Sliding Suffix Trees Revisited

Laurentius Leonard[1], Shunsuke Inenaga[2], Hideo Bannai[3], and Takuya Mieno[4]

[1] Department of Informatics Science and Technology, Kyushu University, Japan.
laurentiusleonard@outlook.com
[2] Department of Informatics, Kyushu University, Japan.
inenaga.shunsuke.380@m.kyushu-u.ac.jp
[3] M&D Data Science Center, Institute of Integrated Research, Institute of Science Tokyo, Japan.
hdbn.dsc@tmd.ac.jp
[4] Graduate School of Informatics and Engineering, University of Electro-Communications, Japan.
tmieno@uec.ac.jp

Abstract. The *sliding suffix tree* (Fiala & Greene, 1989) is a suffix tree that is maintained for a sliding window $W_i = T[i..i+d-1]$ of size d that shifts over an input text T of length n from left to right, for increasing $i = 1, \ldots, n-d+1$. It is known that the sliding suffix tree can be maintained in $O(n \log \sigma)$ time with $O(d)$ space, where σ is the alphabet size. Updating the sliding suffix tree from $W_i = T[i..i+d-1]$ to $W_{i+1} = T[i+1..i+d]$ requires the following three major tasks: (1) Delete the leaf that represents the longest suffix W_i, (2) Insert new leaves that represent the suffixes of W_{i+1} that appear exactly once in W_{i+1}, and (3) After the leaf deletion due to Task (1) and each leaf insertion due to Task (2), maintain the label $\langle \ell, r \rangle$ of every edge as a *valid* pair in the new window W_{i+1}, such that $i+1 \leq \ell \leq r \leq i+d$. In this paper, we present the first algorithm that performs Task (3) in $O(1)$ worst-case time per node deletion/insertion, which leads to another alternative to efficient sliding suffix tree construction. This is an improvement over the existing algorithms by Larsson (1996, 1999) and by Senft (2005) both of which can only perform Task (3) in $O(1)$ *amortized* time. Our key data structure is a non-trivial extension of *leaf pointers*, which were originally proposed by Brodnik and Jekovec (2018) for pattern matching with sliding suffix trees.

Keywords: suffix trees, sliding windows, online algorithms, string data structures

1 Introduction

The suffix tree [11] of a string T, denoted $\mathsf{STree}(T)$, is an edge-labeled rooted tree that represents all suffixes of T. It is well known that $\mathsf{STree}(T)$ can be stored in $O(n)$ space for any string T of length n by representing each edge label x with a pair $\langle i, j \rangle$ of positions in T such that $x = T[i..j]$. A suffix tree is a powerful string indexing data structure with a myriad of applications [4].

Among the variations of the suffix tree is the sliding suffix tree, proposed by Fiala and Greene [3]. Let $W_i = T[i..i+d-1]$ be a *sliding window* of (fixed) size d over a text T of length n. The sliding suffix tree is the suffix tree of sliding window W_i that is maintained and updated for increasing $i = 1, \ldots, n-d+1$. Sliding suffix trees are core data structures in data compression [3,5,6] including the original variant of the Lempel-Ziv 77 compression [12], as well as pattern matching [1], computing maximal absent words (MAWs) [2], minimal unique substrings (MUSs) [7], and string net frequencies (NFs) [8], in the sliding window model.

Shifting the current window $W_i = T[i..i+d-1]$ to the next window $W_{i+1} = T[i+1..i+d]$ involves deleting the leftmost character $T[i]$ and appending a new character $T[i+d]$ to the window. Thus, updating $\mathsf{STree}(W_i)$ to $\mathsf{STree}(W_{i+1})$ requires the following major tasks in the data structure:

Laurentius Leonard, Shunsuke Inenaga, Hideo Bannai, Takuya Mieno: Sliding Suffix Trees Revisited, pp. 13–27.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

(1) Delete the leaf that represents the longest suffix $W_i = T[i..i+d-1]$.
(2) Insert new leaves that represent the suffixes of $W_{i+1} = T[i+1..i+d]$ that appear exactly once in W_{i+1}, and
(3) After the leaf deletion due to Task (1) and each leaf insertion due to Task (2), maintain the label $\langle \ell, r \rangle$ of every edge as a *valid* pair in the new window $W_{i+1} = T[i+1..i+d]$, such that $i+1 \leq \ell \leq r \leq i+d$.

We do not scan all edges after each shift. Rather, whenever a local topological update creates, modifies, or needs to materialize an edge label, we compute a fresh index-pair for that edge in $O(1)$ worst-case time. While Task (2) also appears in online suffix tree constructions [10], Tasks (1) and (3) are specific to the sliding window model.

Larsson [5,6] pointed out that Task (1) is a mere operation that takes $O(1)$ *worst-case* time. Larsson [5,6] also showed that Task (2) can be done in $O(1)$ *amortized* time per character, by adapting Ukkonen's online suffix tree construction [10].

The focus of this paper is Task (3) of updating edge labels, for which the following two existing approaches are known:

- The first one is the so-called *credit-based* approach proposed by Larsson [5,6], that is based on the ideas from Fiala and Greene's earlier work [3]. The basic idea of this method is to propagate a credit to each edge label that needs to be updated, and it can be shown that the time cost for updating each edge label $\langle \ell, r \rangle$ takes $O(1)$ *amortized* time. The worst-case time is however $O(d)$ per left-end character deletion and right-end character insertion. In addition, in Section 5 we show that the worst-case time of the credit-based method is $\Omega(d)$ for a single leaf insertion or deletion on some string.
- The second one is the so-called *batch-based* approach proposed by Senft [9]. Senft's method uses standard batch-updates on overlapping blocks of size $2d$ over the text T, shifting a window of size d in each block to the right, reconstructing the suffix tree from scratch when the window no longer fits in the block, moving on to the next block prefixed by the window. It is clear that the batch-based approach takes $O(1)$ *amortized* time. However, the worst-case time is $O(d)$ per character deletion/insertion as well as for leaf deletion/insertion.

