 \le i \le n$ is $O(n \log \sigma)$, which is amortized to $O(\log \sigma)$ per iteration. ◀

Last, the computation time of a new branch addition in each iteration is as follows.

▶ **Lemma 10.** *Our algorithm adds a new leaf and new type-2 nodes between the insertion point and the new leaf in $\mathcal{T}_i$ in $O(\log \sigma)$ amortized time.*

Proof. Given the insertion point for $\mathcal{T}_i$, it is clear that we can insert a new leaf in $O(\log \sigma)$ time. For each new type-2 node in the path from the insertion point and the new leaf for $T[i:]$, there is a corresponding type-1 node in the path above the last leaf $T[i+1:]$ (see also the lower diagram of Figure 3). Thus the cost for inserting all type-2 nodes can be charged to the cost for finding the insertion point for $\mathcal{T}_i$, which is amortized $O(\log \sigma)$ per a new type-2 node by Lemma 9. ◀

By Lemmas 9 and 10, we get the following theorem:

▶ **Theorem 11.** *Given a string T of length n, our algorithm constructs $\mathsf{LST}(T)$ in $O(n \log \sigma)$ time and $O(n)$ space online, by reading T from the right to the left.*

4 Left-to-right online algorithm

In this section, we present an algorithm that constructs the linear-size suffix trie of a text T by reading the symbols of T from the left to the right. Our algorithm constructs a slightly-modified data structure called the pre-LST defined as follows: The pre-LST $\mathsf{preLST}(T)$ of a string T is a subgraph of $\mathsf{STrie}(T)$ consisting of two types of nodes,

1. Type-1: The root, branching nodes, and leaves of $\mathsf{STrie}(T)$.
2. Type-2: The nodes of $\mathsf{STrie}(T)$ that are not type-1 nodes and their suffix links point to type-1 nodes.

The main difference between $\mathsf{preLST}(T)$ and $\mathsf{LST}(T)$ is the definition of type-1 nodes. While $\mathsf{LST}(T)$ may contain non-branching type-1 nodes that correspond to non-branching internal nodes of $\mathsf{STree}(T)$ which represent repeating suffixes, $\mathsf{preLST}(T)$ does not contain such type-1 nodes. When T ends with a unique terminal symbol \$, the pre-LST and LST of T coincide.

Our algorithm is based on Ukkonen's suffix tree construction algorithm [23]. For each prefix $T[:i]$ of T, there is a unique position k_i in $T[:i]$ such that $T[k_i : i]$ occurs twice or more in $T[:i-1]$ but $T[k_i - 1 : i]$ occurs exactly once in $T[:i]$. In other words, $T[k_i - 1 : i]$ is

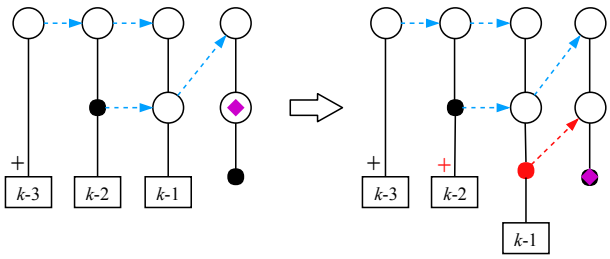

Figure 5 Illustration for updating the parts of $\mathcal{P}_{i-1}$ that correspond to $T[j : i-1]$ for $j < k_i$. The purple diamond shows the active point. The new + sign, node, and its suffix link are colored red.

the shortest suffix of $T[: i]$ that is represented as a leaf in the current pre-LST $\mathsf{preLST}(T[: i])$, and $T[k_i : i]$ is the longest suffix of $T[: i]$ that is represented in the "inside" of $\mathsf{preLST}(T[: i])$. The location of $\mathsf{preLST}(T[: i])$ representing the longest repeating suffix $T[k_i : i]$ of $T[: i]$ is called the *active point*, as in the Ukkonen's suffix tree construction algorithm. We also call k_i the *active position* for $T[: i]$. Our algorithm keeps track of the location for the active point (and the active position) each time a new symbol $T[i]$ is read for increasing $i = 1, \ldots, n$. We will show later that the active point can be maintained in $O(\log \sigma)$ amortized time per iteration, using a similar technique to our pattern matching algorithm on LSTs in Lemma 3. In order to "neglect" extending the leaves that already exist in the current tree, Ukkonen's suffix tree construction algorithm uses the idea of *open leaves* that do not explicitly maintain the lengths of incoming edge labels of the leaves. However, we cannot adapt this open leaves technique to construct pre-LST directly, since we need to add type-2 node on the incoming edges of some leaves. Fortunately, there is a nice property on the pre-LST so we can update it efficiently. We will discuss the detail of this property later. Below, we will give more detailed properties of pre-LSTs and our left-to-right construction algorithm.

