*algorithms*

MDPI

*Article*

# Sliding Suffix Tree

**Andrej Brodnik [1,2] and Matevž Jekovec [1,*]**

[1] Faculty of Computer and Information Science, University of Ljubljana, 1000 Ljubljana, Slovenia; andrej.brodnik@fri.uni-lj.si

[2] Faculty of Mathematics, Natural Sciences and Information Technologies, University of Primorska, 6000 Koper-Capodistria, Slovenia

* Correspondence: matevz.jekovec@fri.uni-lj.si; Tel.: +386-40-566543

Received: 20 June 2018; Accepted: 25 July 2018; Published: 3 August 2018

**Abstract:** We consider a sliding window $W$ over a stream of characters from some alphabet of constant size. We want to look up a pattern in the current sliding window content and obtain all positions of the matches. We present an indexed version of the sliding window, based on a suffix tree. The data structure of size $\Theta(|W|)$ has optimal time queries $\Theta(m + occ)$ and amortized constant time updates, where $m$ is the length of the query string and $occ$ is its number of occurrences.

**Keywords:** suffix tree; online pattern matching; sliding window

## 1. Introduction and Related Work

Text indexing, pattern matching, and big data in general is a well studied field of computer science and engineering. An especially intriguing area is (infinite) streams of data, which are too big to fit onto disk and cannot be indexed in the traditional way, regardless of the data compression or succinct representation used (e.g., [1–3]).

In practice, stream processing is usually done so that queries are defined in advance, and data streams are processed by efficient, carefully engineered filters. One way of implementing string matching including regular expressions is by using finite automata [4,5]. This approach is used by the well-known `grep` command. Instead of matching the pattern characters with the text, we could use a rolling hash and compare the pattern's *fingerprint* with the fingerprint of the current sliding window [6–8]. String matching is also used in digital forensics, where we typically match multiple regular expressions on massive amounts of data, which involves multiple streams and parallelism [9–11]. In intrusion detection systems, finite automata are replaced with neural networks, and by unsupervised machine learning we can avoid hard-coding malicious patterns [12,13].

In our research, we consider a slightly different setup. There is an infinite stream of characters, where the main memory holds the most recent characters in terms of a big sliding window. At any moment, a user wants to find all positions of the given substring in the current window. The stream processing filters described in the previous paragraph could easily solve this issue, if the queries were known in advance. However, since the queries are provided on-the-fly, the filters would need to re-scan the sliding window, which requires at least linear work in terms of the size of the window. The only way to speed up the query for the order of magnitude is to *index* the sliding window.

Fiala and Greene [14] defined an indexed sliding window based on a suffix tree, and they based their work on McCreight's suffix tree construction algorithm [15]. Their primary application was data compression, and the main contribution was the maintenance of the indexed sliding window. However, the query operation was different from ours. For data compression, they wanted to determine *the length and the position of the longest prefix* of the given pattern in the sliding window. Larsson revised work of Fiala and Greene in his PhD thesis [16], and presented it in a new light using the primitives from Ukkonen's more recent suffix tree construction algorithm [17]. Larsson also provided a reference implementation of his data structure. Furthermore, Senft revisited both works by Fiala, Greene, and Larsson and noticed a flaw in the window maintenance proof [18]. Fortunately, the underlying data structure was designed correctly and he only needed to correct the proof.

Another approach for indexing the sliding window is by using the compact directed acyclic word graph, also known as the suffix automaton [19]. The suffix automaton can be constructed on-line for the whole text [20] or maintained as a sliding window [21]. Both approaches use Ukkonen's suffix tree construction algorithm as a base. Originally, the suffix automaton was developed to efficiently decide whether the query string is included in the text. However, obtaining *all positions* of the query in the text requires performing depth first traversals over the automaton and takes linear time in terms of the size of the sliding window. For our application, we cannot afford this time.

Recently, a practical suffix array based a sliding window was proposed by Ferreira et al. [22,23], with speed improvements by Salson et al. [24] . Their approach supports efficient substring query operations, but updating the suffix array requires at least linear time due to the nature of the array, i.e., insertion and/or removal of an element requires the other elements to shift by one slot. Consequently, the time needed to perform the updates grows quadratically in terms of the sliding window size. For smaller sliding window sizes, the approach seems feasible (the authors reported measurements for windows up to $32,768$ characters long). However, our aim is to maintain a sliding window of the size of the main memory, therefore the suffix array as proposed by Ferreira et al. is not appropriate.

In this paper, we extend previous work by Fiala, Greene, Larsson, and Senft. Our main contribution is an indexed sliding window which allows a user to find *all positions* of an arbitrary substring in optimal time and space. It turns out that this operation is not trivial, due to details hidden in Ukkonen's suffix tree construction algorithm. This is the first data structure for on-the-fly text indexing which requires amortized $O(1)$ time for updates and worst case optimal time for queries.

In the following section we define the notation and preliminary data structures and algorithms. In Section 3 we formally present a sliding suffix tree, and we conclude in Section 4 with the discussion and open problems.

