Journal of Discrete Algorithms 18 (2013) 32–48

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Near real-time suffix tree construction via the fringe marked ancestor problem ☆

Dany Breslauer [a,1], Giuseppe F. Italiano [b,*,2]

[a] Caesarea Rothschild Institute for Interdisciplinary Applications of Computer Science, University of Haifa, Haifa, Israel
[b] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Rome, Italy

ARTICLE INFO

Article history:
Available online 10 August 2012

Keywords:
String algorithms
Suffix trees

ABSTRACT

We contribute a further step towards the plausible *real-time* construction of suffix trees by presenting an on-line algorithm that spends only $O(\log\log n)$ time processing each input symbol and takes $O(n\log\log n)$ time in total, where n is the length of the input text. Our results improve on a previously published algorithm that takes $O(\log n)$ time per symbol and $O(n\log n)$ time in total. The improvements are obtained by adapting Weiner's suffix tree construction algorithm to use a new data structure for the fringe marked ancestor problem, a special case of the nearest marked ancestor problem, which may be of independent interest.

1. Introduction

The suffix tree is a ubiquitous data structure at the heart of numerous text algorithms. Weiner [49] introduced suffix trees and gave a linear-time on-line algorithm for their reverse right-to-left construction. Ukkonen [48] derived a linear-time left-to-right on-line algorithm that is a close relative of an earlier off-line algorithm by McCreight [43]. Although Weiner's, McCreight's and Ukkonen's algorithms all take $O(n)$ time, where n is the length of the input text, the analysis of all three algorithms is amortized and the algorithms may spend up to $O(n)$ time processing some input symbols, while traversing long paths in the suffix tree to find the insertion points of new suffixes. Suffix arrays, that were introduced by Manber and Myers [42], provide similar theoretical benefits to suffix trees and are much more efficient in practice thanks to their use of a compact array representation, but loose some of their advantages in the on-line setting. Throughout this paper, unless specified otherwise, we assume that the input alphabet has constant size.

Amir et al. [5] were the first to achieve some progress towards constructing the suffix tree in *real-time*, namely, attempting to limit the time spent while processing each individual input symbol in the worst case (see also [4,38]). Their algorithm uses balanced search trees to maintain a *balanced indexing structure* that quickly finds the *suffix tree insertion points* of each suffix of an input text that is extended from right to left over an arbitrarily large but ordered alphabet, spending $O(\log n)$ time processing each symbol and $O(n\log n)$ time in total. They also note that similar results could be derived by using existing more complicated dynamic data structures for searching multidimensional keys [28,31]. The *suffix tree insertion point* of any given suffix is its longest prefix that has already appeared earlier in the text, also called sometimes *longest repeated*

☆ A preliminary version of this paper was presented at SPIRE 2011 – the 18th International Symposium on String Processing and Information Retrieval (Breslauer and Italiano, 2011 [12]).

* Corresponding author.
E-mail address: pino.italiano@gmail.com (G.F. Italiano).

[1] Partially supported by the European Research Council (ERC) Project SFEROT and by the Israeli Science Foundation Grants 686/07, 347/09 and 864/11.
[2] Partially supported by the 7th Framework Programme of the EU (Network of Excellence "EuroNF: Anticipating the Network of the Future – From Theory to Design") and by MIUR, the Italian Ministry of Education, University and Research, under Project AlgoDEEP.

1570-8667/$ – see front matter
http://dx.doi.org/10.1016/j.jda.2012.07.003

prefix, *longest repeated suffix* or *longest previous factor* in the context of off-line computation on suffix arrays; it has numerous applications in text algorithms and in data compression [6,11,16,17,20–22,27].

In related work, Kosaraju [39] and Amir and Nor [2] solve the real-time pattern matching and indexing problems by building a suffix tree in *quasi real-time* using the "candelabra" approach (the term "candelabra" was coined by Amir and Nor). Although Slisenko [46] claimed to have solved these problems and even to classify all periodicities in a string in real-time, a convincing solution was considered to be an open problem [2,29,39] for many years. *Quasi real-time* means that sufficient parts of the suffix tree are built "just in time" before needed by an algorithm that is traversing the suffix tree *starting from its root*. The size of the candelabra can be quite large, however, and neither algorithm guarantees any meaningful upper bound on the time required to find the insertion points of a specific text suffix, but only that parts of the suffix tree will be completed before they are reached. Thus, finding the suffix tree insertion points is *at least as hard, if not harder*, than the real-time pattern matching and indexing problems that offer further amortization opportunities. Consequently, it seems natural to ask the following question: *is it possible to compute the suffix tree insertion points in real-time*?

In this paper we present an on-line suffix tree construction algorithm that spends only $O(\log\log n)$ time processing each input symbol and takes $O(n\log\log n)$ time in total, thus contributing a further step towards the plausible real-time construction of suffix trees. To achieve these superior worst-case time bounds, we design a new dynamic data structure for the *fringe marked ancestor* problem, a special case of the *nearest marked ancestor* problem on trees [1,3,50], where the marked nodes form a contiguous subtree at the root, i.e., if a node is marked, then all its ancestors on the path to the tree's root are also marked. Our data structure, which may be of independent interest, supports updates and queries in worst-case $O(\log\log n)$ time. We then use our dynamic fringe marked ancestor data structure in an adaptation of Weiner's [49] right-to-left on-line suffix tree construction algorithm, shortcutting the occasional long path traversals using fringe-ancestor queries and de-amortizing some internal updates over time. Perhaps, it is worthwhile to emphasize that the suffix tree for a right-to-left extended text appears to be more amenable to construction in real time than the suffix tree for left-to-right extended text, since when the text is extended from right to left, upon each input symbol only one suffix leaf and possibly one internal branching node need to be added to the suffix tree.

Suffix tree construction algorithms [18,32,40,43,48,49] use *suffix links*, which are internal features of these algorithms that are not part of the suffix tree's definition itself, but are, nonetheless, useful in many applications. Throughout the paper, we will refer to the suffix tree itself as the *visible* component of the algorithms and to the suffix links as the *invisible* internal components of the algorithms. Our suffix links, which are related to the edges of the directed acyclic word graph (DAWG) [8,9,14,16,18,40] and were also used by Kosaraju [39] and by Amir and Nor [2], turn out to be extremely helpful in navigating the suffix tree; indeed, these suffix links allow us to use the fringe marked ancestor data structure instead of the nearest marked ancestor data structure that was used by Breslauer [10] to build the suffix tree of a tree in a related approach that spends $O(\frac{\log n}{\log\log n})$ worst-case time per text symbol [1]. However, these invisible suffix links need to be individually created and repeatedly updated, tasks that are delayed and later de-amortized over time while *the complete visible suffix tree is available immediately* at all times. The invisible internal suffix links could be made externally available "just in time", if required, to any algorithm that traverses the suffix tree starting from its root, similarly to Kosaraju's [39] "quasi" real-time construction. Our use of the word "quasi" in this context has a double meaning here: not only parts of the construction are de-amortized and completed later over time, but the time spent processing each input symbol is up to $O(\log\log n)$ rather than constant time.

The on-line construction of a suffix tree from left to right is a much more complicated undertaking that requires a suitably defined "just-in-time" de-amortization. In this case, indeed, all the suffixes of the text are extended simultaneously, including those so-called "implicit node" suffixes that do not yet branch out of the suffix tree, and therefore, the visible suffix tree may undergo multiple structural changes at the same time. Such visible structural changes may accumulate over multiple suffix extensions and eventually need to be carried out in large batches that insert many new internal nodes and leaves into the suffix tree at once [13,18,32,40,48]. To obtain a quasi real-time left-to-right suffix tree construction, we shortcut certain steps in Ukkonen's on-line algorithm [48] by exploiting our near real-time adaptation of Weiner's right-to-left algorithm, applied this time to the reverse left-to-right text which is perceived as being extended from right to left. Our adaptation of Ukkonen's left-to-right algorithm de-amortizes the visible new suffix tree node and leaf insertions over time, but with suitably relaxed suffix tree representation, applies the invisible suffix link adjustments immediately in real-time (i.e. at most one new suffix link creation or one existing suffix link update) and adds only constant extra time to the processing of each input symbol relative to our particular adaptation of Weiner's right-to-left algorithm.

The paper is organized as follows. We define the dynamic fringe marked ancestor problem and present our new data structure for its solution in Section 2. We then overview suffix trees and suffix links in Section 3 and adapt Weiner's right-to-left suffix tree algorithm to use the fringe marked ancestor data structure in Section 4. We next show in Section 5 how to adapt also Ukkonen's left-to-right suffix tree algorithm in order to de-amortize its updates by means of the adapted Wiener's right-to-left algorithm. Finally, we conclude with some remarks and open problems in Section 6.