In this paper, we present the first method, named the *leaf-pointer* based approach, that is able to perform Task (3) in $O(1)$ *worst-case* time for each edge label update that is incidental to a leaf insertion or deletion. We first make a simple observation that maintaining the label for each edge $u \to v$ reduces to maintaining a pointer from node v to a leaf in the subtree rooted at v (Lemma 2 in Section 3). Then, we present how to maintain a pointer from each node to a specific leaf in its subtree in $O(1)$ worst-case time (Section 4). This is achieved by introducing our concepts of *primary* and *secondary* nodes and by maintaining leaf pointers built on them, which may be of independent interest and may be used in other applications. Table 1 summarizes the comparison of the existing methods and our proposed method.

Related work. Brodnik and Jekovec [1] used a similar notion to our leaf pointers. They showed how to perform pattern matching in $O(m \log \sigma + occ)$ time on the sliding suffix tree with $O(d)$ working space, where m is the query pattern length and occ is the number of *all* pattern occurrences in the current window. Since Brodnik and Jekovec [1] utilize the credit-based algorithm to maintain their leaf pointers, their method takes $O(1)$ *amortized* time for each leaf insertion/deletion.

Table 1. Comparison of algorithms for maintaining a suffix tree of a sliding window of size d on a string of length n over an alphabet of size σ. In the table, "amortized" and "worst case" refer to "amortized $O(1)$ time" and "worst case $O(1)$ time", respectively. The $O(\log \sigma)$ term in the total time complexities is due to maintaining branches of nodes.

algorithms	Task (1)	Task (2)	Task (3) per node deletion/insertion	total time	working space
credits [5,6]	worst case	amortized	amortized	$O(n \log \sigma)$	$O(d)$
batch [9]	worst case	amortized	amortized	$O(n \log \sigma)$	$O(d)$
leaf pointers [ours]	worst case	amortized	worst case	$O(n \log \sigma)$	$O(d)$

2 Preliminaries

2.1 Strings

Let Σ be an alphabet of size σ. An element of the set Σ^* is a string. The length of a string w is denoted by $|w|$. For a string $w = pts$, p, t, and s are called a *prefix*, *substring*, and *suffix* of w, respectively. A suffix s of w that occurs twice or more in w is said to be *repeating* in w. For a string w, $w[i]$ denotes the i-th character of string w for $1 \leq i \leq |w|$, and $w[i..j]$ denotes the substring $w[i]w[i+1] \cdots w[j]$ of string w for $1 \leq i \leq j \leq |w|$.

2.2 Sliding suffix trees

The suffix tree [11,4] of a string T, denoted $\mathsf{STree}(T)$, is the compacted trie of all suffixes of T. Note that each non-leaf node of $\mathsf{STree}(T)$ has at least two children (unless T is a unary string), and thus, the number of nodes in $\mathsf{STree}(T)$ is linear in the length of T. Each edge is labeled by a non-empty substring of T and the labels of all outgoing edges of the same node begin with distinct characters. Any substring w of T can be associated with a corresponding location (either on a node or a position on an edge) in the suffix tree where the concatenated path label from the root to the location is w. Depending on the context, we will use strings and their locations in the suffix tree interchangeably. An edge of a suffix tree from node u to node v is denoted by $u \to v$. The string label x for each edge $u \to v$ is represented by an index-pair $\langle \ell, r \rangle$ such that $x = T[\ell..r]$. This way, $\mathsf{STree}(T)$ can be stored in space linear in the length of T. The sliding suffix tree [3,9,1] is the suffix tree for a sliding window $W_i = T[i..i+d-1]$ of size d, that shifts over the input string T of length n from left to right, for increasing $i = 1, \ldots, n-d+1$. This is a version of suffix trees a.k.a. Ukkonen trees [10], in which non-repeating suffixes of W_i are represented by leaves while repeating suffixes of W_i are represented by (possibly implicit, non-branching) internal nodes. Let us consider a sliding window $W_i = T[i..i+d-1]$ over an input string T. For ease of description, let $\ell_i = i$ and $r_i = i + d - 1$. Let $W_i[s..r_i]$ be the longest repeating suffix of W_i. For any $\ell_i \leq k < s$, $\mathit{leaf}_i(k)$ denotes the leaf of the suffix tree of W_i that represents the suffix $W_i[k..r_i]$. The sliding suffix tree $\mathsf{STree}(W_i)$ for W_i consists of the leaves $\mathit{leaf}_i(k)$ for $\ell_i \leq k < s$, branching internal nodes, and the root. When there is no risk of confusion, we simply denote it by $\mathit{leaf}(k)$. Figure 1 shows an example of how a sliding suffix tree changes between two iterations. On the left side of Figure 1, strings $\mathtt{a}, \mathtt{abaca}$ and the empty string are considered to be located on nodes, while strings $\mathtt{bac}, \mathtt{c}$ are considered to be located on edges.

In our sliding suffix tree algorithm, only the last d characters of T are explicitly stored to achieve $O(d)$ space efficiency, as done in the credit-based approach by Larsson [5,6]. Thus, any edge whose index-pair contains an index $k < \ell_i$, or in other words

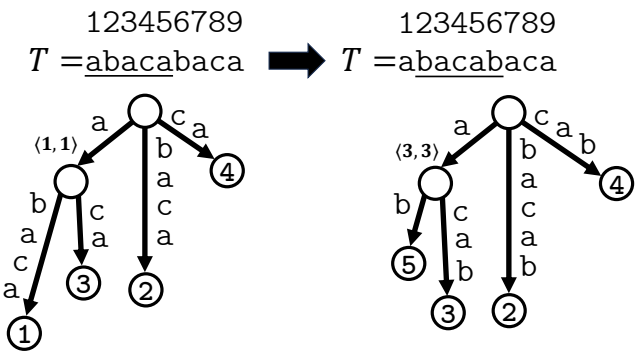


Figure 1. Sliding suffix tree across two iterations with $T =$ `abacabaca` and $|W| = 5$. $leaf(1)$ is deleted in the second iteration due to its corresponding suffix `abaca`, which would grow into `abacab`, not existing in the new window. The edge labeled `a` shows an example of an edge whose index-pair becomes outdated; if it was represented by $\langle 1, 1\rangle$ in the first iteration, an update is required as the index 1 contained in the interval is no longer inside the window.