Let $\mathcal{P}_i = \mathsf{preLST}(T[: i])$ be the pre-LST of $T[: i]$. Our algorithm constructs $\mathcal{P}_i$ from $\mathcal{P}_{i-1}$ incrementally when a new symbol $c = T[i]$ is read.

There are two kinds of leaves in $\mathsf{preLST}(T[: i])$, the one that are +-nodes and the other ones that are not +-nodes. There is a boundary in the suffix link chain of the leaves that divides the leaves into the two groups, as follows:

▶ **Lemma 12.** *Let $T[j : i]$ be a leaf of $\mathcal{P}_i$, for $1 \leq j < k$. There is a position l such that $T[j : i]$ is a +-node for $1 \leq j < l$ and not a +-node for $l \leq j < k_i$.*

Proof. Assume on the contrary there is a position j such that $T[j : i]$ is not a +-node and $T[j+1 : i]$ is a + node. Since $T[j : i]$ is not a +-node, $T[j : i-1]$ is a node. By definition, $T[j+1 : i-1]$ is also a node. Thus $T[j+1 : i]$ is not a +-node, which is a contradiction. ◀

Intuitively, the leaves that are +-nodes in $\mathcal{P}_i$ are the ones that were created in the last step of the algorithm with the last read symbol $T[i]$.

When updating $\mathcal{P}_{i-1}$ into $\mathcal{P}_i$, the active position k_{i-1} for $T[: i-1]$ divides the suffixes $T[j : i-1]$ into two parts, the $j < k_{i-1}$ part and the $j \geq k_{i-1}$ part. First, we consider updating the parts of $\mathcal{P}_{i-1}$ that correspond to $T[j : i-1]$ for $j < k_{i-1}$.

▶ **Lemma 13.** *For any leaf $T[j : i-1]$ of $\mathcal{P}_{i-1}$ with $j < k_{i-1} - 1$, $T[j : i-1]$ is implicit in $\mathcal{P}_i$.*

Proof. Consider updating $\mathcal{P}_{i-1}$ to $\mathcal{P}_i$. $T[k_{i-1} - 1 : i-1]$ cannot be a type-1 node in $\mathcal{P}_i$. Therefore, $T[k_{i-1} - 2 : i-1]$ is implicit in $\mathcal{P}_i$. $T[j : i-1]$ for $j < k_{i-1} - 1$ are also implicit. ◀

▶ **Lemma 14.** *If $T[j : i-1]$ is a leaf in $\mathcal{P}_{i-1}$, then $T[j : i]$ is a $+$-leaf in $\mathcal{P}_i$, where $1 \leq j < k_{i-1} - 1$.*

Proof. Assume on the contrary that $T[j : i-1]$ is a leaf in $\mathcal{P}_{i-1}$ but $T[j : i]$ is not a $+$-leaf in $\mathcal{P}_i$. Then $T[j : i-1]$ is a node in $\mathcal{P}_i$. Since $T[j : i-1]$ is a leaf in $\mathcal{P}_{i-1}$, $T[j : i-1]$ cannot be a type-1 node in $\mathcal{P}_i$. Moreover, $T[j+1 : i-1]$ is a leaf in $\mathcal{P}_{i-1}$, thus $T[j+1 : i-1]$ cannot be a type-1 node in $\mathcal{P}_i$ and $T[j : i-1]$ cannot be a type-2 node in $\mathcal{P}_i$. Therefore, $T[j : i-1]$ is neither type-1 nor type-2 node in $\mathcal{P}_i$, which contradicts the assumption. ◀

Lemma 13 shows that we do not need to add nodes on the leaves of $\mathcal{P}_{i-1}$ besides $T[k-1 : i]$ leaf and Lemma 14 shows that we can update all leaves $T[j : i]$ for $l \leq j < k-1$ to a $+$-leaf. Therefore, besides the leaf for $T[k-1 : i]$, once we update a leaf to $+$ node, we do not need to update it again. Figure 5 shows an illustration of how to update this part.