2. The fringe marked ancestor problem

The fringe marked ancestor problem is a special case of the nearest marked ancestor problem with the additional restriction that the marked nodes must form a contiguous subtree at the root. Specifically, the fringe marked ancestor problem is

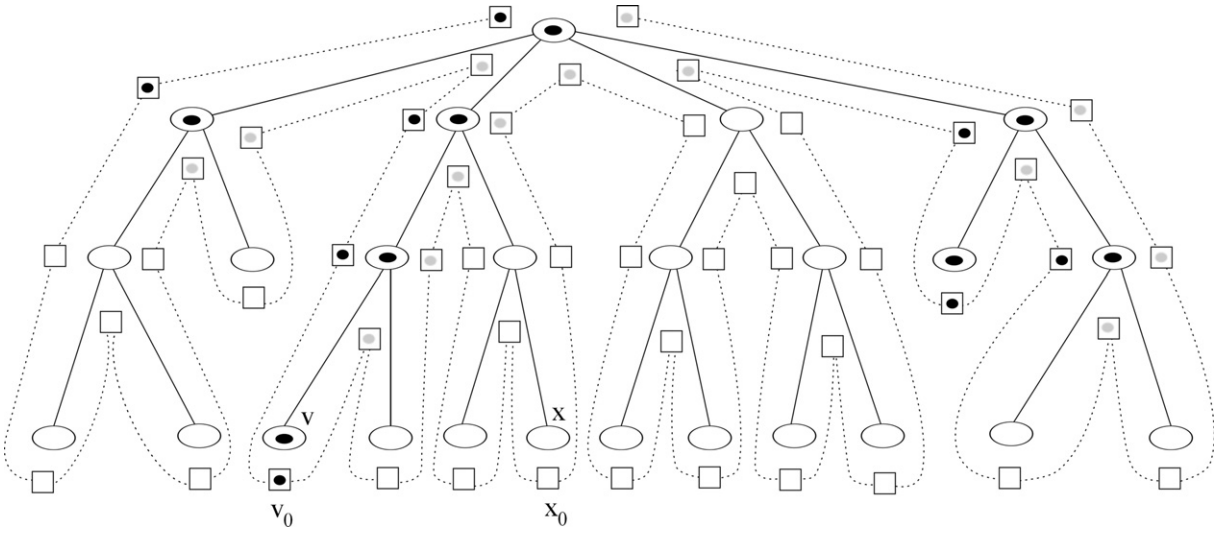

Fig. 1. The Euler tour of a tree with some marked nodes. In Lemma 2.1, only the first Euler tour representative of each marked node is marked solid black: in this example, $v_0 = find(x_0)$ and $fringe\text{-}ancestor(x) = lca(v, x)$. In Lemma 2.3, the suffix tree has constant bounded degree and all the Euler tour representative of each node are *marked (both solid black and light grey)* and $fringe\text{-}ancestor(x) = find(x_0)$, eliminating the need for the additional dynamic *lca* data structure.

concerned with maintaining a rooted tree whose nodes are either marked or unmarked, under an intermixed sequence of the following operations:

make-tree(x) returns a tree consisting of only an unmarked node x;
insert(u, x, v) inserts a new node x in the middle of an edge (u, v), where x becomes a child of u, a parent of v and adopts v's marked status;
insert-leaf(u, x) inserts a new unmarked node x as a child of u;
delete(u, x, v) deletes node x with an only child v and replaces it with an edge (u, v);
delete-leaf(x) deletes leaf x from the tree;
mark(x) marks node x, if x is the root or x's parent is already marked;
unmark(x) unmarks node x, if x has no marked children;
fringe-ancestor(x) returns the nearest marked ancestor of x (which is x itself if it is marked).

These operations maintain the invariant that the marked nodes constitute a contiguous subtree at the root, i.e., if a node is marked then all its ancestors on the path to the tree's root are also marked, a restriction that enables faster algorithms, circumventing an $\Omega(\frac{\log n}{\log\log n})$ worst-case time lower bound for the unrestricted nearest marked ancestor problem [1].

Let T be the given tree. We maintain an Euler tour $ET(T)$ of T, as follows. $ET(T)$ is a path that starts and ends at the tree root, and traverses each edge exactly twice, once from the parent to the child and once from the child to the parent, according to a depth-first traversal of the tree. Note that for each edge (x, y) in T there are has exactly two corresponding edges in $ET(T)$, and for each node in T of degree k there are exactly k corresponding nodes in $ET(T)$ (except for the root that has $k + 1$). A tree T and its Euler tour $ET(T)$ are illustrated in Fig. 1.

We store the Euler tour $ET(T)$ in a linear list, such that each tree node in T holds pointers to all its corresponding elements in the linear list, and each edge in T store pointers to its two corresponding edges in $ET(T)$. We maintain this linear list as dynamic union-split-find data structure, which is capable of performing the following operations:

add(x, y) inserts a new element y after element x in $ET(T)$;
remove(x) deletes element x from $ET(T)$;
split(x) marks element x if x was not marked already;
union(x) unmarks element x if x was previously marked;
find(x) returns the previous marked element (closest to x) in the linear list.

Using the data structure by Dietz and Raman [23], each of these five operations can be implemented in $O(\log\log n)$ time in the worst case. In addition, we also maintain in tandem a copy of the tree T in the *least common ancestor* (*lca*) data structure of Cole and Hariharan [15], which supports the following operations in worst-case constant time:

insert(u, x, v) inserts a new node x in the middle of an edge (u, v);
insert-leaf(u, x) inserts a new node x as a child of u;

delete(u, x, v) deletes node x with an only child v and replaces it with an edge (u, v);
delete-leaf(x) deletes leaf x from the tree;
lca(x, y) returns the *lca* of x and y.

We now show how these data structures are used to implement the operations in the fringe marked ancestor problem. A *make-tree* operation creates a tree consisting of a single node and no edge. The corresponding Euler tour consists of one single element and is initialized in $O(1)$ worst-case time. Operations *insert-leaf* and *insert* require the insertion of one or two elements into the Euler tour $ET(T)$, respectively, and thus are implemented in $O(\log\log n)$ worst-case time with a constant number of *add* operations in the dynamic union-split-find data structure. Symmetrically, operations *delete* and *delete-leaf* can be carried out in $O(\log\log n)$ worst-case time with a constant number of *remove* operations in the dynamic union-split-find data structure. In both cases, the *lca* data structure is maintained in tandem in $O(1)$ worst-case time for each update. To *mark* a tree node x, we perform a *split* on the *first* element corresponding to x in the Euler tour $ET(T)$ in $O(\log\log n)$ time. Similarly, to *unmark* x it suffices to perform a *union* on the *first* element corresponding to x in $ET(T)$. Note that this use of the union-split-find data structure guarantees that only the first appearance of a node in the list representing $ET(T)$ can be marked (see Fig. 1). Finally, the *fringe-ancestor* query is supported through one *find* query in the union-split-find data structure and one *lca* query in the *lca* data structure, taking $O(\log\log n)$ time in the worst-case, as the following lemma shows.

Lemma 2.1. *Let T be a tree, let x be a node of T and let x_0 be the first element of the Euler tour $ET(T)$ corresponding to node x. Let v_0 be the closest marked element to the left of element x_0 in $ET(T)$ and let v be the tree node corresponding to v_0. Then $lca(v, x)$ is the fringe ancestor of x in T.*

Proof. Let y be the fringe ancestor of x in T, and assume by contradiction that $y \neq lca(v, x)$. Node v_0 is a marked element of $ET(T)$, and thus the corresponding node v in T must be marked. Since the Euler tour $ET(T)$ follows a depth-first visit of tree T, denote by $\mathrm{DFS}(u)$ the depth-first number of node u according to the Euler tour. The fact that v_0 is the closest marked element to the left of x_0 in the Euler tour is equivalent to saying that $\mathrm{DFS}(v) \leqslant \mathrm{DFS}(x)$ and that no marked node u is such that $\mathrm{DFS}(v) < \mathrm{DFS}(u) \leqslant \mathrm{DFS}(x)$ (i.e., a depth-first traversal in T enters v before entering x and it does not enter any other marked node while going from v to x). Since $lca(v, x)$ is an ancestor of v and v is marked, $lca(v, x)$ must be marked as well; furthermore, since $lca(v, x)$ is an ancestor of x and $lca(v, x) \neq y$, $lca(v, x)$ must be a proper ancestor of y, i.e., y must be in the tree path from $lca(v, x)$ and x. But then, the depth-first traversal of T would enter marked node y while going from v to x, clearly a contradiction. □

Thus, we have proved the following theorem.

Theorem 2.2. *The above data structure solves the fringe marked ancestor problem on an unbounded degree tree in $O(\log\log n)$ worst-case time per operation.*

We observe that the data structure can be further simplified in the application at hand. Indeed, in our suffix trees we can solve the fringe marked ancestor problem even without using the *lca* data structure, since the degree of each suffix tree node is bounded by the constant alphabet size. To do this, we modify the *mark* (*unmark*) operation to perform a *split* (*union*) on *all the elements* corresponding to x in the Euler tour $ET(T)$. This does not infringe the time bounds, as there are only a constant number of elements in $ET(T)$ corresponding to a suffix tree node. Now a *fringe-ancestor* query can be directly supported through a *find* query in the incremental union-split-find data structure, as the following lemma shows (see Fig. 1).

Lemma 2.3. *Let T be a bounded degree tree, let x be a node of T and let x_0 be the first element of the Euler tour $ET(T)$ corresponding to node x. Then the closest marked element to the left of element x_0 in $ET(T)$ must correspond to the fringe ancestor of x in T.*

Proof. Let v_i be the closest marked element to the left of x_0 in $ET(T)$, and let v be its corresponding node in the tree T. Note that, since v_i is a marked element of $ET(T)$, then v must be a marked node of T. Since the Euler tour $ET(T)$ follows a depth-first visit of tree T, and all the elements corresponding to a marked node in T are marked in the Euler tour $ET(T)$, this is equivalent to saying that a depth-first traversal in T encounters v before entering x and it does not encounter any marked node (except for v) while going from v to x. Let $lca(v, x)$ be the least common ancestor of v and x in T: note that a depth-first traversal of T must encounter $lca(v, x)$ while going from v to x. Since v is marked, and $lca(v, x)$ is an ancestor of v, then $lca(v, x)$ must be marked as well. Since v is last marked node encountered before x, this is possible only if $v = lca(v, x)$ is an ancestor of x: since there is no marked node from v to x (except for v), this implies that v must be the fringe ancestor of x. □

3. Suffix trees and suffix links

We assume that the reader has some textbook familiarity with suffix trees [18,32,40] and with the three classical suffix tree construction algorithms by Weiner [49], McCreight [43] and Ukkonen [48]; Giegerich and Kurtz [30] offer an interesting comparative analysis of these three algorithms. Given a text $w = w_1 \cdots w_n$ over the alphabet Σ, the *suffix tree* of w, denoted as $\mathcal{T}_w$, is a rooted tree whose edges and nodes are labeled with substrings of w. The suffix tree of w satisfies the following properties:

1. Edges leaving any given node are labeled with non-empty strings v that start with different alphabet symbols (v is a substring of w);
2. Each node is labeled with a string v formed by the concatenation of the edge labels on the path from the root to that node (v is a substring of w);
3. Each branching internal (non-leaf) node has at least two descendants, with the only possible exception of the root in the degenerate case when a string is empty or it is formed by repetitions of a single alphabet symbol;
4. For each substring v of w, there exists a node labeled u, such that v is a prefix of u.