## 2. Notation and Preliminaries

Capital letters $A, B, C \ldots$ denote strings of characters from alphabet $\Sigma$, where $|\Sigma| = O(1)$, and lower case letters $i, j \ldots$ denote integers, except for $c$, which denotes an arbitrary character from $\Sigma$ . Furthermore, lower case Greek letters $\alpha, \beta \ldots$ represent nodes in a tree, and calligraphic capital letters like $\mathcal{T}$ represent tree-based data structures. We denote concatenation of two strings by simply writing one string beside the other, e.g., $AB$, and the length of a string $A$ as $|A|$. By $A[i : j]$ we denote a substring of $A$ starting at position $i$ and ending at $j$ inclusive, where $1 \leq i \leq j \leq |A|$. A suffix of $A$ starting at $i$ is $A[i :]$ and a prefix of $A$ ending at $j$ is $A[: j]$, both inclusive.

We denote by $W$ the sliding window over an infinite input stream of characters which are from an alphabet of constant size. By $n$, we denote the number of all characters read so far. To store a suffix starting at the current position, we store the current $n$. At any later time, we can retrieve the contents of this suffix as $W[n' - (n - |W|) :]$, where $n'$ is the stored position of the suffix and $n - n' < |W|$ for the suffix to be present in $W$. For clarity, by $n$ we denote the number of *all* characters read so far. In practice, however, the sliding window content is usually stored as a circular buffer in the main memory. Therefore, $n$ is always bounded by $|W|$, and the suffix at position $n'$ can be retrieved as $W[n' : n]$ if $n \geq n'$, or $W[n' :]W[: n]$ if $n < n'$.

### 2.1. Suffix Tree

A *suffix tree* is a dictionary containing each suffix of the text as a key, and its position in the text as its value. The data structure is implemented as a compressed trie. To reduce space, we only store edge labels implicitly, so that each internal node stores the *start index* of the label on the incoming edge and its *length*, whereas each leaf stores the *position* of the suffix it represents in the text (see Figure 1). Additionally, each node $\alpha$ stores a *shortcut to any of the descendant leaves* denoted by $leaf(\alpha)$. If $\alpha$ is a leaf, then $leaf(\alpha) = \alpha$. During the shift operation of the window, this shortcut will be updated so that it always points to a valid leaf. For example, on Figure 1 the leaf shortcut from the internal node labeled 1:1 could point to a leaf labeled 11, 8, 1, 4, or 6.

We denote by $|\alpha|$ a *string depth* operation of some node $\alpha$ in a suffix tree, and define it as a sum of all label lengths of edges from the root to $\alpha$. Consequently, a label is a substring of a suffix starting and ending at the string depth of the originating node and the string depth of the terminating node, respectively. We say a node $\alpha$ *spells out* string $A$, where $A$ is a concatenation of all labels from the root to $\alpha$. For example, the only internal node of depth 2 in Figure 1 (node with value 2:3) has a string depth of 4, and spells out a string ABRA.

Let $A$ and $A'$ be strings which are spelled out by nodes $\alpha$ and $\alpha'$, respectively. We define a suffix link as an edge from node $\alpha'$ to $\alpha$ if $A = A'[2:]$, and denote this by $\alpha = \text{suffix\_link}(\alpha')$. Suffix links in Figure 1 are represented as green dotted lines.

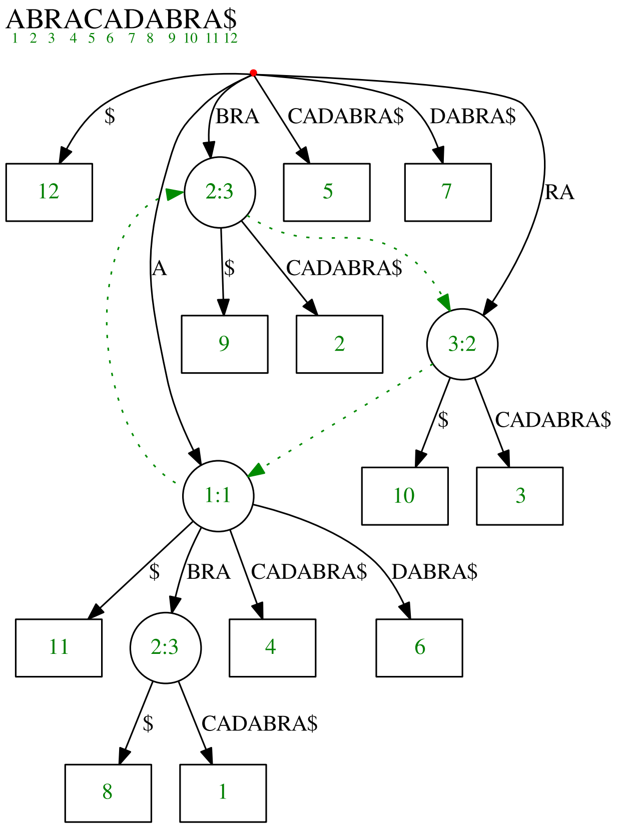

**Figure 1.** Illustration of the suffix tree for text ABRACADABRA$ including the implicit labels. Suffix links connecting internal nodes are drawn as dotted green lines. Values in each internal node represent the start index and the length of a label on the incoming edge, whereas the value of a leaf represent the suffix position in the text.