Next, we consider updating the parts of $\mathcal{P}_{i-1}$ that correspond to $T[j : i-1]$ for $j \geq k_{i-1}$. If $T[k_{i-1} : i]$ exists in the current LST (namely $T[k_{i-1} : i]$ occurs in $T[: i-1]$), then the $j \geq k_{i-1}$ part of the current LST does not need to be updated. Then we have $k_i = k_{i-1}$ and $T[k_i : i]$ is the active point of $\mathcal{P}_i$. Otherwise, we need to create new nodes recursively from the active point that will be the parents of new leaves. There are three cases for the active point $T[k_{i-1} : i-1]$ in $\mathcal{P}_{i-1}$:

Case 1. $T[k_{i-1} : i-1]$ is a type-1 node in $\mathcal{P}_{i-1}$. Let $T[p : i]$ be the longest suffix of $T[k_{i-1} : i]$ that exists in $\mathcal{P}_{i-1}$. Since $T[k_{i-1} : i-1]$ is a type-1 node, $T[j : i-1]$ is also a type-1 node for $k_{i-1} \leq j < p$. Therefore, we can obtain $\mathcal{P}_i$ by adding a leaf from the node representing $T[j : i-1]$ for every $k \leq j < p$, with edge label c by following the suffix link chain from $T[k_{i-1} : i-1]$. In this case, we only need to add one new type-2 node, which is $T[k_{i-1}-1 : i-1]$ that is connected to the type-1 node $T[k_{i-1} : i-1]$ by the suffix link. Moreover, p will be the active position for $T[: i]$, namely $k_i = p$.

Case 2. $T[k_{i-1} : i-1]$ is a type-2 node in $\mathcal{P}_{i-1}$. Similarly to Case 1, we add a leaf from the node representing $T[j : i-1]$ for every $k_{i-1} \leq j < p$ with edge label c by following the suffix link chain from $T[k_{i-1} : i-1]$, where p is defined as in Case 1.. Then, $T[k_{i-1} : i-1]$ becomes a type-1 node, and a new type-2 node $T[k_{i-1}-1 : i-1]$ is added and is connected to this type-1 node $T[k_{i-1} : i-1]$ by the suffix link. Moreover, for any symbol d such that $dT[k_{i-1} : i-1]$ is a substring of $T[: i]$, a new type-2 node for $dT[k_{i-1} : i-1]$ is added to the tree, and is connected by the suffix link to this new type-1 node $T[k_{i-1} : i-1]$. These new type-2 nodes can be found in the same way as in Lemma 8 for our right-to-left LST construction. Finally, p will become the active position for $T[: i]$, namely $k_i = p$.

Case 3. $T[k_{i-1} : i-1]$ is implicit in $\mathcal{P}_{i-1}$. In this case, there is a position $p > k_{i-1}$ such that $T[p : i-1]$ is a type-2 node. We create new type-1 nodes $T[j : i-1]$ and leaves $T[j : i]$ for $k \leq j < p$, then do the same procedure as Case 2 for $T[j : i-1]$ for $p \leq j$.

Figure 6 shows an illustration of how to add new leaves. Algorithm 3 shows a pseudo-code of our left-to-right online algorithm for constructing LSTs. In Case 1 or Case 2, the algorithm checks whether there is an out-going edge labeled with $c = T[i]$, and performs the above procedures (lines 19–29). In Case 3, we perform readEdge to check if the active point can proceed with c on the edge. The function readEdge returns the location of the new active point and sets *mismatch* = false if there is no mismatch, or it returns the mismatching position and sets *mismatch* = true if there is a mismatch. If there is no mismatch, then we just update the $T[j : i-1]$ part of the current LST for $j < k_{i-1}$. Otherwise, then we create new nodes as explained in Case 3, by split in the pseudo-code.

A snapshot of right-to-left LST construction is shown in Figure 9 of the Appendix.

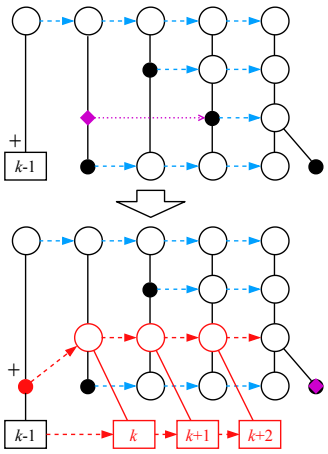

Figure 6 Illustration for updating the parts of $\mathcal{P}_{i-1}$ that correspond to $T[j : i-1]$ for $j \geq k_{i-1}$. The purple diamond and arrow show the active point and its virtual position when reading the edge. The new branches, nodes, and their suffix links are colored red.