The *locus* of a substring v of w is the unique location in $\mathcal{T}_w$ that is labeled with v. Whenever possible, it is convenient to append at the end of the text w a special unique terminating alphabet symbol \$ which does not appear anywhere within w. This guarantees that $\mathcal{T}_{w\$}$ has exactly $|w| + 1$ leaves that are labeled with all the distinct non-empty suffixes of $w\$$. The number of branching internal nodes is no larger than $|w|$. However, in on-line algorithms that construct the suffix trees for a left-to-right streaming text, it would be too costly to append and then remove the special terminating alphabet symbol \$ at each step. Therefore, an on-line algorithm for left-to-right extended text must be able to handle also suffix tree nodes representing text suffixes which may not be branching out of the tree. The locus of such text suffixes may be in the middle of a suffix tree edge or may coincide with an internal branching suffix tree node. In the former case, the corresponding position (in the middle of a suffix tree edge) is referred to as an *implicit node*, since this node is not represented explicitly in the suffix tree. In the latter case, the suffix tree node is referred to as an *explicit node*. A locus in the middle of a suffix tree edge is defined by the node v immediately below the edge and by its distance in number of symbols to v. We refer to thls distance as the *slack distance* of the locus.

M-links. Weiner, McCreight and Ukkonen all augment the suffix tree $\mathcal{T}$ with shortcuts called *suffix links* that are used to efficiently traverse the suffix tree. For a suffix tree node $v = v_1 \cdots v_k$, the M-link $\mathcal{M}(v) = u$, McCreight suffix link (also used by Ukkonen), is defined to be a pointer to the suffix tree node $u = v_2 \cdots v_k$ that is obtained by chopping off the first symbol $a = v_1$ of $v = au$. These M-links are well defined for any branching suffix tree node (except for the root): indeed, if the node $v = au$ branches with edges that begin with alphabet symbols b and c, $b \neq c$, then the suffix tree also contains the substrings ub and uc and there must be a node labeled u, branching on the symbols b and c. The situation with suffix tree leaves is a little more complicated. Leaves clearly represent text suffixes, since only suffixes of the text end abruptly on their right side. If the text is terminated with a special unique alphabet symbol \$, then all suffixes of the text are leaves and, therefore, if $v = au$ is a leaf then its suffix u must also be a leaf. However, if the text is not terminated with the special symbol \$, then the suffix u could either coincide with an existing branching node or be an implicit node in the middle of an edge. In the latter case, the M-link $\mathcal{M}(v)$ of a leaf v might be an implicit node, and therefore undefined. Nonetheless, each leaf $v = au$ represents a suffix and if the M-link $\mathcal{M}(v) = u$ is defined then it is also a suffix. Since each non-root node has one M-link and M-links cannot introduce cycles, M-links define a tree rooted at the suffix tree root, which becomes a trie when labeled with the chopped first symbols. This trie, which we call the *suffix link trie*, is a subtree of the suffix trie for the reverse text. Fig. 2 illustrates a suffix tree (Fig. 2(a)) and its M-links (Fig. 2(b)) in the case where all leaves have their M-links defined. If some suffix tree leaves do not have their M-links defined, we can still define the suffix link trie by considering the suffix links for all suffix tree nodes and implicit nodes, as shown in Fig. 3. The path in the suffix link trie from the longest leaf representing the full text to the root goes through all the suffixes of the text, which are the only substrings that get extended while the input text is processed from left to right. We call this path the *suffix chain* (see Fig. 3(b)). The following simple observation was formalized in [13]:

Lemma 3.1. *The suffix chain can always be partitioned into the following three consecutive segments*:

(1) *leaves*;
(2) *implicit nodes* (*first within external leaf edges and next within internal edges*); *and*
(3) *explicit nodes.*

W-links. Similarly to M-links, the W-link $\mathcal{W}_a(u) = v$, Weiner's suffix link, of a suffix tree node u and symbol $a \in \Sigma$, is defined to be a pointer to the suffix tree locus labeled $v = au$, obtained by appending the symbol a before u. The W-link is only defined for those symbols $a \in \Sigma$, such that $v = au$ is a substring of the text w, and undefined otherwise. If $v = au$ is a suffix tree node, then the W-link $\mathcal{W}_a(u) = v$ is called a *hard W-link* and it is just the opposite pointer of the M-link $\mathcal{M}(v) = u$. However au may also be the locus in the middle of the edge ending at $vx = aux$, $x \neq \epsilon$, rather than a suffix tree

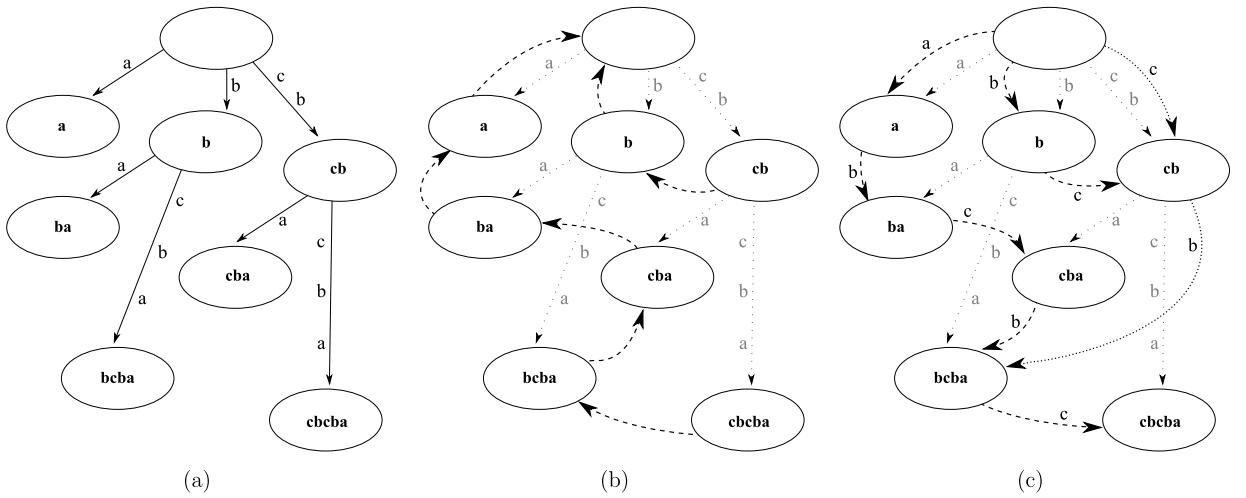

Fig. 2. (a) The suffix tree $\mathcal{T}_w$ of the text $w = cbcba$. (b) The M-links of $\mathcal{T}_w$ that constitute the suffix link trie. (c) The hard and soft W-links of $\mathcal{T}_w$; hard W-links (opposite of M-links) are shown dashed and soft W-links are shown dotted.

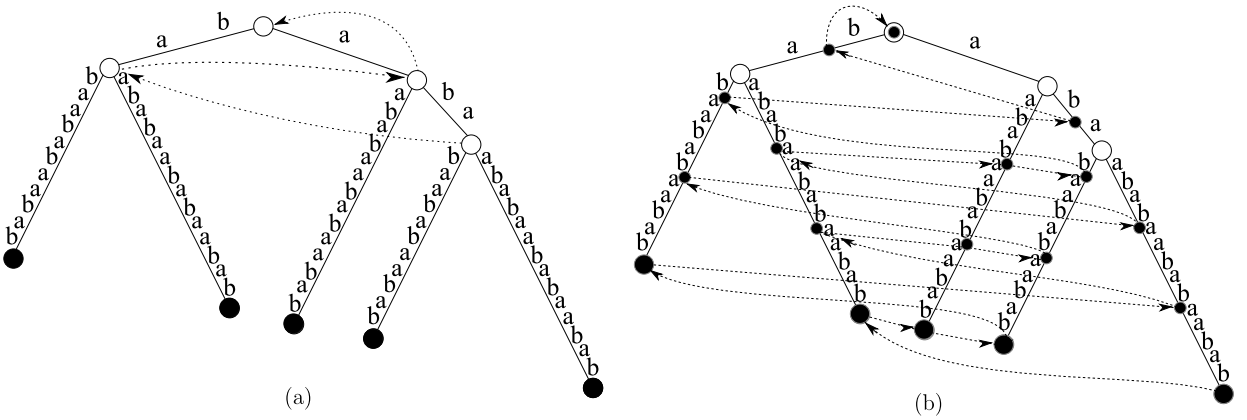

Fig. 3. (a) The suffix tree for the text $w = abaababaababaabab$ with the M-links defined for the internal branching nodes. (b) The suffix tree with implicit nodes and the suffix chain. The suffix link trie consists of the suffix chain in figure (b) plus the M-links shown in (a).

node, in which case $\mathcal{W}_a(u)$ is a *soft W-link* that is defined as a pointer to the shortest extension $vx = aux$ that is a branching suffix tree node or a leaf. Note that the definition of W-links may also be extended to implicit nodes in the middle of an edge. Thus, if $\mathcal{W}_a(u)$ is defined, then au is always prefix of the node $\mathcal{W}_a(u)$. Observe that when new nodes are inserted into the suffix tree, if a new node auy is created between au and $\mathcal{W}_a(u) = aux$, then the soft W-link $\mathcal{W}_a(u)$ must be updated to point to auy; hard W-links and their opposite M-links are not affected by the insertion of new nodes.

Let v be a suffix tree node: throughout this paper, we denote by $d(v)$ the depth of v in the suffix tree (note that implicit nodes do not contribute to the depth). The following simple properties of W-links are widely used by suffix tree construction algorithms.

Lemma 3.2. *For each suffix tree locus $v = au$, the suffix tree depths satisfy $d(v) \leqslant d(u) + 1$.*

Proof. Each node on the path from $v = au$ to the root, except for the root, has an M-link pointing to a different corresponding node on the path from $u = \mathcal{M}(v)$ to the root (see Fig. 4). □

The next lemma shows that the W-links are contiguous in the suffix tree, which allows us to use the fringe marked ancestor data structure from Section 2.