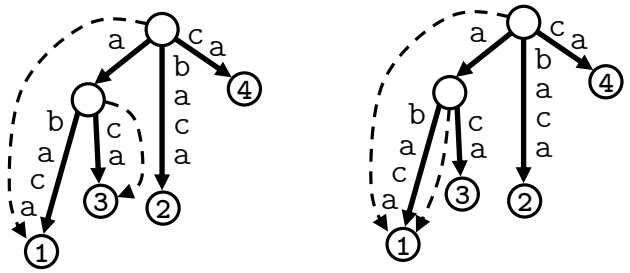


Figure 2. Two possible configurations of leaf pointers for the suffix tree of `abaca`. The dashed arrows depict leaf pointers.

any edge with an index-pair corresponding to an occurrence of the edge-label that is no longer entirely inside the window, needs to be updated to a *fresh* index-pair $\langle \ell, r\rangle$ such that $[\ell..r] \subseteq [\ell_i..r_i]$. A *leaf pointer* on a suffix tree is defined as a pointer from each internal node to any of its descendant leaves. Leaf pointers were originally proposed by Brodnik and Jekovec [1] for their use in online pattern matching. Figure 2 shows configuration examples of leaf pointers. Unlike with online suffix trees, maintaining them for a sliding suffix tree is not a trivial task. Brodnik and Jekovec [1] claimed that one can update each leaf pointer in $O(1)$ amortized time per iteration, by extending the credit-based method by Larsson [5,6].

In the rest of this paper, we will prove the following theorem:

Theorem 1. *Let T be a string of length n and d be the size of the sliding window $W_i = T[\ell_i..r_i]$ over T. There is an algorithm of $O(d)$ space that maintains a leaf pointer of every node in the sliding suffix tree in $O(1)$ worst-case time per update.*

3 Getting fresh index-pairs from leaf pointers

We assume that the topology of the suffix tree is maintained, along with the string length corresponding to each node and the index k for each $leaf(k)$. Provided that leaf pointers are maintained, we can find a fresh index-pair of any edge $u \to v$ with edge label x in constant time as follows: If v is a leaf, it is clear that $\langle k_v, r_i\rangle$ is a fresh index-pair of the edge where $v = leaf(k_v)$ and r_i is the right-end of the current

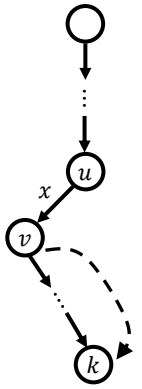


Figure 3. A figure showing the relationship between u,v, and $leaf(k)$ in the context of computing the index-pair in the general case.

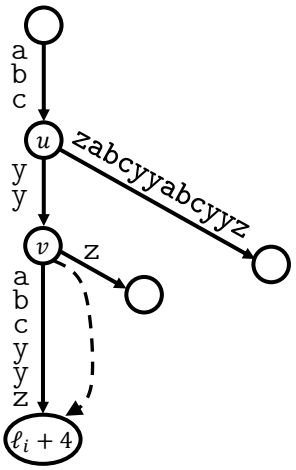


Figure 4. The part of the suffix tree that starts with the character a in Example 3.

window. Otherwise, traverse the leaf pointer of $v = ux$ to the destination leaf $leaf(k)$. The existence of the leaf implies that the suffix $T[k..r_i]$ must occur inside the current window W_i, and ux is a prefix of this suffix. Thus, $\langle k + |u|, k + |ux| - 1\rangle$ is a fresh index-pair of $u \to v$, corresponding to the substring x. Additionally, the edge label $\langle k + |u|, k + |ux| - 1\rangle$ is also *strongly fresh*, meaning that the preceding u at start-index k is also inside the window. Figure 3 illustrates the relationship between u,v, and $leaf(k)$. The following lemma immediately follows:

Lemma 2. *Assume that the leaf pointers for all internal nodes are maintained. Then, for any edge $u \to v$, one can obtain in $O(1)$ worst-case time an index-pair $\langle \ell, r\rangle$ such that $T[\ell..r]$ matches the edge's label, and $[\ell - |u|..r] \subseteq [\ell_i..r_i]$.*

Example 3. Consider $W_i =$ `abczabcyyabcyyz` and $T = PW_i$ for some $P \in \Sigma^*$, $u =$ `abc`, $v = ux =$ `abcyy`. Then, u and v are branching nodes. Regardless of the content of P, we can get an index-pair of the edge $u \to v$ with label $x =$ `yy` as follows: Traverse the leaf pointer of v to arrive at one of its descendant leaves, say, $leaf(\ell_i + 4)$ corresponding to the suffix `abcyyabcyyz`. Then, we get the index-pair $\langle \ell_i + 4 + |u|, \ell_i + 4 + |u| + |x| - 1\rangle = \langle \ell_i + 7, \ell_i + 8\rangle$, which gives us the correct label, since $W_i[8..9] =$ `yy`. Figure 4 shows the relevant part of the suffix tree of W_i, namely the locations starting with `a`.

We note that, even if the edge $u \to v$ with index-pair $\langle \ell, r\rangle$ is a fresh edge in the weak sense, i.e., $[\ell..r] \subseteq [\ell_i..r_i]$, we cannot use the opposite of the above method to obtain a leaf pointer of node u. We can obtain the start-index $\ell - |u|$, but the corresponding $leaf(\ell - |u|)$ may already be deleted, i.e., $\ell - |u| < \ell_i$, unless the edge label is strongly fresh, which cannot be assumed for sliding suffix tree construction algorithms in general.

Theorem 1 and Lemma 2 immediately lead to the following:

Corollary 4. *Let T be a string of length n and d be the size of the sliding window $W_i = T[\ell_i..r_i]$ over T. There is an algorithm of $O(d)$ space that resolves Task (3) in $O(1)$ worst-case time per leaf insertion/deletion.*

We will prove Theorem 1 in Section 4.