## 2.2. Ukkonen's Online Suffix Tree Construction Algorithm

In Reference [17], Ukkonen presented an incremental suffix tree construction algorithm which builds the data structure in a single pass. During the construction, the algorithm maintains the following invariants in amortized constant time:

- Implicit buffer $B$, which corresponds to the longest repeated *suffix* of the text processed so far,
- the *active node* $\beta$, which represents a node where we end up by navigating $B$ in the suffix tree constructed so far, i.e., $B$ is a prefix of a string spelled out by $\beta$.

The execution of the algorithm can be viewed as an automaton which reads a character $c$ from the input stream. The automaton starts and remains in the *buffering state* while $c$ matches the next character of the incoming edge's label of $\beta$. If the end of the label is reached (i.e., $|B| = |\beta|$), a child in the direction of $c$ is taken, and this child becomes a new active node $\beta$. Finally, $c$ is (implicitly) appended to $B$.

On the other hand, if $B$ cannot be extended by $c$ in a tree, the automaton enters the *expanding state*. First, it inserts a new branch in the direction of $c$, with a single leaf storing the suffix position $n - |B| + 1$. If $|B| = |\beta|$, the new branch is added as a child to $\beta$. Otherwise, if $|B| < |\beta|$, the incoming edge of $\beta$ is split, such that the string depth of the newly inserted internal node is $|B|$, and the new branch is added to this node. Once the branch is inserted, the first character is removed from $B$, obtaining new $B' = B[2:]$. A new active node corresponding to $B'$ is found in the following way. Let $\alpha$ denote the parent of the original active node $\beta$. Then the new active node $\beta'$ is a node obtained by navigating suffix $B'[|\alpha|:]$ from a node suffix_link$(\alpha)$. When $\beta'$ is obtained, $c$ is reconsidered. If a branch in the direction of $c$ exists, the automaton switches back to buffering state. Otherwise, it remains in the expanding state and repeats the new branch insertion. Each time the expanding state is re-entered, $B$ is shortened by one character. In the worst case, if $c$ does not occur yet in the text, the suffix links will be followed all the way up to the root node, and $c$ will be added as a new child to the root node. In this case, the implicit buffer $B$ will be an empty string.

To wrap up, the currently constructed suffix tree is *un-finalized*, until $B$ is completely emptied. Moreover, there are exactly $|B|$ leaves missing in the un-finalized tree, and they correspond to suffixes of $B$. For finite texts, we finalize the suffix tree at the end by appending a unique character \$, which forces the algorithm to empty $B$ and finalize the tree. For infinite streams, however, there is no final character. Consequently, we need to support:

1. Queries: When performing a query, we need to report the occurrences, both in the partially constructed suffix tree and in $B$.
2. Maintenance: The original Ukkonen's algorithm supports adding a new character to the indexed text. When a window is shifted, we also remove the oldest (longest) suffix from the text.

## 3. Sliding Suffix Tree

The *sliding suffix tree* is an indexed version of the current sliding window content $W$. Formally, we define two operations:

- find$(W, Q)$ — returns all positions of the query string $Q$ in $W$ (obviously, $|Q| \leq |W|$) .
- shift$(W, c)$ — appends a character $c$ to $W$ and removes the oldest character from $W$.

Initially, $W$ is empty, and until the length of $W$ reaches the desired size, the shift operation only appends new characters.

We build the sliding suffix tree on top of Ukkonen's online suffix tree construction algorithm in a similar way to Larsson's approach [16]. We maintain a possibly un-finalized suffix tree $\mathcal{T}$, including the implicit buffer $B$ and the active node $\beta$ (Figure 2 on the left). Figure 2 on the right illustrates the position of $W$ and $B$ in a stream. Notice that $B$ is always a proper suffix of $W$.

In the next two subsections we show how to perform the find operation in time $\Theta(m + occ)$ in the worst case and the shift operation in amortized $O(1)$ time. As a model of computation, we use the standard RAM model.

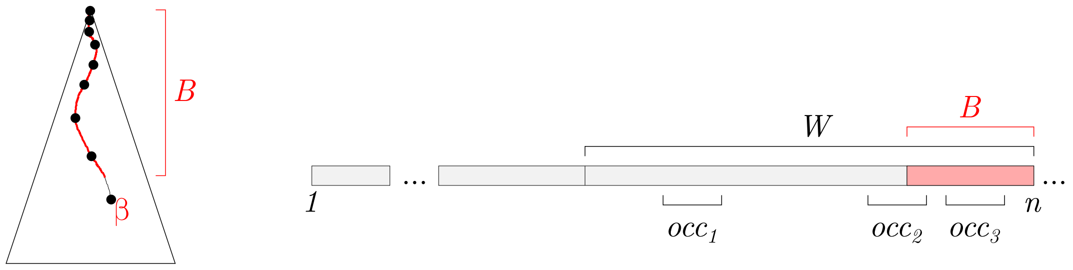

**Figure 2.** On the left: Illustration of partially constructed suffix tree $\mathcal{T}$ with implicit buffer $B$ and active node $\beta$. On the right: Illustration of the stream, the sliding window $W$, the implicit buffer $B$, and three cases for positions of the query strings $occ_1$, $occ_2$, and $occ_3$.