Lemma 3.3. *If the W-link $\mathcal{W}_a(u)$ is defined for a node $u \in \mathcal{T}_w$ and symbol $a \in \Sigma$, then all ancestors $u' \in \mathcal{T}_w$ of u must also have their W-link $\mathcal{W}_a(u')$ defined.*

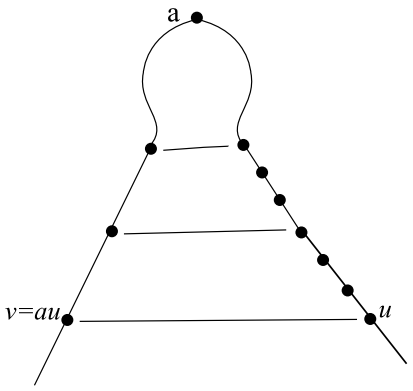

Fig. 4. Loci depths' satisfy $d(au) \leqslant d(u) + 1$, while their lengths satisfy $|au| = |u| + 1$.

Proof. If $u \in \mathcal{T}_w$ has a W-link $\mathcal{W}_a(u)$ defined, then au is a substring of w and any prefix au' of au is also a substring of w. □

Remarkably, the total number of W-links, which are the same as the edges in the directed acyclic word graph (DAWG), is not too large [9,16,18,40], independent of the alphabet size.

Lemma 3.4. *(See Blumer et al. [9] and Crochemore [16].) There are at most* $3 \cdot |w|$ *defined W-links.*

R-links. Given the text $w = w_1 \cdots w_n$, we denote its *reverse* by $\widetilde{w} = w_n \cdots w_1$. Some of the algorithms presented in this paper will also maintain $\mathcal{T}_{\widetilde{w}}$, the suffix tree of $\widetilde{w}$. We will need to identify each node $v \in \mathcal{T}_w$ with its reverse node $\widetilde{v} \in \mathcal{T}_{\widetilde{w}}$, if such node exists. If both $v \in \mathcal{T}_w$ and $\widetilde{v} \in \mathcal{T}_{\widetilde{w}}$ are internal branching nodes, then we define the R-link to be a pointer from the node v in the suffix tree $\mathcal{T}_w$ to the node $\widetilde{v}$ in the reverse suffix tree $\mathcal{T}_{\widetilde{w}}$. Such R-link pointers can be maintained in both directions.

We say that an internal branching node v of a suffix tree is a *core node* if v has at least two distinct W-links, i.e., $\mathcal{W}_a(v)$ and $\mathcal{W}_b(v)$, with alphabet symbols $a \neq b$. We refer to the set of core nodes as *the core* of the suffix tree. Note that the core nodes are exactly those nodes that have their R-links defined. Clearly, by Lemma 3.3, any ancestor u of a core node v is also a core node. Consequently, the core of a suffix tree forms a contiguous subtree at the tree root, which allows us to use the fringe marked ancestor data structure from Section 2 to find the nearest ancestor core node for any given node in the suffix tree.

4. Right-to-left construction

In this section we show how to adapt Weiner's right-to-left on-line suffix tree construction algorithm [18,32,40,49] into an algorithm that takes $O(\log\log n)$ worst-case time processing each input symbol and spends $O(n \log\log n)$ time in total. Since Weiner's algorithm constructs the suffix tree for a text that is extended from right to left, in each step the existing set of suffixes does not change and only one new suffix, i.e., the longest suffix equal to the whole text, is added to the suffix tree. In this case, we can also conveniently assume that the text is terminated with the special unique symbol \$, and therefore, that all the text's suffixes are represented by leaves.

4.1. Weiner's algorithm

We now recall the detailed individual steps of Weiner's algorithm. Suppose that the text $w\$$ is extended from right to left with the next alphabet symbol $a \in \Sigma$. Then, the suffix tree $\mathcal{T}_{w\$}$ has to be updated to become $\mathcal{T}_{aw\$}$ by inserting the new leaf $aw\$$ hanging off some internal branching node v: this internal node v might already exist in $\mathcal{T}_{w\$}$ or it might need to be inserted as well. Observe that the insertion point v is the longest prefix of the extended text $aw\$$ that is equal to some substring of $w\$$, also called sometimes the *longest repeated prefix*. Unless v is the suffix tree root, which may only happen if $a \in \Sigma$ is a new alphabet symbol never seen before (in which case $aw\$$ will be hanging off the root), the suffix tree $\mathcal{T}_{aw\$}$ must also contain the branching node $u = \mathcal{M}(v)$. Furthermore, this node u was already in the suffix tree $\mathcal{T}_{w\$}$ before $w\$$ was extended to $aw\$$. Observe that u is an ancestor and a prefix of $w\$$. Moreover, $u \in \mathcal{T}_{w\$}$ is the deepest ancestor and longest prefix of $w\$$, that has the W-link $v'' = \mathcal{W}_a(u)$ defined. The possibly new node $v = au \in \mathcal{T}_{aw\$}$, is an ancestor of v''.

To extend $\mathcal{T}_{w\$}$ into $\mathcal{T}_{aw\$}$, Weiner's algorithm finds the node u by tracing the path from the leaf $w\$$ towards the root until it encounters the nearest ancestor u of $w\$$ whose W-link $\mathcal{W}_a(u)$ is defined (see Fig. 5). Let $u' = \mathcal{M}(v')$ be the nearest ancestor of $w\$$ having a hard W-link $\mathcal{W}_a(u')$ defined. If $v'' = \mathcal{W}_a(u)$ is a hard W-link, then $u' = u$ and the insertion point $v = v''$ already exists in the suffix tree $\mathcal{T}_{w\$}$. If $v'' = \mathcal{W}_a(u)$ is a soft W-link, then $u' \neq u$ and a new branching node v,

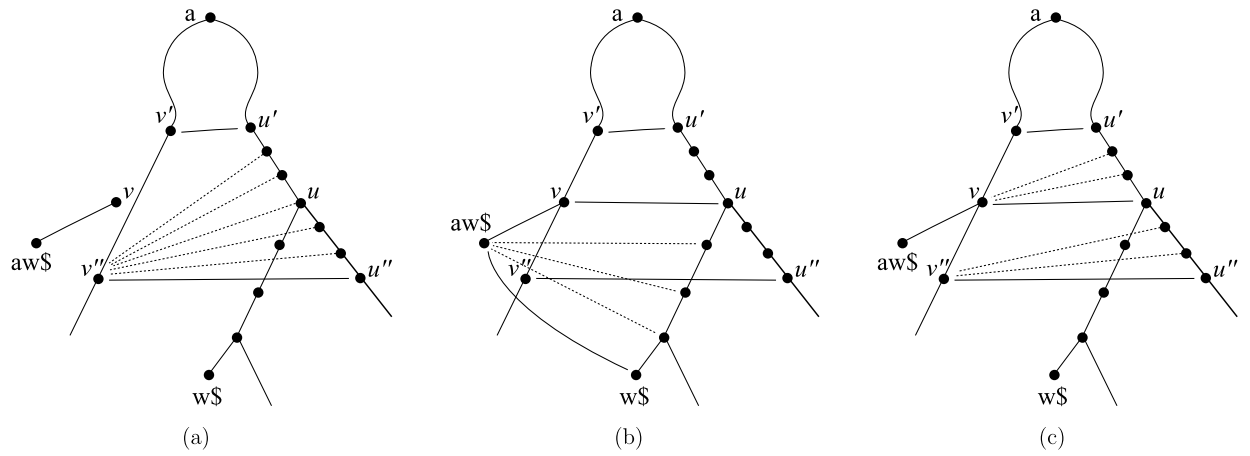

Fig. 5. Extending $\mathcal{T}_{w\$}$ to $\mathcal{T}_{aw\$}$. Throughout this figure, soft W-links are shown dashed and hard W-links are shown solid. (a) W-links before extending. (b) New W-links to $aw\$$ are created from all nodes on the path between $w\$$ up to u (W-link creation tasks). (c) W-links to v'' from all nodes on the path between u up to u' are updated to point to v instead (W-link update tasks). Observe that each one of these adjusted W-links ($d(w\$)$–$d(u)$ created and $d(u)$–$d(u')$ updated) is at a different depth and the insertion depth was reduced by the number of adjusted soft W-links; the new hard W-link $\mathcal{W}_a(w\$) = aw\$$ and updated hard W-link $\mathcal{W}_a(u) = v$ have corresponding nodes on the path to $aw\$$. The new node v also adopts all the outgoing W-links of v'' (not shown).

such that $\mathcal{M}(v) = u$, must be created on the edge between v'' and its parent v' in the suffix tree $\mathcal{T}_{w\$}$ and its W-links are initialized to be precisely the same W-links as its child v''.

Throughout this process, some W-links need to be adjusted along the way, i.e., either created or updated. The number of adjustments involving hard W-links is constant, as we have to create one new hard W-link from the leaf $w\$$ to the new leaf $aw\$$, and change the existing soft W-link $\mathcal{W}_a(u) = v''$ to a hard W-link $\mathcal{W}_a(u) = v$, if a new branching node v had to be created. The adjustments involving soft W-links are more numerous, and we refer to them as *W-link adjustment* tasks. In particular, W-link adjustment tasks consist of *W-link creation* tasks and *W-link update* tasks, defined as follows. W-link creation tasks create new soft W-links from the ancestors of $w\$$ at depths $d(u)+1, \ldots, d(w\$)-1$ to the new leaf $aw\$$ (see Fig. 5(b)). If a new node v had to be created, the W-link update tasks move the existing soft W-links of the ancestors of $w\$$ at depths $d(u')+1, \ldots, d(u)-1$ from the node v'' to the newly created node v (see Fig. 5(c)).

Observe that each one of these new and updated W-links is at a different suffix tree depth, and therefore, the W-link adjustments can be implicitly identified efficiently, in constant time, which is going to be crucial for the time analysis. In addition, if a new branching node v was created, then v also adopts all the W-links of v'' (all adopted W-links become soft, including those W-links of v'' that were hard).

The amortized analysis of Weiner's algorithm is based on the fact that the suffix tree depth of $aw\$$ is by at most one larger than the suffix tree depth of $w\$$. This is the case because each ancestor of $aw\$$, except for the root, has an M-link that points to a different ancestor of $w\$$, by Lemma 3.2. Thus, the number of steps traversing the path from $w\$$ towards the root to find u is bounded by the depth reduction and the overall total depth increases throughout the algorithm are bounded by the number of text symbols.