4 Maintaining leaf pointers in $O(1)$ worst-case time

Our leaf pointer algorithm maintains the following invariants: For each internal node x with $c > 0$ children, exactly one child node is designated as a *primary* node, while each of the remaining $c - 1$ children are designated as *secondary* nodes. An edge is similarly called primary or secondary depending on its destination node. For convenience, the root will be considered a secondary node.

Let S be the set of (arbitrarily chosen) primary nodes. Then, for each secondary internal node x, we maintain the *primary leaf pointer* $PLP_S(x)$ that points to the leaf node reached by traversing down the primary edges from x. It is clear that the path of primary edges from a secondary node x leads to a leaf (see also Figure 5 for illustrations). For each secondary leaf node x, we let $PLP_S(x) = x$. When the choice S of primary nodes in the sliding suffix tree is clear from the context, we abbreviate $PLP_S(x)$ as $PLP(x)$.

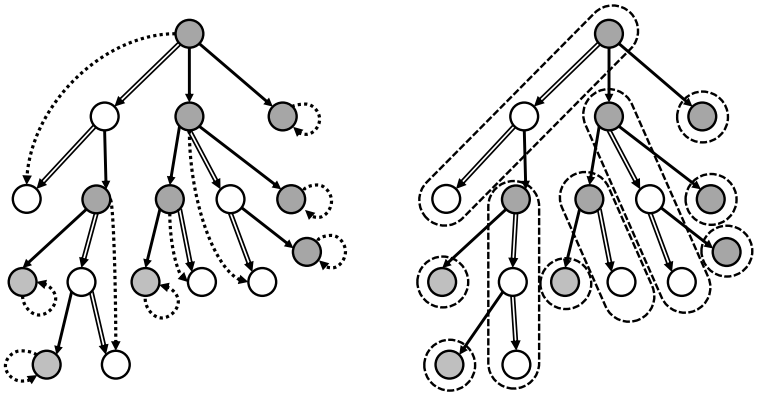


Figure 5. The white and gray nodes represent primary and secondary nodes, respectively. The double-lined solid arrows represent primary edges, while the single-lined solid arrows represent secondary edges. Left: The dotted arrow from each secondary node x represents the primary leaf pointer $PLP(x)$. Right: Our primary/secondary edges induce a path-decomposition of the tree which can be seen as a relaxed version of the well-known heavy-path decomposition, where our primary edge of each node can be *any* of its outgoing edges.

Assuming the above invariants are maintained, we can answer the leaf pointer query for any internal node x as follows: If x is secondary, return $PLP(x)$. Otherwise, x is a primary (thus non-root) internal node, meaning it has at least two children, so it must have a secondary child y. In this case, return $PLP(y)$ for an arbitrarily chosen secondary child y. See Figure 6 for a concrete example.

The invariants above also guarantee that each leaf only has one primary leaf pointer pointing to it, making updates in constant time and linear space feasible.

Lemma 5. *For each leaf u of the tree, there is exactly one primary leaf pointer that points to u. Equivalently, the primary paths induce a one-to-one correspondence between secondary nodes and leaves. Hence every leaf has a unique inverse PLP pointer.*

Proof. By categorizing the edges of the tree into primary or secondary ones, one can induce a decomposition of the tree to a disjoint set of paths (see also Figure 5). Here, each path $\mathcal{P}$ in the decomposition begins with a secondary node x, continues with a series of primary nodes (possibly empty), and ends with a leaf u. Thus, the leaf u is pointed by the primary leaf pointer from the secondary node x that is the beginning

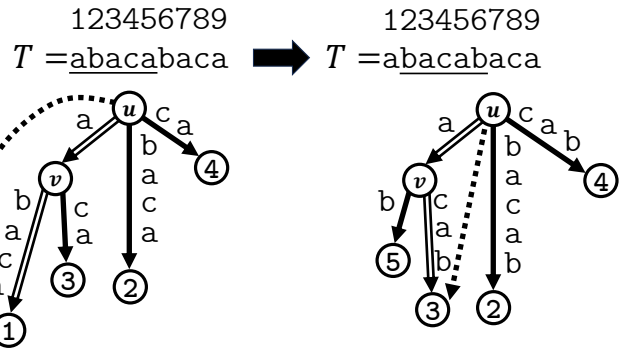


Figure 6. Primary leaf pointers maintained across two iterations of a sliding suffix tree. Primary edges are depicted by double-lined arrows, secondary edges by single-lined arrows, and primary leaf pointers by dotted arrows. The primary leaf pointers from each secondary leaf that point to themselves are omitted in this figure. Left (before the update): The secondary branching node u returns $PLP(u) = leaf(1)$, while a leaf pointer query on the primary branching node v returns the primary leaf pointer of its secondary child $leaf(3)$, resulting in $PLP(leaf(3)) = leaf(3)$. Right (after the update): Queries on u and v return $leaf(3)$ and $leaf(5)$, respectively.

node of the path $\mathcal{P}$. It is clear that the leaf u is pointed only by this secondary node x that belongs to the same path $\mathcal{P}$. □

When the suffix tree for $W_{i-1} = T[\ell_{i-1}..r_{i-1}]$ is updated to the suffix tree for $W_i = T[\ell_i..r_i]$ at the i-th iteration, the structure of the suffix tree can change. This structural change requires us to maintain primary leaf pointers efficiently, as some leaves and internal nodes can be removed and/or added. The following property is known:

Lemma 6 (Larsson [5]). *For any string T of length n, the total number of nodes removed and/or added is $O(n)$ regardless of the (fixed) size of the sliding window.*

Below, we describe our algorithm which maintains primary leaf pointers in $O(1)$ worst-case time per leaf inserted into or deleted from the sliding suffix tree. This together with Lemma 6 will give us an $O(n)$ total time updates of the sliding suffix tree.

The key to our efficient algorithm is Lemma 5 which states that the primary leaf pointer function $PLP(\cdot)$ is an injection. Namely, only a constant number of primary leaf pointers need to be updated per leaf insertion/deletion. We will also assume that the inverse pointers of PLPs are maintained, so that when $PLP(x) = u$ we can also access x from u in constant time. Whenever we create, delete, or redirect a PLP, we also update its inverse pointer accordingly.