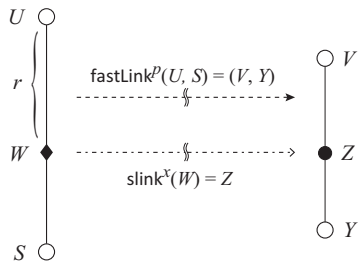

Figure 7 Illustration for our analysis of the cost to maintain the active point. The diamond shows the current location of the active point. New leaves will be created from W to Z by following the (virtual) suffix link chain of length x. When we have reached the edge (V, Y), we have already retrieved the corresponding prefix of the label between U and W. The rest of the label can be retrieved by at most r applications of fastLink from edge (V, Z).

We discuss the time complexity of our left-to-right online construction for LSTs. To maintain the active point for each $T[: i]$, we use a similar technique to Lemma 3.

▶ **Lemma 15.** *The active point can be maintained in $O(f(n) + \log \sigma)$ amortized time per each iteration, where $f(n)$ denotes the time for accessing* fastLink *in our growing LST.*

Proof. We consider the most involved case where the active point lies on an implicit node W on some edge (U, S) in the current LST. The other cases are easier to show. Let $r = |W| - |U|$, i.e., the active point is hanging off U with string depth r. Let Z be the type-2 node from which a new leaf will be created. By the monotonicity on the suffix link chain there always exists such a type-2 node. See Figure 7 for illustration. Let p be the number of applications of fastLink from edge (U, S) until reaching the edge (V, Y) on which Z lies. Since such a type-2 node Z always exists, we can sequentially retrieve the first r symbols with at most r applications of fastLink by the same argument to Lemma 3. Thus the number of applications of fastLink until finding the next location of the active point is bounded by $p + r$. If x is the number of (virtual) suffix links from W to Z, then $p \leq x$ holds. Recall that we create at least $x + 1$ new leaves by following the (virtual) suffix link chain from W to Z. Now r is charged to the number of text symbols read on the edge from U, and p is charged to the number of newly created leaves, and both of them are amortized constant as in Ukkonen's suffix tree algorithm. Thus the number of applications of fastLink is amortized constant, which implies that it takes $O(f(n) + \log \sigma)$ amortized time to maintain the active point. ◀

To maintain fastLink in our growing (suffix link) tree, we use the nearest marked ancestor (NMA) data structure [1] that allows marking, unmarking, and NMA query in an online manner in $O(\log n / \log \log n)$ time each, using $O(n)$ space on a dynamic tree of size n. By maintaining the tree of suffix links of edges enhanced with the NMA data structure, we have $f(n) = O(\log n / \log \log n)$ for Lemma 15. This leads to the final result of this section.

▶ **Theorem 16.** *Given a string T of length n, our algorithm constructs* LST(T) *in $O(n(\log \sigma + \log n / \log \log n))$ time and $O(n)$space online, by reading T from the left to the right.*

5 Conclusions and Future Work

In this paper we proposed a right-to-left online algorithm which constructs linear-size suffix trees (LSTs) in $O(n \log \sigma)$ time and $O(n)$ space, and a left-to-right online algorithm which constructs LSTs in $O(n(\log \sigma + \log n / \log \log n))$ time and $O(n)$ space, for an input string of length n over an ordered alphabet of size σ. Unlike the previous construction algorithm by Crochemore et al. [7], our algorithms do not construct suffix trees as an intermediate structure, and do not require to store the input string. Fischer and Gawrychowski [12] showed how to build suffix trees in a right-to-left online manner in $O(n(\log \log n + \log^2 \log \sigma / \log \log \log \sigma))$ time for an integer alphabet of size $\sigma = n^{O(1)}$. It might be possible to extend their result to our right-to-left online LST construction algorithm. An improvement of the running time of left-to-right online LST construction is also left for future work.

Takagi et al. [22] proposed *linear-size CDAWGs* (*LCDAWG*), which are edge-labeled DAGs obtained by merging isomorphic subtrees of LSTs. They showed that the LCDAWG of a string T takes only $O(e + e')$ space, where e and e' are respectively the numbers of right and left extensions of the maximal repeats in T, which are always smaller than the text length n. Belazzougui and Cunial [2] proposed a very similar CDAWG-based data structure that uses only $O(e)$ space. It is not known whether these data structures can be efficiently constructed in an online manner, and thus it is interesting to see if our algorithms can be extended to these data structures. The key idea to both of the above CDAWG-based structures is to implement edge labels by *grammar-compression* or *straight-line programs*, which are enhanced with efficient grammar-compressed data structures [14, 3]. In our online setting, the underlying grammar needs to be dynamically updated, but these data structures are static. It is worth considering if these data structures can be efficiently dynamized by using recent techniques such as e.g. [15].