4.2. Quasi real-time right-to-left construction

Our quasi real-time adaptation of Weiner's algorithm maintains a W-link adjustments *de-amortization stack* with all the delayed soft and hard W-link adjustment tasks (both W-link creations and W-link updates). We observe that the soft W-link update tasks would be at depths $d(u')+1, \ldots, d(u)-1$, while the soft W-link creation tasks would be at depths $d(u)+1, \ldots, d(w\$)-1$ (see Fig. 5). This has two important consequences. First, we do not need to push explicitly all those tasks onto the de-amortization stack, as we can do this implicitly in constant time. Second, all those tasks will be at different depth levels at the time they are (implicitly) pushed onto the de-amortization stack. With this approach, tasks will be then executed from the de-amortization stack according to their increasing depth, i.e., from shallow to deep, in reverse order of Weiner's original algorithm. Consider now $aw\$$, the next suffix to be inserted. Clearly, $d(aw\$) = d(v)+1$ and by Lemma 3.2, $d(v) \leqslant d(u)+1$ and $d(v) \leqslant d(u')+2$, which implies that $d(aw\$) \leqslant d(u')+3$. Thus, when the next suffix $aw\$$ is inserted, only tasks whose tree depth is at most $\leqslant d(aw\$)-1 \leqslant d(u')+2$ may be pushed onto the de-amortization stack. To keep the invariant that all the delayed tasks are at different increasing depth levels at the time they are put on the de-amortization stack, each time a new suffix is inserted it would suffice to execute a constant number (i.e., at least two) of previously delayed W-link adjustment tasks from the top of the de-amortization stack. Not only this will make room for the new tasks, but it will also allow us to catch up eventually with all the delayed tasks, thanks to the amortized analysis of Weiner's algorithm.

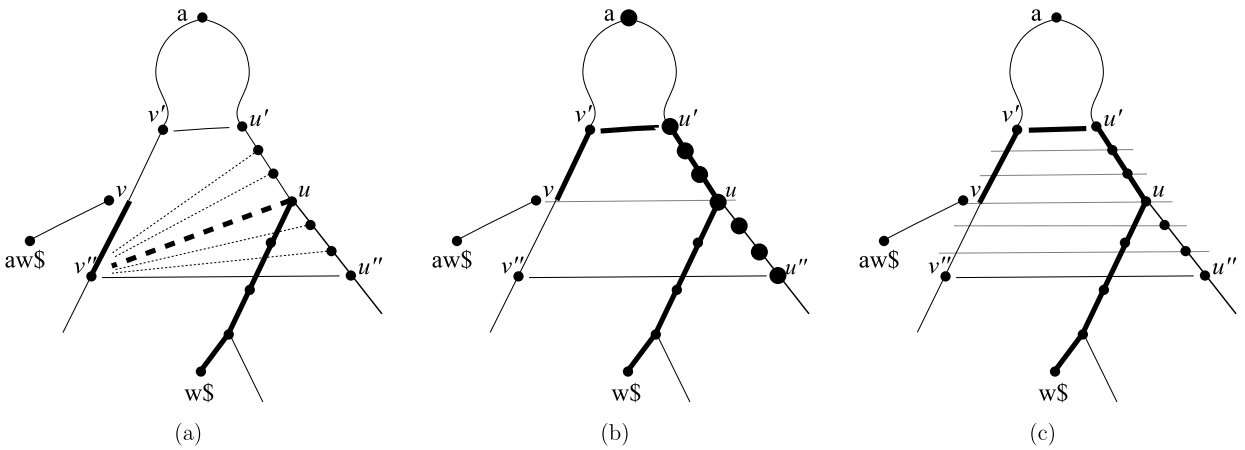

Fig. 6. Finding the insertion point v. Throughout this figure, soft W-links are shown dashed and hard W-links are shown solid. (a) Soft and hard W-links: follow the first ancestor u with W-link $v'' = \mathcal{W}_a(u)$ defined. (b) Hard W-links and indicators (nodes with indicators shown as thicker dots): follow the first ancestor u' with hard W-link $v' = \mathcal{W}_a(u')$ defined, using the offset of the first ancestor u with set indicator. (c) Only hard W-links: follow the first ancestor u' with hard W-link $v' = \mathcal{W}_a(u')$ defined and compute the offset of v from v' in a second scan.

Since the current suffix tree insertion depth increases at most by one with each text symbol, by executing a constant number of tasks (at least two) from the de-amortization stack in each step, we can be sure that all tasks at the current depth were already accomplished. Note that by inserting new branching nodes the depth of existing tasks on the de-amortization task can increase, but never decrease: thus, the depths used for these stack depth properties are the depth of the tasks at the time the tasks are put on the stack.

Theorem 4.1. *We can adapt Weiner's right-to-left on-line suffix tree algorithm over constant size alphabets to take up to* $O(\log\log n)$ *time processing each input symbol and spend* $O(n \log\log n)$ *time in total.*

Proof. For each alphabet symbol $a \in \Sigma$, we maintain in tandem a separate fringe marked ancestor data structure mirroring the suffix tree, where a node u is marked if and only if the W-link $\mathcal{W}_a(u)$ is defined. By Lemma 3.3, the W-links are contiguous and the fringe marked ancestor data structure may be used. Over constant size alphabets, all these data structures are updated in additional $O(\log\log n)$ worst-case time per each new suffix node and leaf and each new W-link (W-link updates do not affect the fringe marked ancestor data structure). Thus, we created an alternative mechanism to find the suffix tree insertion point instead of tracing the path to the root; we use symbol a's fringe marked ancestor data structure to directly find the nearest ancestor u of $w\$$ that is marked, or in other words, has W-link $\mathcal{W}_a(u)$ defined, in $O(\log\log n)$ worst-case time.

Specifically, the algorithm jumps from the leaf $w\$$ to its first ancestor u with defined W-link $v'' = \mathcal{W}_a(u)$, then to the parent v' of v'' and then following the M-link to $u' = \mathcal{M}(v')$. See Fig. 6(a). The new branching node v in created between v' and v'', such that the lengths $|v| - |v'| = |u| - |u'|$. If the W-link $v'' = \mathcal{W}_a(u)$ was a hard W-link, then the branching node $v = v' = v''$ already exists and no new node v needs to be created simplifying the process; in this case also $u = u' = v''$ and there are no W-link update tasks.

While this allows us to insert the new leaf and branching node quickly (with their associated M-links, opposite hard W-links and the adopted W-links), we also need to create the new soft W-links on the path between $w\$$ and u and update the existing soft W-links on the path between u and u'. We maintain a de-amortization stack for these soft W-link adjustment tasks, and execute these tasks later, adjusting the W-links from shallow to deep. The affected nodes at depths $d(u') + 1, \ldots, d(u) - 1$ require existing soft W-link adjustments and at depths $d(u) + 1, \ldots, d(w\$) - 1$ get new soft W-links. Since the depth of the suffix tree insertion point increases at most by one in each step, if we update at least one or more W-links from the de-amortization stack at each step, we guarantee that the depths of the remaining pending W-link adjustment tasks on the stack are strictly increasing. □

Note that the hard W-links and their opposite M-links may be created immediately for each new branching node and new leaf as they are inserted, but the leaf's hard W-link creation in the fringe marked ancestor data structure must take place according to the de-amortized stack depth order to preserve the fringe property. However, an algorithm may also wish to gain access to the internal invisible soft W-links through "just in time" de-amortization, as shown in the following corollary.

Corollary 4.2. *An algorithm may access the internal invisible W-links that will be available "just in time", if it executes the delayed W-link adjustment tasks from the de-amortization stack, provided that such algorithm traverses the suffix tree starting from the root.*

Proof. Suppose that an algorithm is traversing the suffix tree starting from its root. In each step, the algorithm moves from one suffix tree locus to another locus that is at most one tree level deeper. Such moves can be done by following a suffix tree edge one symbol or even the full edge length or by following W-links. The W-links on the de-amortization stack that need to be adjusted are each at different increasing depth level. If the algorithm sweeps the de-amortization stack and executes the delayed updates ahead of using the W-links, it is guaranteed that the W-links at the current level that is visited are ready before used. □

Remark. The common textbook description of Weiner's [18,32,40] algorithm, that can probably be traced back to Seiferas' [44] simplified presentation, uses Boolean indicator variables instead of soft W-links (Fig. 6(a)–(b); the fringe marked ancestor data structure is actually used to maintain these indicators). This is very appealing since neither hard W-links nor Boolean indicators need to be updated once set, unlike soft W-links that require constant maintenance. We wish to point out a rather trivial observation, that the Boolean indicators are not required at all and it suffices to maintain only the hard W-links and use a second quick scan to locate the suffix tree insertion point v, borrowing a technique from McCreight's [43] and Ukkonen's [48] algorithms. See Fig. 6(c), where the edge between v' and its child v'' corresponds to the path between $u' = \mathcal{M}(v')$ and $u'' = \mathcal{M}(v'')$; $u = \mathcal{M}(v) = lca(w\$, u'')$ and the edge labels on the path between u' and u are equal to the corresponding parts of the edge label between v' and v. Thus, u is the most shallow node on this path between u' and u'', where the first symbol on the edge at u is not equal to its corresponding symbol on the edge between v' and v''. The space requirements of maintaining the suffix tree edges as pointers from nodes to their parents and the hard W-links are very similar to those of maintaining the M-links (pointers towards the root of the suffix link trie) and the branching suffix tree edges in McCreight's and Ukkonen's algorithms.

5. Left-to-right construction

Constructing the suffix tree from left to right in near real time is a much more complicated undertaking that requires more sophisticated de-amortization techniques. Indeed, when the text is extended from left to right, all the suffixes are simultaneously extended and the suffix tree may undergo multiple structural changes at the same time: such structural changes may occur in large batches which may insert many new visible internal nodes and leaves at once [13,18,32,40, 43,48]. Observe that unlike Weiner's [49] right-to-left algorithm, now we cannot conveniently append to the text w the special suffix terminating symbol \$ that guarantees that all suffixes are represented by leaves, but we can still append this terminating symbol to the reverse text $\widetilde{w}\$$ (i.e., at the beginning of the text $\$w$).