## 3.1. Queries

To find all occurrences of query $Q$ in $W$, we first navigate $Q$ in $\mathcal{T}$. Let $\mathcal{T}_Q$ correspond to a subtree rooted at the node where we finished the navigation. Leaves of $\mathcal{T}_Q$ make up the first part of the resulting set. In Figure 2, $occ_1$ corresponds to this type of occurrence. Also, the position of $occ_2$ in the same figure will be contained in one of the leaves of $\mathcal{T}_Q$, since $\mathcal{T}$ contains all suffixes that *start* at the beginning of $W$, up to the beginning of $B$.

The second part of the resulting set are the missing leaves of $\mathcal{T}_Q$ due to the un-finalized state of $\mathcal{T}$. More formally, we want to find the missing leaves which correspond to suffixes of $B$ beginning with $Q$. $occ_3$ in Figure 2 illustrates one such position. We consider three cases:

Case 1: $|B| < |Q|$

Obviously, there are no matches of $Q$ in $B$, and we solely return the leaves of $\mathcal{T}_Q$.

Case 2: $|B| = |Q|$

We check whether the active node $\beta$ is the root node of $\mathcal{T}_Q$. If this is true, then $B$ equals $Q$, and we add position $n - |B| + 1$ to the resulting set.

Case 3: $|B| > |Q|$

First, we claim that the navigated subtree $\mathcal{T}_Q$ always exists, if there are any occurrences of $Q$ to be found in $B$.

**Lemma 1.** *If $Q$ exists in buffer $B$, then a subtree $\mathcal{T}_Q$ exists by navigating the query $Q$ in $\mathcal{T}$.*

**Proof.** If $Q$ exists somewhere in $B$, then $Q$ is a substring of the string spelled out by $\beta$. From the property of the suffix tree, by following the suffix links from $\beta$ we will find a root of $\mathcal{T}_Q$ which spells out a string starting with $Q$. $\square$

In this case, we find all occurrences of $Q$ in two steps:

**Step 1: Map positions of $Q$ from $\mathcal{T}_Q$ to $B$**

We consider the relation of each leaf in $\mathcal{T}_Q$ to $\beta$. Take $leaf(\beta)$ and let this leaf store some position $x$. Then, for each leaf in a subtree rooted at $\mathcal{T}_Q$, let $i$ correspond to the position stored in a leaf. If $x \leq i \leq x + |B| - |Q|$, then $Q$ also occurs at position $n - |B| + (i - x) + 1$ . We add this position to the resulting set.

By the definition of $leaf$, $x$ corresponds to the position stored in *any* descendant leaf of $\beta$. It's fine if $x \leq n - 2|B|$, as the leaves of $\mathcal{T}_Q$ store a superset of positions relative to all leaves of $\beta$, due to the condition $|\beta| \geq |\mathcal{T}_Q|$. Therefore, taking any descendant leaf of $\beta$ suffices to compute *all* occurrences of $Q$ in $B$ once we visit all leaves of $\mathcal{T}_Q$. However, if $x > n - 2|B|$, we need to proceed to Step 2.

**Step 2: Find all repeated occurrences of $Q$ in $B$**

Additional occurrences of $Q$ may exist in $B$ if $x > n - 2|B|$, because the suffix tree (and consequently $\mathcal{T}_Q$) is finalized only up to position $n - |B|$. Observe in Figure 3a that the string $B$ starting at $x$ overlaps the string $B$ starting at $n - |B| + 1$ . Because two equal strings overlap, the first string determines the beginning of the second one and, consequently, also the beginning of the first one. This overlapping yields a repetitive pattern inside $B$, and if $Q$ occurs in the beginning of $B$ then it will also occur at each subsequent repetition of this pattern. The following lemma formally defines the repetitive pattern $P$ and Figure 3b illustrates the occurrences of $P$ in $B$.

**Lemma 2** (**The Buffer Pumping Lemma**). *Let $leaf(\beta)$ store the position $x$, such that $x > n - 2|B|$. Then, $B$ consists of repetitive patterns $P$, where $P = W[x : n - |B|]$.*

**Proof.** Observe two occurrences of $B$ in Figure 3a. Let $P = W[x : n - |B|]$. Due to overlapping, string $W[x : x + |P| - 1]$ equals string $W[n - |B| + 1 : n - |B| + |P|] = B[1 : |P|]$ . Consequently, due to new overlapping, $W[x + |P| : x + 2|P| - 1]$ now equals $W[n - |B| + |P| : n - |B| + 2|P| - 1] = B[|P| + 1 : 2|P|]$ . Repeating this observation, we construct $B = P^k P'$ where $k = \lfloor \frac{|B|}{|P|} \rfloor$ and the last repetition $P'$ is a prefix of $P$, and may be empty. □