References

1. Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked Ancestor Problems. In *Proc. FOCS 1998*, pages 534–544, 1998. doi:10.1109/SFCS.1998.743504.
2. Djamal Belazzougui and Fabio Cunial. Fast Label Extraction in the CDAWG. In *Proc. SPIRE 2017*, pages 161–175, 2017. doi:10.1007/978-3-319-67428-5_14.
3. Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random Access to Grammar-Compressed Strings and Trees. *SIAM J. Comput.*, 44(3):513–539, 2015. doi:10.1137/130936889.
4. Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M.T. Chen, and Joel Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985. doi:10.1016/0304-3975(85)90157-4.
5. Anselm Blumer, J. Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987. doi:10.1145/28869.28873.
6. Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013. doi:10.1016/j.jda.2012.07.003.
7. Maxime Crochemore, Chiara Epifanio, Roberto Grossi, and Filippo Mignosi. Linear-size suffix tries. *Theoretical Computer Science*, 638:171–178, 2016. doi:10.1016/j.tcs.2016.04.002.
8. Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching*, pages 116–129, 1997. doi:10.1007/3-540-63220-4_55.

9 Maxime Crochemore and Renaud Vérin. On compact directed acyclic word graphs. In *Structures in Logic and Computer Science: A Selection of Essays in Honor of A. Ehrenfeucht*, pages 192–211. Springer Berlin Heidelberg, 1997. doi:10.1007/3-540-63246-8_12.

10 Andrzej Ehrenfeucht, Ross M. McConnell, Nissa Osheim, and Sung-Whan Woo. Position heaps: A simple and dynamic text indexing data structure. *Journal of Discrete Algorithms*, 9(1):100–121, 2011. doi:10.1016/j.jda.2010.12.001.

11 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

12 Johannes Fischer and Pawel Gawrychowski. Alphabet-Dependent String Searching with Wexponential Search Trees. In *Proc. CPM 2015*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

13 Yuta Fujishige, Yuki Tsujimaru, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing DAWGs and Minimal Absent Words in Linear Time for Integer Alphabets. In *MFCS 2016*, pages 38:1–38:14, 2016. doi:10.4230/LIPIcs.MFCS.2016.38.

14 Leszek Gasieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant. Real-Time Traversal in Grammar-Based Compressed Files. In *Proc. DCC 2005*, page 458, 2005. doi:10.1109/DCC.2005.78.

15 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal Dynamic Strings. In *Proc. SODA 2018*, pages 1509–1528, 2018. doi:10.1137/1.9781611975031.99.

16 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016. doi:10.1016/j.tcs.2016.03.005.

17 Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Giancarlo Mauri, and Giulio Pavesi. On-line construction of compact directed acyclic word graphs. *Discrete Applied Mathematics*, 146(2):156–179, 2005. doi:10.1016/j.dam.2004.04.012.

18 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

19 Gregory Kucherov. On-line construction of position heaps. *Journal of Discrete Algorithms*, 20:3–11, 2013. doi:10.1016/j.jda.2012.08.002.

20 Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

21 Kazuyuki Narisawa, Hideharu Hiratsuka, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient Computation of Substring Equivalence Classes with Suffix Arrays. *Algorithmica*, 79(2):291–318, 2017. doi:10.1007/s00453-016-0178-z.

22 Takuya Takagi, Keisuke Goto, Yuta Fujishige, Shunsuke Inenaga, and Hiroki Arimura. Linear-Size CDAWG: New Repetition-Aware Indexing and Grammar Compression. In *SPIRE 2017*, volume 10508, pages 304–316, 2017. doi:10.1007/978-3-319-67428-5_26.

23 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

24 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE, 1973. doi:10.1109/SWAT.1973.13.