5.1. Ukkonen's algorithm

Differently from right-to-left extensions, when the input text is streamed on-line from left to right all the text suffixes must be simultaneously extended in the suffix tree. We have shown in Section 3 that all these suffixes belong to the suffix chain (i.e., Fig. 3(b)), which by Lemma 3.1, is partitioned into the following consecutive segments: (1) leaves; (2) implicit nodes (external, internal); (3) explicit nodes. As the suffix tree is being constructed, the loci of the implicit nodes on the suffix chain that end in the middle of suffix tree edges may be converted into explicit internal branching nodes and new leaves may be inserted hanging off these new and off existing internal branching nodes. Ukkonen's algorithm specifies how this process may be implemented efficiently.

One of the important observations made by Ukkonen is that once some text suffix becomes a leaf, it will remain forever a leaf after all future left-to-right suffix extensions. This implies that every time a new input symbol is added at the right end of the text, i.e., a new suffix is added to the suffix tree, the edges leading into every existing leaf must be automatically extended by that symbol. In other words, upon each suffix extension of the text we do not need to explicitly update the leaves, since leaves will always stretch up to the current end of the text. Hence, by labeling the external suffix tree edges leading to leaves "open ended", reaching to the current, continuously growing end of the text, Ukkonen invented an automatic gratuitous extension mechanism for the leaf labels.

To handle the extensions of the remaining text suffixes, Ukkonen's algorithm maintains throughout its execution the *active point*, which is the longest suffix of the text that has not yet branched out of the suffix tree to become a leaf. If the text is extended from $\$w$ to $\$wa$, then the active point is the locus of the longest suffix of the extended text $\$wa$ in $\mathcal{T}_{\$wa}$ that is equal to a substring of $\$w$, also called the *longest repeated suffix*. The updates carried out by Ukkonen's algorithm hinge on active points: only the suffixes above the active point need to be updated, as all the longer suffixes are leaves and thus are gratuitously extended as explained earlier. In particular, Ukkonen's algorithm walks through the remaining suffixes along the suffix chain, starting from the active point and progressing towards the suffix tree root (i.e., the empty suffix), testing each node for update eligibility.

Consider the text $\$w$ that is extended to $\$wa$. When the algorithm encounters along the suffix chain the first node that can be extended with symbol a within the current suffix tree $\mathcal{T}_{\$w}$, i.e., the first node for which there is already an outgoing edge whose label starts with the symbol a, then such a node will be implicitly updated advancing within the suffix tree, which involves no structural update at all in the visible suffix tree. This first node along the suffix chain is called the *end point*, and the algorithm may stop the suffix tree updates here: indeed, every shorter suffix than the end point may also be extended with a within the current suffix tree $\mathcal{T}_{\$w}$ (if a particular suffix can be extended with symbol a within the suffix

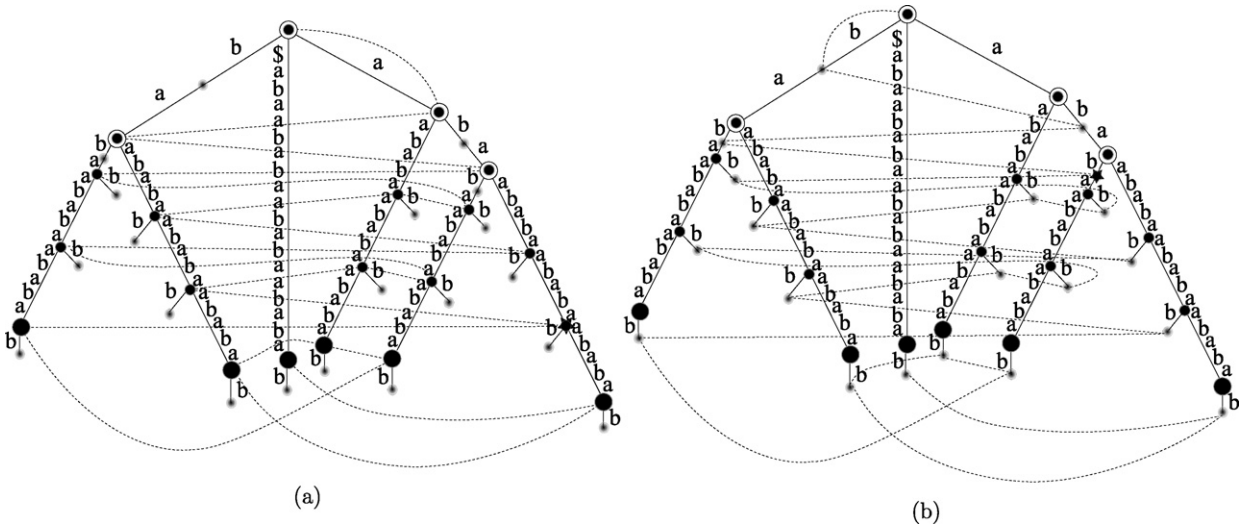

Fig. 7. Ukkonen's algorithm: the suffix chain and the active point (drawn as a star-shaped node). (a) The suffix tree when the text $\$w = \$abaababaababaabaaba$ is extended on the right by symbol b. The large and small dark nodes indicate leaves and implicit nodes of $T_{\$w}$, respectively. Grey nodes indicate the new leaves and implicit nodes of the suffix tree $\mathcal{T}_{\$wb}$ after the suffix extension. The suffix chain of $T_{\$w}$ is represented by the dashed edges. The suffix insertion path of $T_{\$w}$ is the subpath of the suffix chain between the active point (at locus *abaababaababa*) and the end point (at locus *aba*); in this example, all the nodes in the suffix insertion path yield a branching node insertion task. (b) The suffix chain of $T_{\$wb}$, after the text $\$w$ is extended on the right by symbol b: the new active point is at locus *abab*.

tree, then also all the shorter suffixes can be extended with a). On the other hand, every suffix longer than the end point (up to and including the active point) will be extended into a branching leaf node in the new suffix tree $\mathcal{T}_{\$wa}$: thus, as the suffix tree gets updated from $\mathcal{T}_{\$w}$ to $\mathcal{T}_{\$wa}$, the new active point can be simply found by advancing from the end point following the symbol a within the suffix tree.

In summary, an iteration of Ukkonen's algorithm works on all suffixes between the active point and up to (and excluding) the end point. We refer to this sub-path of the suffix chain as the *suffix insertion path*. In other words, the suffix insertion path is a sub-path of the suffix chain that contains all the nodes that during the iteration must branch out (if they have not done already so) and get new leaves representing suffixes. The suffix insertion path may be empty when the active point coincides with the end point, i.e., the active point can be extended with the next symbol a. We refer to all the updates carried out in a suffix insertion path as an *insertion batch*. Fig. 7 illustrates the notions of active point, suffix insertion path and the insertion batch performed by the algorithm after processing a new text symbol. More specifically, for each node in the suffix insertion path Ukkonen's algorithm performs the following tasks:

1. If the node was an implicit node in the middle of an edge, it must branch out by creating a branching node by splitting the edge and inserting a leaf hanging off this new branching node. We refer to this as a *branching node insertion task*. Note that the suffix chain nodes of this type are precisely all the new internal branching nodes that are created by the algorithm and the new branching nodes are all implicit suffixes between the *longest implicit suffix* and the *shortest implicit suffix*.
2. If the node was an an existing explicit node, it must branch out by creating a leaf hanging off that existing explicit node. We refer to this as a *leaf insertion task*. Note that the nodes of this type are precisely the existing internal branching nodes on the suffix chain between the *longest explicit suffix* and (excluding) the end point.

We recall here that Ukkonen's algorithm maintains information about suffixes in *canonical form*, i.e., the locus of an implicit node is represented by a pointer to its closest ancestor in the suffix tree and by an offset from the ancestor within the corresponding suffix tree edge. To carry out an insertion batch, the algorithm traces the suffix insertion path starting from the active point by following M-links (at the immediate ancestor of implicit nodes), and possibly some downward suffix tree edges, until the end point (and thus the new active point) is reached. At each step in this process, Ukkonen's algorithm navigates the suffix tree by selecting the edges at each branching node according to their first occurring symbol, quickly moving downward the suffix tree path towards the implicit node. This information is updated while tracing the suffix tree, selecting the appropriate branch at each internal suffix tree node according to the first symbol on the branching edges and verifying the remaining symbols on the current edge. While following M-links, however, some new suffix tree nodes may appear on the path between the suffix tree node and the offset representing the implicit node's locus and thus the implicit node's representation has to be updated to the *canonical form* specifying the locus of implicit nodes by their offset relative to the beginning of the edge where they are now located.