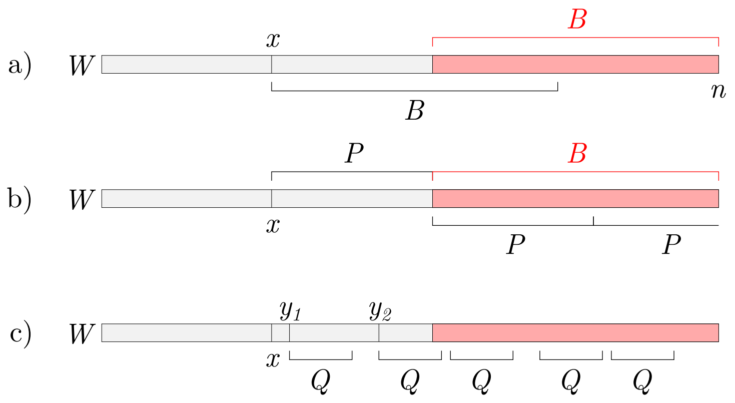

**Figure 3.** (**a**) Occurrences of $B$ in the window, if $x > n - 2|B|$. (**b**) Repeated pattern $P$. (**c**) Potential occurrences of $Q$. $y_1$ and $y_2$ are positions stored in the finalized suffix tree. The second two positions of $Q$ are obtained in Case 3, Step 1. The last position of $Q$ is obtained in Case 3, Step 2.

Let $Y = y_1, y_2, \ldots$ represent positions stored in such leaves of $\mathcal{T}_Q$ that $n - 2|B| + 1 \le y_i < n - |B|$ for each $y_i \in Y$. $y_1$ and $y_2$ in Figure 3c represent two such positions. In Step 1, we computed positions of $Q$ inside the first occurrence of $P$ in $B$, as seen in Figure 3c by the third and the fourth $Q$. To compute subsequent positions of $Q$ in the repetitions of $P$ in $B$, for each $y_i \in Y$ we add positions $y_i + |P|, y_i + 2|P| \ldots$ up to $n - |Q| + 1$ to the resulting set. This approach will add all remaining occurrences of $Q$ in $B$ and complete the query. The fifth occurrence of $Q$ in Figure 3c is in the repetition of $P$ in $B$.

Algorithm 1 combines all three cases presented above in a simplified, clear way.

**Algorithm 1:** Query procedure in the sliding suffix tree.

```
Input: Query $Q$, Unfinalized suffix tree $\mathcal{T}$, Active node $\beta$, Buffer $B$, Current position in a
       stream $n$
Output: Positions of $Q$ in the window
S ← empty list
$x \leftarrow leaf(\beta)$
$\mathcal{T}_Q \leftarrow navigate(\mathcal{T}, Q)$
for each leaf $v$ in $\mathcal{T}_Q$ do
    $y \leftarrow position(v)$
    add $y$ to $S$
    if $x \leq y < x + |B|$ then
        $|P| \leftarrow n - |B| - x$
        $y' \leftarrow y + |P|$
        while $y' \leq n - |Q|$ do
            add $y'$ to $S$
            $y' \leftarrow y' + |P|$
return $S$
```

Time Complexity

The time complexity of the query first requires $O(m)$ time to navigate $Q$ in $\mathcal{T}$, and then time to solve the three cases above. The first two cases are solved in $O(1)$ time. Step 1 of the third case is solved in $\Theta(|\mathcal{T}_Q|) = O(occ)$ time, since we add at most one position per leaf of $\mathcal{T}_Q$ to the resulting set. Step 2 of the third case may add multiple positions per leaf of $\mathcal{T}_Q$, and is bounded by $O(occ)$ time in the worst case.

**Theorem 1.** *We can find all occurrences of query $Q$ in a sliding suffix tree in time $\Theta(m + occ)$ for constant alphabet size.*

## *3.2. Maintenance*

The maintenance of the sliding suffix tree is based on the work by Larsson and Senft [16,18], with our addition of maintaining leaf shortcuts in internal nodes.

Path Compression ([16], p. 26)

During the expanding state of Ukkonen, we add to $\mathcal{T}$ either one node (a new leaf is added to the active node) or two nodes (the incoming edge of the active node is split and a new leaf is added). When the expanding state is visited the first time, a new internal node $\gamma'$ is added to $\mathcal{T}$. In the next step, either an expanding state is re-entered or a buffering state is entered. If the expanding state is re-entered, we repeat the procedure, obtaining a new node $\gamma$ and a suffix link from $\gamma'$ to $\gamma$ . If the buffering state is entered, either a root node or a node containing the matched character is reached. Instead of creating a new node in $\mathcal{T}$ as we did in the expanding state, we only add a suffix link to an existing node in $\mathcal{T}$.