In what follows, the *primary path* of a PLP from a secondary node x denotes the path consisting of the primary edge(s) from x to the leaf destination of $PLP(x)$. Similarly, the primary path of any node x refers to the path of primary edges from x to a leaf.

4.1 Handling node insertions on nodes

Our procedure for inserting leaf u as a child of an existing internal node w is as follows:

Case 1-1: When w has no child (Figure 7).

Procedure: Make u primary and set $PLP(w) = u$.
Correctness: This case only happens when w is the root because inserting a new leaf under an existing leaf does not occur in suffix trees of non-empty strings. The correctness is trivial.

Case 1-2: When w has a child (Figure 8).
Procedure: Make u secondary and initialize $PLP(u) = u$.
Correctness: In this case, no existing primary path changes, and therefore no existing PLP needs to be updated. A new secondary node u is created, thus $PLP(u)$ needs to be created. Since u is a leaf, u itself is the correct destination of $PLP(u)$.

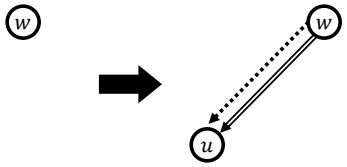


Figure 7. Case 1-1.

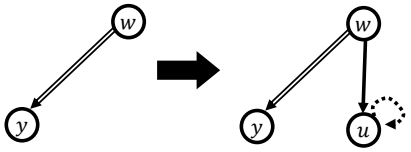


Figure 8. Case 1-2.

4.2 Handling node insertions on edges

Here, we describe our procedure for inserting a leaf u, such that an edge $x \to y$ is split into $x \to w$ and $w \to y$, where w is a newly created branch node and is the parent of u.

Case 2-1: When y is primary (Figure 9).
Procedure: Make w primary, u secondary, and initialize $PLP(u) = u$.
Correctness: In this case, the primary path that includes $x \to y$ is split. Thus, the primary path that contained $x \to y$ has changed. However, its destination clearly remains the same, namely the leaf at the end of the primary path of y. Thus, no change for existing PLPs is needed. Here, again a secondary leaf node u is created, and thus its PLP needs to be created to point to itself.

Case 2-2: When y is secondary (Figure 10).
Procedure: Make w secondary, u primary, and initialize $PLP(w) = u$.
Correctness: In this case, the edge being split $x \to y$ is secondary, thus no existing PLP changed its primary path. Additionally, a new secondary node w is created, so $PLP(w)$ needs to be created and initialized. Since y is made to remain secondary, the primary child of w has to be the newly created u. Since u is a primary leaf child of w, it is the correct destination of $PLP(w)$.

4.3 Handling leaf deletions under root parent node

Here, we describe our procedure for deleting a leaf u, when its parent node w is the root. Note that no merging occurs in Cases 3-1–3-3 since w has no parent.

Case 3-1: When u is primary and w has no other children (Figure 11).
Procedure: Set $PLP(w) = w$.
Correctness: In this case, the tree consists only of the root after the update, thus the only PLP that exists is from the root w pointing to itself.

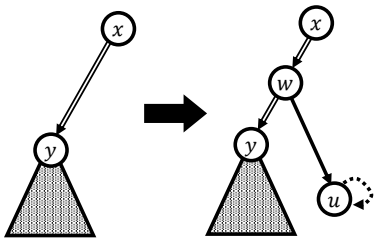


Figure 9. Case 2-1.

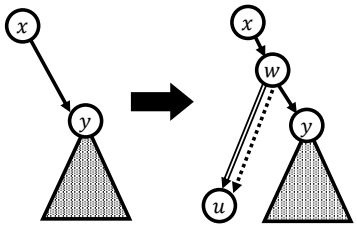


Figure 10. Case 2-2.

Case 3-2: When u is primary and w has at least one other child (Figure 12).
Procedure: Make y primary and set $PLP(w) = v$, where y is any secondary child of w and v is the leaf that $PLP(y)$ points to.
Correctness: Here, the primary child u of w is deleted, but w has at least one more child y remaining. Thus, y is changed into primary to maintain the invariant of child-having nodes having exactly one primary child. Consequently, y no longer has a PLP after the update. Also, the primary path from w changes into $w \to y \to \cdots$. Since the subtree of y remains unchanged, clearly the destination of this primary path, which is also the correct new destination of $PLP(w)$, is the former destination of $PLP(y)$ before the update.
Case 3-3: When u is secondary (Figure 13).
Procedure: No further updates.
Correctness: Since the node and edge deleted are secondary, and since no merging occurs, all existing primary paths remain the same, necessitating no change in existing PLPs that remain (i.e., undeleted).

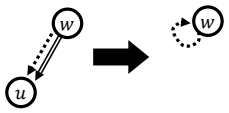


Figure 11. Case 3-1.

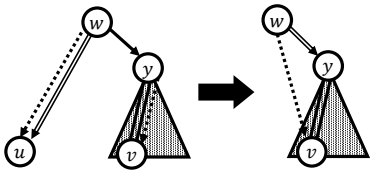


Figure 12. Case 3-2.

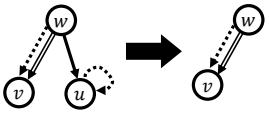


Figure 13. Case 3-3.

4.4 Handling primary leaf deletions under non-root parent node

Here, we describe our procedure for deleting a primary leaf u, when its parent node w is not the root. Let x be the parent of w, y any secondary child of w, v the leaf $PLP(y)$ points to, and z the node such that $PLP(z) = u$. Since w is not the root, it has at least two children.

Case 4-1: When w has more than two children (Figure 14).
Procedure: Make y primary, and set $PLP(z) = v$.
Correctness: In this case, z is w or its ancestor, and w does not get deleted. Here, the deletion of the primary node u and its incoming primary edge necessitates the appointment of another child of w to become primary, and also induces a change on the primary path of the PLP that pointed to u, namely that from z. Similarly to Case 3-2, y is chosen as the new primary child of w and thus the primary path from z now goes into y, and due to the unchanged subtree of y, v is the correct new destination of $PLP(z)$.