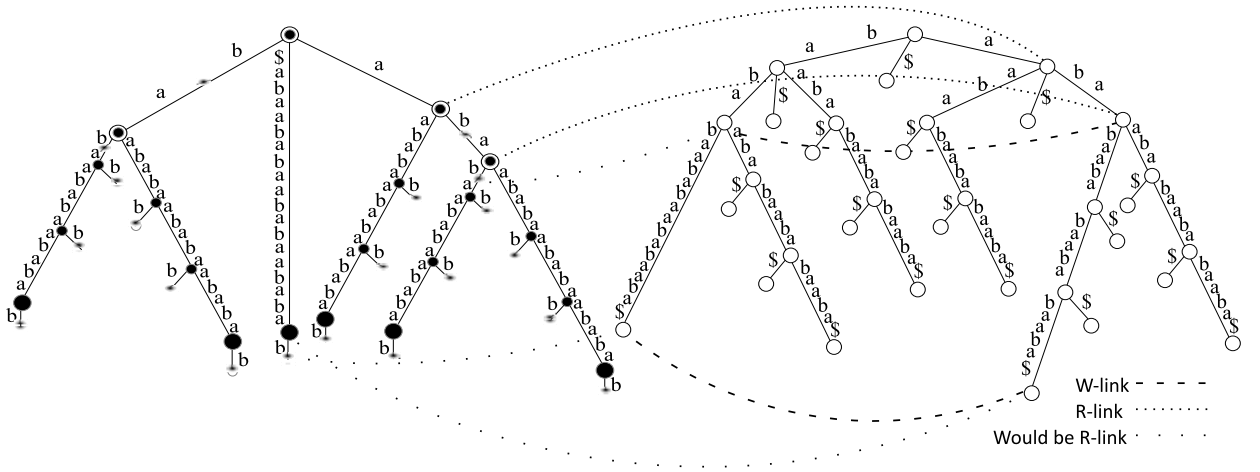

Fig. 8. R-links between the suffix tree of the text $\$w = \$ababaababaabaababa$ (left, see Fig. 7) and the corresponding reverse suffix tree $\widetilde{w}\$ = ababaababaabaababa\$$ (right). The suffix trees are depicted as the text $\$w$ is extended on the right by the symbol b, where the grey nodes indicate the new leaves representing the extended suffixes that branch out of the suffix tree. The leftmost edge in the right tree exists only after inserting the new symbol b, and the insertion point of the next suffix $b\widetilde{w}\$$ in the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$ is at locus $baba$, which is exactly the reverse of Ukkonen's new active point in the suffix tree $\mathcal{T}_{\$wb}$; this active point is reached by following the R-link from node $aba = \mathcal{M}(baba)$ in the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$ to its reverse aba in the suffix tree $\mathcal{T}_{\$wb}$ and then extending this locus on the right by b. The suffix chain of $\mathcal{T}_{\$w}$ corresponds to all prefixes of the reverse text $\widetilde{w}\$ = ababaababaababaaba\$$, and in particular the suffix insertion path of $\mathcal{T}_{\$w}$ corresponds to all the prefixes of the reverse text $\widetilde{w}\$$ between $ababaababaaba$ and (excluding) aba. Depicted "*would be R-links*" are shown for illustrative purpose and are not maintained for leaves and implicit nodes.

Ukkonen showed that the total amount of work required by the algorithm is linear in the input text, and thus it can be amortized to constant time per text symbol. First, the total number of leaves inserted in the suffix tree (i.e., both leaf and branching node insertions tasks) can be amortized against the total number of symbols in the text. Second, the total work performed during the canonization steps can be amortized against the suffix tree depth. Despite its amortized efficiency, a single insertion batch might still require the insertion of many new suffixes and the execution of many canonization steps, and therefore might be very expensive in the worst case. Observe that unlike Weiner's algorithm, Ukkonen's algorithm has a dual amortization argument: one for the node insertion based on the text length and another for the canonization steps based on the suffix tree depth (which is in turn bounded by the text length).

5.2. Quasi real-time left-to-right construction

Our quasi real-time adaptation of Ukkonen's algorithm hinges on a de-amortization of the time consuming visible insertion batches, which is reminiscent of our de-amortization of the invisible W-links adjustments in Weiner's algorithm described in Section 4. The high-level ideas behind our approach are the following. We maintain a node insertion *de-amortization stack* with the delayed visible suffix tree leaf and the branching node insertion tasks, that will be performed from short to long insertion points, i.e., in reverse order with respect to Ukkonen's original algorithm. We get to the end of the suffix insertion path by using our adaptation of Weiner's right-to-left algorithm: we maintain the suffix tree $\mathcal{T}_{\widetilde{w}\$}$ of the reverse text and use R-links to connect from the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$ back to our left-to-right suffix tree $\mathcal{T}_{\$w}$. We then traverse the suffix insertion path backwards by following W-links: recall that hard W-links are the opposite of the M-links used by Ukkonen's original algorithm and as we shall see, the soft W-links can be used similarly with an additional efficiency benefit that avoids the canonization steps in Ukkonen's algorithm. Thus, our adaptation of Ukkonen's algorithm requires W-links instead of Ukkonen's M-links: we will show how to maintain these invisible W-links in real-time via a suitable relaxation of the suffix tree representation.

R-links from $\mathcal{T}_{\widetilde{w}\$}$ to $\mathcal{T}_{\$w}$. The following straightforward observation, that is illustrated in Fig. 8, demonstrates the relationship between the active point in Ukkonen's algorithm and the insertion point in Weiner's algorithm as applied to the reverse text, laying the foundation to our simultaneous construction of the right-to-left reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$ in tandem with the left-to-right suffix tree $\mathcal{T}_{\$w}$.

Lemma 5.1. *Ukkonen's algorithm active point in the suffix tree $\mathcal{T}_{\$wa}$ is the reverse of Weiner's algorithm insertion point of the suffix $a\widetilde{w}\$$ in the suffix tree $\mathcal{T}_{\widetilde{w}\$}$.*

Proof. In Ukkonen's algorithm, the active point v in $\mathcal{T}_{\$wa}$ is the longest repeated suffix of the text $\$wa$ that also appeared earlier in $\$w$. This is equivalent to saying that the reverse string $\widetilde{v}$ is the longest repeated prefix of the reverse text $a\widetilde{w}\$$ that appeared earlier in $\widetilde{w}\$$. Thus, $\widetilde{v}$ is the insertion point of the suffix $a\widetilde{w}\$$ in $\mathcal{T}_{\widetilde{w}\$}$ for Weiner's algorithm. □

Assume that both suffix trees $\mathcal{T}_{\$w}$ and $\mathcal{T}_{\widetilde{w}\$}$ are available. First, we compute the insertion point of the new suffix $a\widetilde{w}\$$ in $\mathcal{T}_{\widetilde{w}\$}$, by trying to extend from right to left the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$ with symbol a. By using our quasi real-time implementation of Weiner's right-to-left algorithm presented in Section 4, we are guaranteed that the insertion point of the new suffix $a\widetilde{w}\$$ in $\mathcal{T}_{\widetilde{w}\$}$ is computed right at the first step, i.e., after $O(\log\log n)$ time. By Lemma 5.1, we can use this information to jump ahead at the end of the insertion batch produced by Ukkonen's algorithm when the text $\$w$ is extended from left to right with symbol a. To do that, we maintain the two suffix trees $\mathcal{T}_{\$w}$ and $\mathcal{T}_{\widetilde{w}\$}$ in tandem, by executing each step of our quasi real-time implementation of Weiner's right-to-left algorithm applied to the reverse text $\widetilde{w}$, immediately before each step of our quasi real-time implementation of Ukkonen's left-to-right algorithm, to be presented next.

Lemma 5.2. *We can find Ukkonen's new active point in $\mathcal{T}_{\$wa}$ in constant time with the help of Weiner's reverse suffix tree $\mathcal{T}_{a\widetilde{w}\$}$, provided that the R-links are correctly maintained.*

Proof. First, we apply the text extension to our adaptation of Weiner's algorithm to get $\mathcal{T}_{a\widetilde{w}\$}$. There are two cases. If the new insertion point v is exactly one symbol longer than the old insertion point, then the old insertion point was u, the immediate ancestor of $\widetilde{w}\$$ with defined W-link $\mathcal{W}_a(u) = v$. In this case, the length of Ukkonen's active point also increases by one and its locus is computed by moving downward by symbol a in the suffix tree $\mathcal{T}_{\$w}$.

Otherwise, the search for the ancestor u with defined W-link $\mathcal{W}_a(u)$ took a few steps up the reverse suffix tree $\mathcal{T}_{a\widetilde{w}\$}$. We claim that in Weiner's algorithm, the node u must have another second W-link $\mathcal{W}_b(u)$ defined for some alphabet symbol $b \neq a$, in addition to the W-link $\mathcal{W}_a(u)$. Indeed, in this case, in the suffix tree $\mathcal{T}_{\$w}$ there must be at least one suffix tree node, say z, between $\widetilde{w}\$$ and u, and z must have a W-link $\mathcal{W}_b(z)$ defined. Note that it must be $b \neq a$, otherwise u would not be the lowest ancestor of $\widetilde{w}\$$ with $\mathcal{W}_a(u)$ defined. Furthermore, $\mathcal{W}_b(u)$ must be defined by Lemma 3.3 since u is an ancestor of z. Since node u has W-links $\mathcal{W}_a(u)$ and $\mathcal{W}_b(u)$ defined, $b \neq a$, $\widetilde{u}$ must be a branching node in Ukkonen's suffix tree $\mathcal{T}_{\$w}$. Consequently, we can get to the node $\widetilde{u}$ in $\mathcal{T}_{\$w}$ in constant time by following the R-link from u in the reverse suffix tree $\mathcal{T}_{a\widetilde{w}\$}$ to $\mathcal{T}_{\$w}$, and then advance to the locus of $\widetilde{u}a$ by following the alphabet symbol a in $\mathcal{T}_{\$w}$. □

Note that in the proof of Lemma 5.2, after following an R-link, we still need to move one step down to get to the proper locus in the suffix tree $\mathcal{T}_{\$w}$. This is due to the fact that we did not define R-links for leaves or for implicit nodes of $\mathcal{T}_{\$w}$, but only for internal branching nodes. The main reason for this restriction is that R-links for implicit nodes would be too expensive to maintain, while R-links for leaves do not appear to be very useful. First, we analyze implicit nodes, which play a role in the suffix chain of $T_{\$w}$. In particular, there is a mapping between suffixes in $T_{\$w}$ and prefixes in the reverse tree $T_{\widetilde{w}\$}$. Consider indeed the longest leaf $\$w \in \mathcal{T}_{\$w}$ representing the whole text $\$w$ and the longest leaf representing the reverse whole text $\widetilde{w}\$ \in \mathcal{T}_{\widetilde{w}\$}$: the suffix chain in $T_{\$w}$, i.e., the nodes on the path of M-links starting at $\$w \in \mathcal{T}_{\$w}$, maps to the loci of the reverse prefixes in $T_{\widetilde{w}\$}$ of the leaf $\widetilde{w}\$$ (see e.g., Fig. 8). R-links for implicit nodes would have made it possible to trace those two paths side by side. However, had we defined R-links for implicit nodes, when the text $\$w$ is being extended from left to right, all the suffixes the suffix chain in $T_{\$w}$ would have to be extended simultaneously, and consequently the R-links of all those implicit nodes would have to be updated. The leaves in $T_{\$w}$ do not change with suffix extensions, and thus the maintenance of R-links for those leaves would not be problematic. However, R-links for leaves do not look particularly useful either, as in our approach R-links are used to get to internal nodes of the suffix tree $T_{\$w}$. Fig. 8 shows also some of those undefined R-links for leaves or for implicit nodes as "would be R-links".