To remove the oldest stored suffix from $\mathcal{T}$, we first find the corresponding leaf (e.g., by taking the next leaf from some queue storing all inserted leaves). If the leaf's parent has three or more children, the parent remains unchanged and we just remove the leaf from $\mathcal{T}$. On the other hand, if the leaf's parent has exactly two children, we remove the leaf from $\mathcal{T}$ and also its parent $\gamma$ from $\mathcal{T}$. To remove $\gamma$, we merge its incoming edges and the remaining outgoing edges.

**Lemma 3.** *Let $\gamma$ be a node with two children in $\mathcal{T}$, where one child $\omega$ is a leaf storing the position of the longest suffix $n - |W| + 1$ . Then, $\gamma$ is not a terminating node of any suffix link.*

**Proof by contradiction.** Let a subtree $\tau$ be the other child of $\gamma$. Assume there is $\gamma'$, which has suffix link terminating at $\gamma$. Because $\gamma'$ exists, it also has to have at least two children. However, since the subtree rooted at $\gamma'$ is the subtree rooted at $\gamma$ and the latter has only children $\omega$ and $\tau$, $\gamma'$ has respective children $\omega'$ and $\tau'$. Observe that $\omega$ spells out a string $W$, and, consequently, because of the suffix link relationship $\omega'$ spells out a string $cW$ for some character $c \in \Sigma$. Furthermore, since $W$ is the longest string in the window, $cW$ cannot be suffix in $W$, which contradicts the original assumption. To wrap up, there is no suffix link terminating at $\gamma$. □

Avoiding Unwanted Suffix Removals and Keeping a Valid Insertion Point ([16], pp. 26–28)

At the moment of the oldest suffix removal, the corresponding leaf can be an active node $\beta$. If this is the case, then $B$ appears at two positions in the window: At the very beginning as a prefix of the removed suffix and at the very end as a suffix of the window which has not been finalized yet . Recall that, by definition, $B$ corresponds to the longest repeated suffix of the window. Consequently, if we shift the window for one position to the right, the new longest repeated suffix becomes $B[2:]$. By doing this, we also need to execute the expanding state of Ukkonen's algorithm and add a leaf storing the position $n - |B| + 1$, corresponding to the original $B$. We can combine the removal of the oldest leaf and the addition of the new one by simply updating the position stored in the oldest leaf from $n - |W| + 1$ to $n - |B| + 1$. Figure 4 illustrates this case.

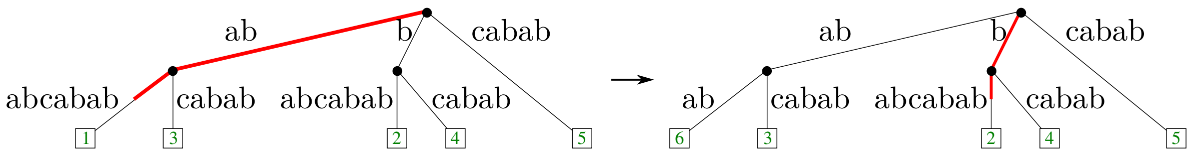

**Figure 4.** Shifting the window *ababcabab* to the right for one character, where $B = abab$ denoted by red labels. Notice the "lost" leaf *abab* on the right, where instead of removing the oldest leaf, we only update its position. For brevity, the figure omits the newly appended character to the window.

To find a new $\beta$ and an edge corresponding to the updated $B$, we simply follow the suffix link of the $\beta$'s parent and navigate the remainder of $B$ from the obtained node.

Keeping Edge Labels and Leaf Shortcuts Valid ([16], pp. 28–30)

The final step of the maintenance is to update internal node labels and shortcuts to leaves ([16], pp. 28–30). Recall that the edge labels are implicit, and they only store start and end references to the text. In time, these references might become stale and point to positions outside $W$. To keep labels up-to-date, on each removal of a leaf we could take the position stored in one of its siblings and update all labels on a root to removed the leaf's path. However, this would require $O(|W|)$ time. We amortize this time by assigning to each internal node a credit, which is initially 0. When adding or deleting a leaf, the leaf issues a credit to its parent. The parent updates its label position according to its child and increases its credit by 1. If the credit is already 1, beside updating its label, it also recursively issues a credit to its parent and clears the credit to 0. Let an ancient node be an internal node which contains stale labels pointing to positions outside $W$. This approach will update stale labels in amortized $O(1)$ time. The following theorem assures the labels stored in internal nodes are always up-to-date:

**Theorem 2.** *Every ancient node received a fresh credit from all subtrees rooted in its children.*

**Proof.** See Reference [18], p. 45. □

Similar to updating edge labels, we update shortcuts from the internal node to the leaf. An internal node can store a shortcut to *any* valid leaf in its subtree. When adding a new leaf, we can recursively propagate its reference to the parent. When removing a leaf, we propagate a reference to one of its siblings to the parent. By using the credit system, shortcuts will be updated the same way as edge labels in amortized $O(1)$ time.