Case 4-2: When w has two children and is primary (Figure 15).
Procedure: Make y the primary child of x, and set $PLP(z) = v$.
Correctness: In this case, z is x or its ancestor. Since w, which was primary, becomes non-branching and gets deleted due to merging, its parent x needs to appoint a new primary child, for which y is chosen. Thus, the primary path from z now goes into y, and again we know the primary path destination from y is v, the former destination of $PLP(y)$ before the update, and therefore v is the correct destination of $PLP(z)$.
Case 4-3: When w has two children and is secondary (Figure 16).
Procedure: Make y a secondary child of x.
Correctness: In this case, $z = w$, and w gets deleted due to merging. Since w is secondary, the only affected primary path is the deleted path from w to u; all primary paths corresponding to remaining PLPs stay unchanged.

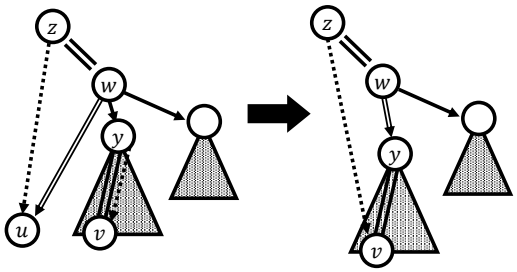


Figure 14. Case 4-1.

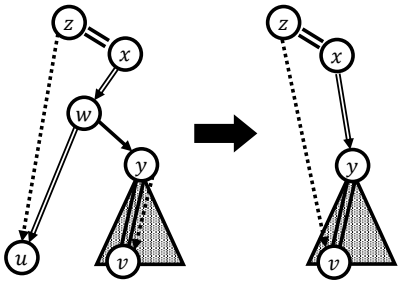


Figure 15. Case 4-2.

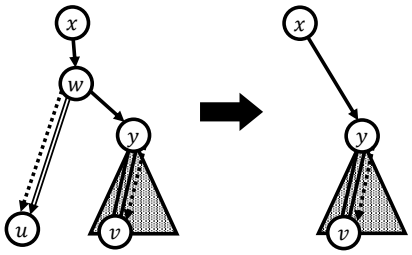


Figure 16. Case 4-3.

4.5 Handling secondary leaf deletions under non-root parent node

Here, we describe our procedure for deleting a secondary leaf u and its parent node w is not the root. Let x be the parent of w and y the primary child of w. Since w is not the root, it has at least two children.

Case 5-1: When w has more than two children (Figure 17).
Procedure: No further updates.
Correctness: In this case, w does not get deleted. Since only a secondary node is deleted and no merging occurs, no existing primary paths change.
Case 5-2: When w has two children and is primary (Figure 18).
Procedure: Make y the primary child of x.

Correctness: In this case, the deletion of u causes w to be deleted due to merging, and thus the primary path that included $x \to w \to y$ changes. However, the destination remains the same, namely the destination of the primary path from y. Thus, no existing PLPs change. Additionally, x remains having exactly one primary child, by letting y remain primary.

Case 5-3: When w has two children and is secondary (Figure 19).

Procedure: Make y a secondary child of x and set $PLP(y) = v$, where v is the leaf pointed to by $PLP(w)$.

Correctness: Here, the deletion of u causes w to be deleted due to merging. Since y is made secondary, the primary child of x remains unchanged, but $PLP(y)$ needs to be initialized. Since y was formerly primary, the former destination of $PLP(w)$ is in the unchanged subtree of y, namely leaf v, and therefore v is the correct destination of the new $PLP(y)$.

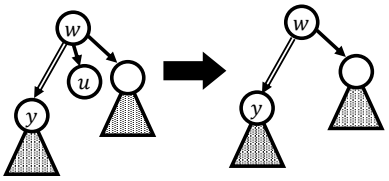


Figure 17. Case 5-1.

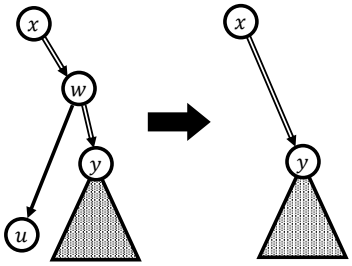


Figure 18. Case 5-2.

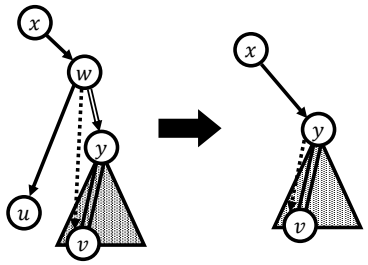


Figure 19. Case 5-3.

To illustrate the whole picture of our proposed method, we show in Figures 20 and 21 the decision trees of our leaf-pointer maintenance algorithm for the cases of right-end insertions and left-end deletions, respectively.

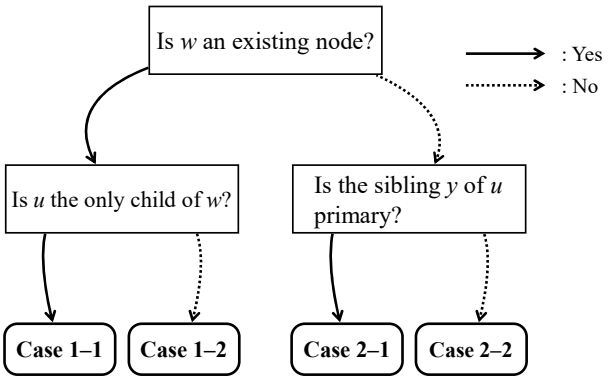


Figure 20. The decision tree representing the four cases of Sections 4.1 and 4.2, when a new node is inserted into the suffix tree where u is the new leaf and w is the parent of u.