A Supplementary Figures

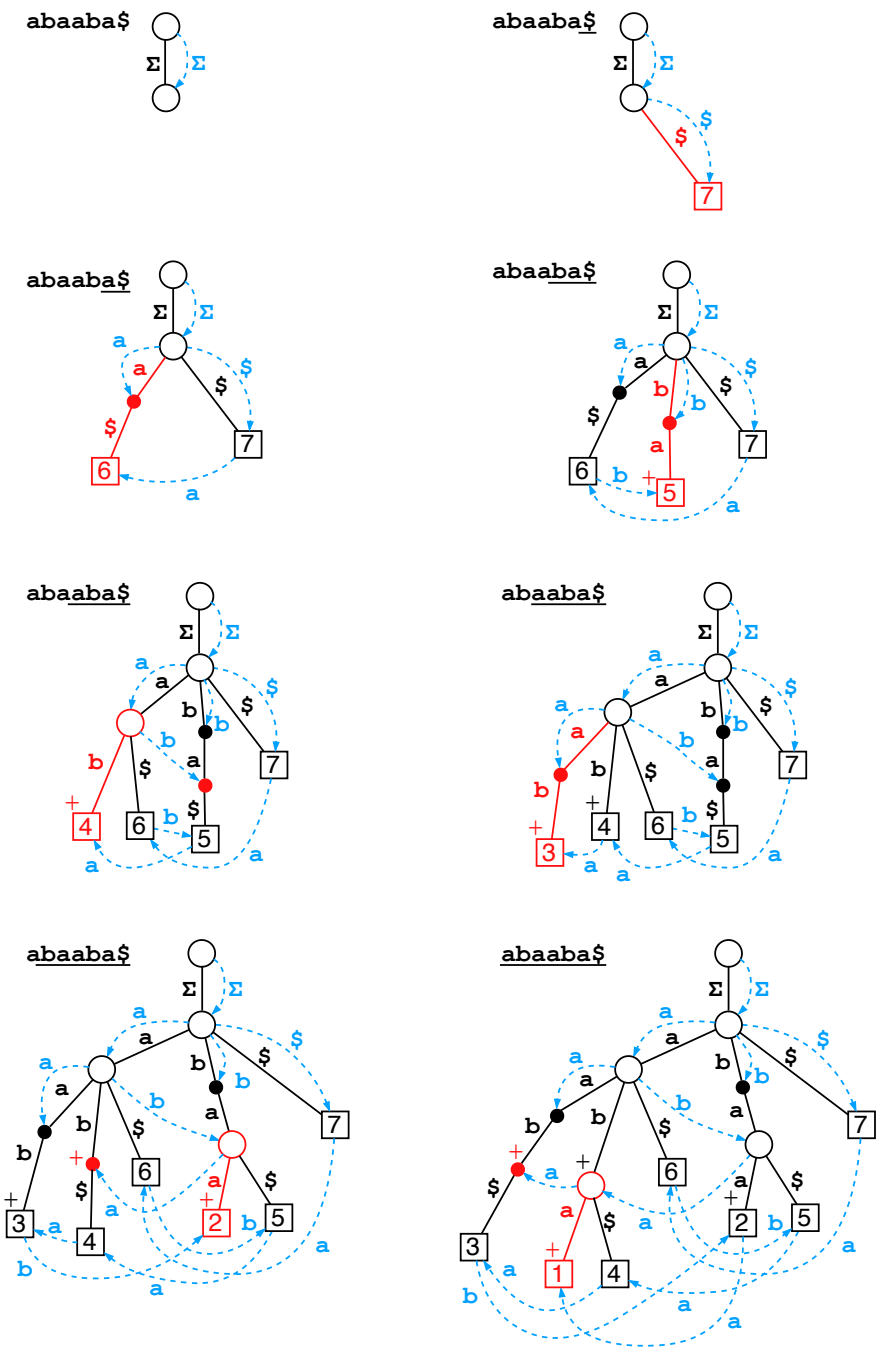

Figure 8 A snapshot of right-to-left online construction of LST(T) with T = abaaba$ by Algorithm 1. The white circles show Type-1 nodes, the black circles show Type-2 nodes, and the rectangles show leaves. The reverse suffix links and its label are colored blue. The new branches and nodes are colored red.