Time Complexity

Ukkonen's algorithm adds a suffix to $\mathcal{T}$ in amortized $O(1)$ time [17]. To remove the longest leaf and its parent, we require $O(1)$ time. To update find new $\beta$, we require $O(1)$ time to follow the suffix link, and amortized $O(1)$ time to navigate the remainder of $B$ in $\mathcal{T}$. Edge labels and leaf shortcuts are also updated in constant $O(1)$ time, because each new leaf will trigger exactly one update to the internal node.

**Theorem 3.** *The sliding suffix tree can be shifted in amortized $\Theta(1)$ time.*

## 4. Conclusions and Open Problems

In this paper, we presented a sliding suffix tree for performing online substring queries on a stream of characters from the alphabet of constant size. By extending previous work by Fiala, Greene, Larsson, and Senft [14,16,18], we designed a novel data structure, the use of which can find all positions of the given substring inside the window in optimal $\Theta(m + occ)$ time, where $m$ is the length of the query and $occ$ the number of its occurrences. Furthermore, the updates are done in amortized $O(1)$ time.

An open question remains whether the data structure can be updated in the worst case $O(1)$ time. There is a well known linear-time suffix-sorting lower bound [25], but to our knowledge, no per-character lower bound has been explored. Ukkonen's algorithm requires, by design, an amortized $O(1)$ time for updates, due to the implicit buffer of un-finalized nodes. To the best of our knowledge, no other online suffix tree construction algorithms have been developed without such an implicit buffer.

As stated, we assume the alphabet $\Sigma$ of constant size. For $\Sigma$ of arbitrary size, the current implementation of the suffix tree requires an additional factor of $\lg |\Sigma|$ time to determine a child in a tree, but keeping the same space complexity. An interesting question is whether the same asymptotic times can be achieved for integer alphabets as was done in [25] for texts of fixed length. In our case $|\Sigma| = O(|W|)$, but the alphabet can change in time.

Streaming algorithms are common in heavy throughput environments and therefore it is desirable to involve parallelism, including distributed processing. Most of the research involving parallelism assumes either a static text (parallel construction of a suffix tree, e.g., [26–29]) or a static data structure (parallelization of a query, e.g., [30,31]). On the other hand, because the data structure between updates does not change, individual queries can generally be done in parallel. In our case, the updates are coming sequentially, which raises the question of whether the presented data structure can be adjusted to support shifts in a pipeline fashion. Moreover, going to the underlying Ukkonen's algorithm, it is not known whether it is possible to parallelize an update itself. Finally, coming back to our case, it is an open question of how to process interleaving queries and updates in parallel.

The presented data structure, while theoretically feasible, should also be competitive in practice. Larsson already provided code for maintaining the indexed sliding window ([16], Appendix A), but without reporting all substring positions. However, the code can be extended to report them. In general, the main practical concern of tree-based data structures is space consumption and cache inefficiency. To alleviate this, the data structure should be implemented succinctly (e.g., [32]), with an appropriate cache-efficient scheme (e.g., [33]).

**Author Contributions:** The authors contribute equally to this work.

**Funding:** The work is partially supported by the Slovenian Research Agency (research core funding No. P2-0359 and N2-0053).

**Acknowledgments:** We would like to thank anonymous reviewers of Combinatorial Pattern Matching 2018 for valuable references to the related work and their ideas for simplifying the queries without using the lowest common ancestor auxiliary data structures used in the previous version of this paper [34].