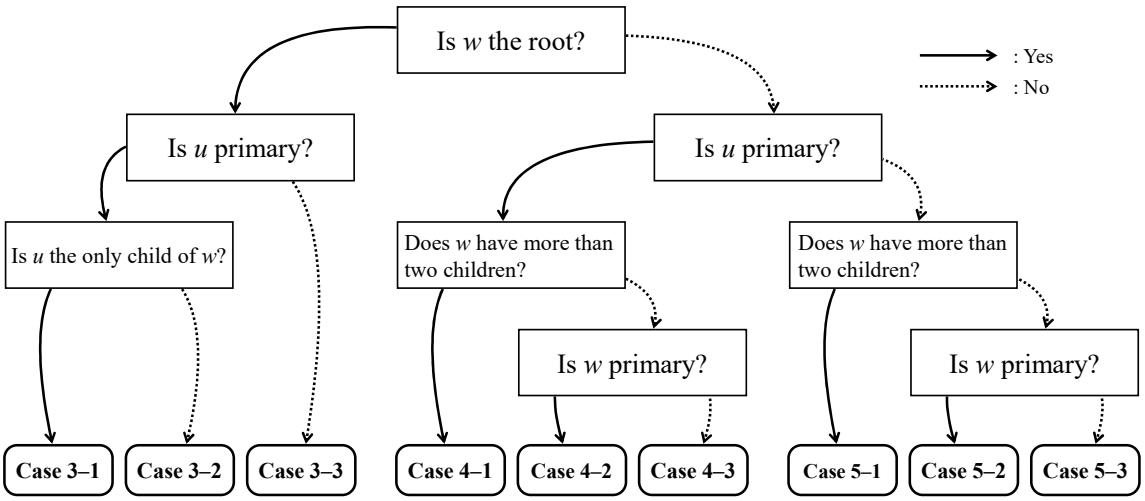


Figure 21. The decision tree representing the nine cases of Sections 4.3, 4.4, and 4.5, when an old leaf is deleted from the suffix tree where u is the leaf to be deleted and w is the parent of u.

We conclude this section with the following theorem:

Theorem 7. *Our algorithm described above correctly maintains leaf pointers on the sliding suffix tree for $W_i = T[\ell_i..r_i]$ in $O(1)$ time per node insertion/deletion, using $O(d)$ working space.*

Proof. The described steps to maintain the invariants cover all possible cases of insertion and deletion in a sliding suffix tree, and each step can clearly be handled in $O(1)$ worst-case time. This can also be observed from our decision trees in Figures 20 and 21. The correctness of all the cases has already been shown. This allows us to answer any leaf pointer query in $O(1)$ worst-case time. The working space is clearly $O(d)$. □

We remark that our leaf-pointer maintenance algorithm works in $O(1)$ worst-case time even when the window size d is not fixed, and the working space is linear in the current window size.

5 $\Theta(d)$-time worst-case lower bound for credit-based method

In this section, we present an instance that requires $\Theta(d)$ worst-case time per single leaf insertion or deletion with the credit-based method [5,6], where d is the window size.

Here, we briefly recall the credit-based method by Larsson [5,6], with a correction by Senft [9] for leaf deletions. When a leaf is created, then it issues a *credit* to its parent. Each internal node u maintains a credit counter $\mathit{credit}(u) \in \{0, 1\}$. We initially set $\mathit{credit}(u) = 0$ when an internal node u is created.

When an internal node u receives a credit from a child v, then there are two cases:

1. If $\mathit{credit}(u) = 0$, then we set $\mathit{credit}(u) = 1$.
2. Otherwise (if $\mathit{credit}(u) = 1$), then we first spend the existing credit in u by setting $\mathit{credit}(u) = 0$, and update the label of the edge leading to u. Second, we pass, to the parent of u, the credit that came from the child v.

When an internal node v is deleted and two adjacent edges $u \to v$ and $v \to w$ are merged into a single edge $u \to w$, then two credits are issued from v independently of the value of $\mathit{credit}(v)$. Then, one credit is used to update the label of the edge $u \to w$, and the other credit is passed to the parent u of the deleted node v.

While we omit the details in this paper, each edge label update can be done in constant time, and the above procedure keeps all edge labels valid (see [5,6,9] for details). At most n leaves are created and deleted for all sliding windows W_i with $1 \leq i \leq n-d+1$ over a string of length n. Since each leaf issues at most three credits (one for its insertion and two for its deletion), the total number of credits issued is at most $3n$. Since each edge label update spends a credit, the time for updating an edge label is amortized $O(1)$ per leaf insertion or deletion.

In the next theorem, we show that this credit-based method requires $\Theta(d)$ time per leaf insertion or deletion in the worst case.

Theorem 8. *There exists a string for which Larsson's credit-based sliding suffix tree algorithm [5,6] requires $\Theta(d)$ edge-label updates for some single leaf insertion or deletion, where d is the window size.*

Proof. The following credit configuration is reachable by the standard Ukkonen-style insertion phase used in the sliding suffix tree algorithm. Here we measure the worst-case cost of an individual leaf-insertion suboperation occurring during one window shift; the deletion suboperation in the same shift is not charged to this leaf insertion.

For the case of leaf insertion, consider a string that has $W = a^k b a^{k-1}$ as its prefix of length $2k = d$. We then append a new character c to the right end of W. See also Figure 22 for illustrations. Since our string begins with $W = a^k b a^{k-1}$, we first build $\mathsf{STree}(W)$ in an online manner, from left to right, as is done by Ukkonen's construction [10]. Below, we explain how the sliding suffix tree is built and updated.

- Build $\mathsf{STree}(a^k)$ that consists only of a single path (see also the first step of Figure 22).
- Then, we update it to $\mathsf{STree}(a^k b)$, in which the new leaves $a^h b$ and their parents a^h are inserted in descending order for $h = k-1, \ldots, 1$ (see also the second step of Figure 22). When the internal node a^h is inserted, we initially set $\mathit{credit}(a^h) = 0$.