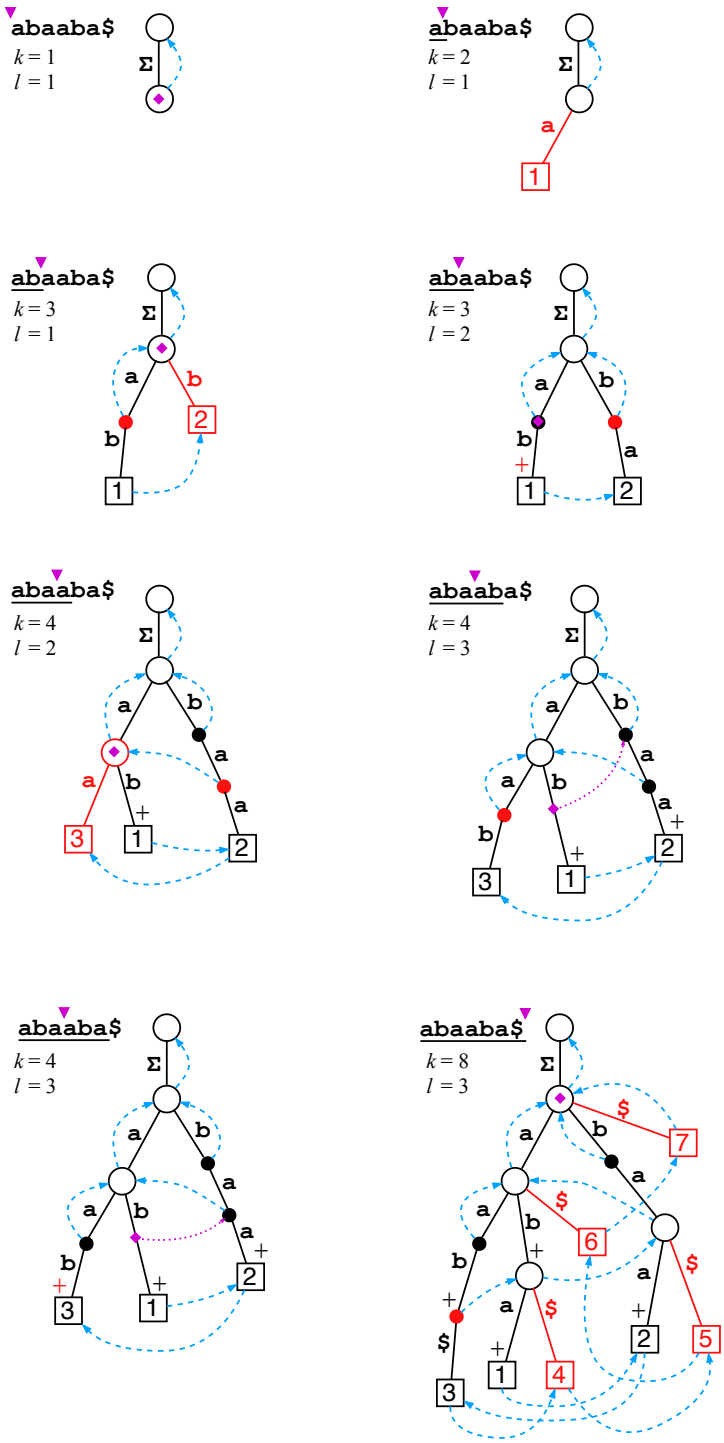

Figure 9 A snapshot of left-to-right online construction of LST(T) with T = abaaba$ by Algorithm 3. The purple diamond and arrow represent the active point and its virtual position when reading the edge label. The suffix links are colored blue. The new branches and nodes are colored red. k is the active position and l is the boundary position for +-leaves and non-+ leaves defined in Lemma 12.

B Pseudo-codes

Algorithm 1: Right-to-left linear-size suffix trie construction algorithm.

```
child(⊥, c) := root for any c ∈ Σ; rlink(⊥, c) := root for all c ∈ Σ;
prevInsPoint := ⊥; prevLeaf := root; prevLabel := NULL;
for i = n to 1 do
    c := T[i]; U := prevInsPoint;
    while rlink(U, c) = NULL do  U := parent(U);
    insertPoint := rlink(U, c);
    if type(insertPoint) = 2 then
        createType2(insertPoint);
        type(insertPoint) := 1;
    create a leaf newLeaf;
    V := prevLeaf; U := prevInsPoint; Y := newLeaf;
    while rlink(U, c) = NULL do
        create a type-2 node X;
        if U = prevInsPoint then  a = prevLabel else  a = label(U, V);
        if +(V) = true or child(U, a) ≠ V then +(Y) := true;
        child(X, a) := Y; rlink(U, c) := X; Y := X;
        V := U;
        repeat U := parent(U) until type(U) = 1;
    if U = ⊥ then  a = c else  a = label(U, V);
    if +(V) = true or child(U, a) ≠ V then +(Y) := true;
    child(insertPoint, a) := Y;
    prevInsPoint := insertPoint; prevLeaf := newLeaf; prevLabel := a;
```