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Navarro, G.; Mäkinen, V. Compressed full-text indexes. *ACM Comput. Surv.* **2007**, *39*, 2. [CrossRef]
2. Abouelhoda, M.I.; Kurtz, S.; Ohlebusch, E. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms* **2004**, *2*, 53–86. [CrossRef]
3. Grossi, R.; Gupta, A.; Vitter, J.S. High-Order Entropy-Compressed Text Indexes. In Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, USA, 12–14 January 2003; Volume 2068, pp. 841–850.
4. Knuth, D.; Morris, J., Jr.; Pratt, V. Fast Pattern Matching in Strings. *SIAM J. Comput.* **1977**, *6*, 323–350. [CrossRef]
5. Boyer, R.S.; Moore, J.S. A fast string searching algorithm. *Commun. ACM* **1977**, *20*, 762–772. [CrossRef]
6. Karp, R.M.; Rabin, M.O. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* **1987**, *31*, 249–260. [CrossRef]
7. Ergun, F.; Jowhari, H.; Sağlam, M. Periodicity in Streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*; Serna, M., Shaltiel, R., Jansen, K., Rolim, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 545–559.
8. Breslauer, D.; Galil, Z. Real-Time Streaming String-Matching. In *Combinatorial Pattern Matching*; Giancarlo, R., Manzini, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 162–172.
9. Casey, E. *Handbook of Digital Forensics and Investigation*, 2nd ed.; Elsevier Academic Press: Burlington, MA, USA, 2009.
10. Cox, R. Regular Expression Matching: the Virtual Machine Approach, 2009. Available online: https://swtch.com/rsc/regexp/ (accessed on 2nd August 2018).
11. Stewart, J.; Uckelman, J. Searching Massive Data Streams Using Multipattern Regular Expressions. In *Advances in Digital Forensics VII*; Peterson, G., Shenoi, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 49–63.
12. Agravat, D.; Vaishnav, U.; Swadas, P.B. Modified Ant Miner for Intrusion Detection. In Proceedings of the 2010 Second International Conference on Machine Learning and Computing, Bangalore, India, 9–11 Febuary 2010; pp. 228–232.
13. Kukielka, P.; Kotulski, Z. Analysis of neural networks usage for detection of a new attack in IDS. *Ann. UMCS Inform.* **2010**, *10*, 51–59. [CrossRef]
14. Fiala, E.R.; Greene, D.H. Data compression with finite windows. *Commun. ACM* **1989**, *32*, 490–505. [CrossRef]
15. McCreight, E.M. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM* **1976**, *23*, 262–272. [CrossRef]
16. Larsson, N.J. Structures of String Matching and Data Compression. Ph.D. Thesis, Lund University, Lund, Sweden, 1999.
17. Ukkonen, E. On-Line Construction of Suffix Trees. *Algorithmica* **1995**, *14*, 249–260. [CrossRef]
18. Senft, M. Suffix tree for a sliding window: An overview. In *Week of Doctoral Students*; Šafránková, J., Ed.; Matfyzpress: Prague, Czech Republic, 2005; pp. 41–46.
19. Blumer, A.; Blumer, J.; Haussler, D.; McConnell, R.; Ehrenfeucht, A. Complete Inverted Files for Efficient Text Retrieval and Analysis. *J. ACM* **1987**, *34*, 578–595. [CrossRef]
20. Inenaga, S.; Hoshino, H.; Shinohara, A.; Takeda, M.; Arikawa, S.; Mauri, G.; Pavesi, G. On-line construction of compact directed acyclic word graphs. *Discrete Appl. Math.* **2005**, *146*, 156–179. [CrossRef]
21. Inenaga, S.; Shinohara, A.; Takeda, M.; Arikawa, S. Compact directed acyclic word graphs for a sliding window. *J. Discrete Algorithms* **2004**, *2*, 33–51. [CrossRef]

22. Ferreira, A.; Oliveira, A.; Figueiredo, M. On the Use of Suffix Arrays for Memory-Efficient Lempel-Ziv Data Compression. In Proceedings of the 2009 Data Compression Conference, Snowbird, UT, USA, 16–18 March 2009; p. 444.
23. Ferreira, A.; Oliveira, A.; Figueiredo, M. Sliding Window Update Using Suffix Arrays. In Proceedings of the 2011 Data Compression Conference, Snowbird, UT, USA, 29–31 March 2011; p. 456.
24. Salson, M.; Lecroq, T.; Léonard, M.; Mouchard, L. A four-stage algorithm for updating a Burrows–Wheeler transform. *Theor. Comput. Sci.* **2009**, *410*, 4350–4359. [CrossRef]
25. Farach-Colton, M.; Ferragina, P.; Muthukrishnan, S. On the sorting-complexity of suffix tree construction. *J. ACM* **2000**, *47*, 987–1011. [CrossRef]
26. Barsky, M.; Stege, U.; Thomo, A.; Upton, C. Suffix trees for very large genomic sequences. In Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09, Hong Kong, China, 2–6 November 2009; ACM Press: New York, NY, USA, 2009; p. 1417.
27. Mansour, E.; Allam, A.; Skiadopoulos, S.; Kalnis, P. ERA: Efficient serial and parallel suffix tree construction for very long strings. *Proc. VLDB Endow.* **2011**, *5*, 49–60. [CrossRef]
28. Comin, M.; Farreras, M. Efficient parallel construction of suffix trees for genomes larger than main memory. In Proceedings of the 20th European MPI Users' Group Meeting on EuroMPI '13, Madrid, Spain, 15–18 September 2013; ACM Press: New York, NY, USA, 2013; p. 211.
29. Shun, J.; Blelloch, G.E. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Trans. Parallel Comput.* **2014**, *1*, 1–20. [CrossRef]
30. Jekovec, M.; Brodnik, A. *Parallel Query in the Suffix Tree*; Technical Report; University of Ljubljana, Faculty of Computer and Information Science: Ljubljana, Slovenia, 2015.
31. Christiansen, A.R.; Farach-Colton, M. Parallel Lookups in String Indexes. In Proceedings of the String Processing and Information Retrieval: 23rd International Symposium, SPIRE 2016, Beppu, Japan, 18–20 October 2016; Inenaga, S., Sadakane, K., Sakai, T., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 61–67.
32. Sadakane, K. Compressed Suffix Trees with Full Functionality. *Theory Comput. Syst.* **2007**, *41*, 589–607. [CrossRef]
33. Brodal, G.S.; Fagerberg, R. Cache-oblivious string dictionaries. In Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm, SODA '06, Miami, Florida, 22–26 January 2006; ACM Press: New York, NY, USA, 2006; pp. 581–590.
34. Brodnik, A.; Jekovec, M. *Sliding Suffix Tree*; Technical Report; University of Ljubljana, Faculty of Computer and Information Science: Ljubljana, Slovenia, 2018.