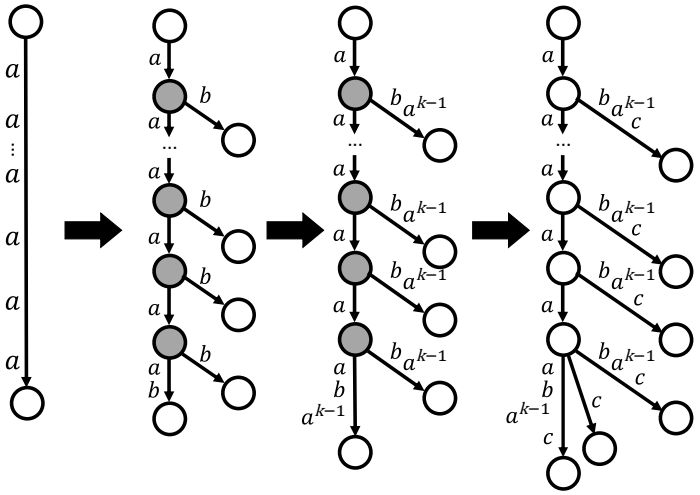


Figure 22. Illustration of $\Theta(d)$-time update of credit-based methods per single leaf insertion. The gray and white nodes represent internal nodes with credit value 1 and 0, respectively. Note that the subtrees starting with characters other than a are omitted for clarity.

As soon as the leaf $a^h b$ is inserted as a new child of a^h, then a^h receives a credit from this new leaf and we set $\mathit{credit}(a^h) = 1$. Thus, every node a^h for $1 \leq h \leq k-1$ obtains $\mathit{credit}(a^h) = 1$ on $\mathsf{STree}(a^k b)$.

- Next, we update $\mathsf{STree}(a^k b)$ to $\mathsf{STree}(a^k b a^{k-1})$ (see also the third step of Figure 22). No new leaves are inserted during this update.
- Finally, we update $\mathsf{STree}(a^k b a^{k-1})$ to $\mathsf{STree}(a^k b a^{k-1} c)$ (see also the fourth step of Figure 22). The first new leaf inserted is $a^{k-1}c$, which is a new child of a^{k-1}. Since $\mathit{credit}(a^{k-1}) = 1$, we reset it as $\mathit{credit}(a^{k-1}) = 0$ and pass the credit from the child $a^{k-1}c$ to the parent a^{k-2}. This propagates to all a^h in decreasing order of $h = k-1, \ldots, 1$.

Thus, the credit-based method requires $\Theta(d)$ edge-label updates for a single leaf insertion in the worst case.

For the case of leaf deletions, we consider a string $W' = a^d b$ of length $d+1$. See also Figure 23 for illustrations. We have $\mathit{credit}(a^h) = 1$ for all internal nodes a^h with $1 \leq h \leq d-1$ in $\mathsf{STree}(a^d b)$. Then we delete the leftmost character a from W' and update $\mathsf{STree}(a^d b)$ to $\mathsf{STree}(a^{d-1} b)$. This deletes the leaf corresponding to the longest suffix $a^d b$ and its parent a^{d-1} that becomes non-branching. After the deletion of the internal node a^{d-1}, a credit is sent to its parent a^{d-2} such that $\mathit{credit}(a^{d-2}) = 1$. We reset $\mathit{credit}(a^{d-2}) = 0$ and pass the credit to its parent a^{d-3}. This propagates to all a^h for decreasing $h = d-1, \ldots, 1$. Thus, the credit-based method requires $\Theta(d)$ edge label updates for a single leaf deletion in the worst case. □

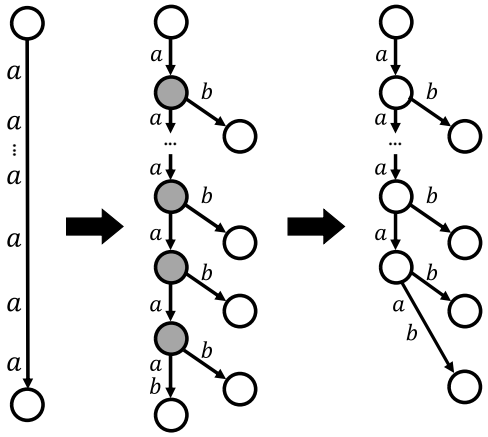


Figure 23. Illustration of $\Theta(d)$-time updates of credit-based methods per single leaf deletion.

Acknowledgements

The authors thank the anonymous referees of earlier versions of this paper for their helpful comments and suggestions for improvement. This work was supported by JSPS KAKENHI Grant Numbers JP23K24808, JP23K18466 (SI), JP24K02899 (HB), and JP24K20734 (TM).

References

1. A. Brodnik and M. Jekovec: *Sliding suffix tree.* Algorithms, 11(8) 2018, p. 118.
2. M. Crochemore, A. Héliou, G. Kucherov, L. Mouchard, S. P. Pissis, and Y. Ramusat: *Absent words in a sliding window with applications.* Inf. Comput., 270 2020, p. 104461.
3. E. R. Fiala and D. H. Greene: *Data compression with finite windows.* Commun. ACM, 32(4) 1989, pp. 490–505.
4. D. Gusfield: *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*, Cambridge University Press, Cambridge, 1997.
5. N. J. Larsson: *Extended application of suffix trees to data compression*, in Proceedings of the 6th Data Compression Conference (DCC '96), Snowbird, Utah, USA, March 31 - April 3, 1996, 1996, pp. 190–199.
6. N. J. Larsson: *Structures of String Matching and Data Compression*, PhD thesis, Lund University, Sweden, 1999.
7. T. Mieno, Y. Fujishige, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *Computing minimal unique substrings for a sliding window.* Algorithmica, 84(3) 2022, pp. 670–693.
8. T. Mieno and S. Inenaga: *Space-efficient online computation of string net occurrences*, in 36th Annual Symposium on Combinatorial Pattern Matching, CPM 2025, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 23:1–23:13.
9. M. Senft: *Suffix tree for a sliding window: An overview*, in 14th Annual Conference of Doctoral Students - WDS 2005, 2005, pp. 41–46.
10. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
11. P. Weiner: *Linear pattern matching algorithms*, in 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973, 1973, pp. 1–11.
12. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Trans. Inf. Theory, 23(3) 1977, pp. 337–343.