Algorithm 2: createType2(U).

```
Function createType2(U)
    V := U; b = label(U); Z := t1child(U, b);
    for d such that rlink(Z, d) ≠ NULL do
        Q := rlink(Z, d);
        P := parent(Q);
        if slink(P) ≠ NULL then
            a := label(P, Q);
            Y := slink(P);
            create a type-2 node R;
            child(P, a) := R; child(R, b) := Q;
            if child(Y, a) ≠ U or +(child(Y, a)) = true then +(R) := true;
            if child(U, b) ≠ Z or +(child(U, b)) = true then +(Q) := true;
```

Algorithm 3: Left-to-right linear-size suffix trie construction algorithm.

```
create root and ⊥; child(⊥, c) := root for any c ∈ Σ;
activePoint = root; i := 1; l := 1; k := 1;
while i ≤ n do
    c := T[i];
    if child(activePoint, c) ≠ NULL then
        V := child(activePoint, c);
        (U, i′, mismatch) := readEdge((activePoint, V), i);
        if type(activePoint) = 1 then
            create a type-2 node W;
            V := parent(leaf[k − 1]);
            child(W, c) := leaf[k − 1]; child(V, label(V, leaf[k − 1])) := W;
            +(W, c) := +(leaf[k − 1]); slink(W) := activePoint;
        else +(leaf[k − 1]) := true;
        while j ≠ k − 1 do +(leaf[l]) := true; l := l + 1;
        if mismatch = false then
            if +(U) = true then +(leaf[k − 1]) := true;
        else split(U, activePoint, c, i, i′);
        activePoint := U; i := i′;
    else
        if type(activePoint) = 2 then
            createType2(activePoint); type(activePoint) := 1;
        while l ≠ k − 1 do +(leaf[l]) := true; l := l + 1;
        create a type-2 node W; V := parent(leaf[k − 1]);
        child(W, c) := leaf[k − 1]; child(V, label(V, leaf[k − 1])) := W;
        +(W, c) := +(leaf[k − 1]); slink(W) := activePoint;
        while child(activePoint, c) = NULL do
            create a leaf U;
            child(activePoint, c) := U; slink(leaf[k − 1]) := U;
            k := k + 1; leaf[k − 1] := U; activePoint = slink(activePoint);
```

Algorithm 4: readEdge$((U, V), i)$.

```
Function readEdge(U, V, i)
    while U ≠ V do
        c := T[i];
        if child(U, c) = NULL then return (U, i, true);
        else
            if +(child(U, c)) = true then
                (W, i, mismatch) := readEdge(fastLink(U, child(U, c)), i);
                if mismatch = true then return (W, i, true);
                U := W;
            else U := child(U, c); i := i + 1;
    return (U, i, false);
```

Algorithm 5: split(U, X, a, i, i').

```
Function split(U, X, a, i, i')
    b = label(U, child(U)); c' := T[i'];
    create a type-1 node W;
    V := parent(leaf[k − 1]);
    child(W, c) := leaf[k − 1]; child(V, label(V, leaf[k − 1])) := W;
    +(W) := +(leaf[k − 1]); newNode := W;
    k := k + 1; Y' := leaf[k − 1];
    while X ≠ U do
        if type(x) = 1 then  Y := child(X, a);
        d = STrieDepth(Y) − STrieDepth(X);
        while d < i' − i do
            X := Y; i := i + d;
            Y := child(X); d := STrieDepth(Y) − STrieDepth(X);
        if X ≠ U then
            create a type-2 node Z; create a leaf Y'; a := label(X, Y);
            child(X, a) := Z; child(Z, b) := Y; createType2(Z);
            type(Z) := 1; child(Z, c') := Y';
            if i' − 1 > 1 then +(Z) := true;
            if d − (i' − 1) > 1 then +(Y) := true;
            slink(newNode) := Z; slink(leaf[k − 1]) := Y';
            k := k + 1; leaf[k − 1] := Y';
            newNode := Z; X := slink(X);
    slink(newNode) := U;
```

Algorithm 6: Fast pattern matching algorithm with the LST.

```
let P be a pattern and i be a global index.
Function fastMatching(P)
    U := root; i := 1;
    while i ≤ |P| do
        if child(U, P[i]) ≠ NULL then
            U := fastDecompact(U, child(U, P[i]));
            if U = NULL then  return false;
        else  return false;
    return true;
Function fastDecompact(U, V)
    while U ≠ V do
        if child(U, P[i]) ≠ NULL then
            if +(child(U, P[i])) = false then
                U := child(U, P[i]);
                i := i + 1;
            else  U = fastDecompact(fastLink(U), fastLink(child(U, P[i])));
            if i > |P| then  return V;
        else  return NULL;
    return V;
```