W-links in Ukkonen's algorithm. W-links replace the M-links in our adaptation of Ukkonen's algorithm. We first prove that W-links are crucial for tracking efficiently the suffix insertion path, then explain how W-links are affected by each left-to-right suffix extension, and finally present a relaxed suffix tree representation that permits to maintain the W-links correctly in real-time. We start by proving that the suffix insertion path can be traced backwards in linear time in its length, as shown by the following lemma. Note that this avoids also the extra amortization due to the canonization steps in Ukkonen's algorithm.

Lemma 5.3. *Given the end point, the suffix insertion path may be traced in increasing order of the insertion point lengths in time proportional to its size by properly following W-links.*

Proof. The first (possibly empty) segment consisting of explicit nodes can be traced in linear time by following hard W-links (the opposite of the M-links used by Ukkonen's algorithm) starting from the end point. If the suffix insertion path contains implicit nodes, we can then jump from the longest explicit suffix to the shortest implicit suffix by following a soft W-link to the explicit node just below the implicit node. From now on, we can trace the second segment of the suffix insertion path following again soft W-links: from each implicit node, we can jump via a soft W-link from the lower endpoint of the edge that is being split to the explicit node just below the next node in the suffix insertion path. Note that each new node splitting an edge will inherit the W-links from the explicit node just below it as in Weiner's algorithm, i.e., at the lower endpoint of the split edge. □

We now analyze the effect of left-to-right suffix extensions on W-links. Adjustments involving hard W-links can be easily taken care of, as they are simply the opposite of M-links that are eventually created with the new branching nodes and

leaves. Hence, here we focus mainly on the creation of new soft W-links and on the updates to existing soft W-links, which are more involved. Recall that, when extended from left to right, the suffix tree may undergo two different types of steps: an *insertion step*, when the suffix extension causes the active point in Ukkonen's algorithm to branch out of the suffix tree, introducing a non-empty insertion batch that creates new internal branching nodes and new leaves; and an *idle step*, when the suffix extension only moves the active point within the suffix tree, without producing any change in terms of the visible suffix tree (apart from implicit nodes).

As a consequence of an insertion step, all the new branching nodes and the new leaves must be eventually inserted into the suffix tree and must be connected by hard W-links and their opposite M-links. As mentioned before, all those leaf and branching node insertions will be postponed, together with their associated hard W-links, by using a de-amortization stack. The soft W-links affected by an insertion step can be characterized as follows: the insertion of each new internal branching node splits an existing edge and, similarly to Weiner's algorithm, may consequently split the soft W-links pointing to the end of that edge: the shallower W-links must be updated to point to the new internal branching node and the deeper W-links will not be affected by the change. Since updating all the soft W-links at once may be too expensive in our real-time scenario, we will prepare for those updates proactively, i.e., without waiting for the actual insertion of the new branching node. To do this, we consider also the locations of all the implicit nodes, and maintain information about the implicit W-links between implicit nodes (which are not represented explicitly) and explicit nodes. By updating the information about those implicit W-links at each idle step, we will be prepared for all the soft W-link updates required by a branching node insertion.

We first claim that an idle step may create at most one new implicit soft W-link or may update at most one existing soft W-link. Indeed, the active point must have a W-link to the shortest leaf, which is its predecessor on the suffix chain. This W-link will be implicit if the active point is an implicit node, and it will become explicit whenever the active point coincides with an existing explicit internal branching node. As a result, those W-links may be created explicitly every time the active point encounters an internal branching node (during an idle step) and also eventually when the active point is inserted as an explicit branching node through the node insertion de-amortization process (during an insertion step). This can be easily performed within our time bounds and accounts for the new soft W-link creations.

We now turn to soft W-link updates. As the active point moves down during an idle step, also the implicit nodes advance within the corresponding suffix tree edges, and this may cause some soft W-links to be updated. An important observation is that at each idle step only the soft W-link from the current longest explicit suffix needs to be updated. More precisely, if the suffix chain contains implicit nodes, then this W-link needs to be moved to point to the shortest implicit suffix. Note that if the suffix chain contains no implicit nodes, then the shortest implicit suffix is undefined and there is no update involving soft W-links. In this case, the longest explicit suffix and the active point coincide, and this explicit node must get a new hard W-link to the shortest leaf as explained before.

To cope efficiently with soft W-link updates, we will relax our suffix tree representation and have all soft W-links point to *shadow nodes*. Shadow nodes represent implicit nodes that have a soft W-link (from an explicit node): a shadow node has a pointer to the explicit node at the end of the corresponding edge, and can be thought as "floating" in the middle of that edge. Throughout a sequence of idle steps, implicit suffixes move down the suffix tree and shadow nodes may get additional W-links from the explicit nodes at the end of their corresponding edges. If an implicit node branches out of the suffix tree because of an insertion step, the corresponding shadow node has all the W-links correctly in place and can thus be inserted as the required branching node. If a shadow node eventually reaches the end of the corresponding edge without branching out of the suffix tree, it merges with the node at the end of that edge. This merge can be done at no extra cost: when a shadow node coincides with the node at the end of its edge, all the soft W-links pointing to the end of the edge have been already transferred to the shadow node, and thus the shadow node can simply replace that node that becomes obsolete. This yields the following lemma.

Lemma 5.4. *In each idle step at most one soft W-link from the longest explicit suffix needs to be updated. This soft W-link may be identified in time* $O(\log\log n)$ *in the worst case, provided that the R-links are correctly maintained.*

Proof. We have shown before that the only soft W-link that needs to be updated during an idle step is the W-link of the longest explicit suffix. We now show that this soft W-link can be found efficiently. Let $\$w$ be the current text, $\mathcal{T}_{\$w}$ the corresponding left-to-right suffix tree, $\widetilde{w}\$$ the reverse text and $\mathcal{T}_{\widetilde{w}\$}$ the reverse right-to-left suffix tree. Let v be the longest explicit suffix in $\mathcal{T}_{\$w}$: namely, v is the longest suffix of $\$w$ such that $\mathcal{T}_{\$w}$ contains also the substrings vb and vc, for alphabet symbols $b \neq c$. This implies that $\widetilde{v}$ is the nearest ancestor of $\widetilde{w}\$$, which has at least two different W-links in $\mathcal{T}_{\widetilde{w}\$}$. In other words, $\widetilde{v}$ is the nearest ancestor in the core of the suffix tree $\mathcal{T}_{\widetilde{w}\$}$ and thus $\widetilde{v}$ can be found by performing one fringe marked ancestor query for the nearest core ancestor in $O(\log\log n)$ time. Once $\widetilde{v}$ is located, we can find the longest explicit suffix v by following the R-link to the suffix tree $\mathcal{T}_{\$w}$. □

De-amortization. In order to de-amortize the numerous operations in each insertion batch of Ukkonen's algorithm, we make use of a de-amortization stack. Specifically, once we jump ahead to the end point of Ukkonen's suffix insertion path as previously explained, we push the whole insertion batch onto the de-amortization stack: the leaf and branching node insertions will be delayed, so that they can be de-amortized over time and consequently performed in reverse order, i.e., going from short to long insertion points. Similarly to the de-amortization stack used in our adaptation of Weiner's algorithm

described in Section 4, we do not need to push explicitly all the insertion batch onto the de-amortization stack, but we can do this implicitly in constant time by just pushing the end point of the suffix insertion path onto the stack. Each time we need to access the next node on the suffix insertion path, we can do this in constant time with the help of the W-links as described in Lemma 5.3: when we pop a node from the de-amortization stack, we can thus push the next node on the suffix insertion path in constant time. Unlike our adaptation of Weiner's algorithm, however, which contained W-link adjustment tasks at increasing *depths*, this time we will maintain the invariant that the de-amortization stack contains insertion tasks whose *length* is strictly increasing. Since the length of the active point in Ukkonen's algorithm may increase at most by one with each input symbol, if at each step we insert into the suffix tree at least one pending suffix from the de-amortization stack, we are guaranteed that the lengths of the pending suffix insertion tasks that are on the de-amortization stack will be always strictly increasing, and thus we will ensure monotonicity in the stack. Moreover, if at each step at least two pending suffixes from the de-amortization stack are inserted into the suffix tree, we will be able to catch up eventually with all the delayed insertions, thanks to the amortized analysis of Ukkonen's algorithm.

Thus, it remains to show that throughout this de-amortization process, the R-links from the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$ to the suffix tree $\mathcal{T}_{\$w}$ can be correctly maintained, which is crucial for Lemmas 5.2 and 5.4.

Lemma 5.5. *The R-links between the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$ and the suffix tree $\mathcal{T}_{\$w}$ can be correctly maintained in both directions.*

Proof. We must show that R-links are correctly maintained when new nodes are added to the suffix tree $\mathcal{T}_{\$w}$ and to the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$. When Weiner's algorithm inserts a new node $\widetilde{v}$ into the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$, Lemma 5.2 gives us the corresponding locus of the Ukkonen's new active point in $\mathcal{T}_{\$w}$: if it is a node in the suffix tree, then we set the R-link accordingly.

An insertion batch in Ukkonen's algorithm is de-amortized, but since we have the new active point, we can proceed while de-amortizing the node insertions, yet still set the R-links when the nodes are eventually inserted by the de-amortization mechanism from shallow to deep. The corresponding loci in Weiner's reverse suffix tree are the contiguous ancestor loci of Weiner's old insertion point, between the node $\widetilde{u}$ that had its W-link set and the old insertion point of $\widetilde{w}\$$; this path can be simply traversed while de-amortizing the corresponding node insertions and each R-link can be set correctly in constant time. See Fig. 8. □

In summary, Weiner's algorithm is first applied to the reverse right-to-left text and produces the end point of the insertion batch. The W-link adjustment tasks are executed in real-time as explained above and the node and leaf insertion tasks are put on the de-amortization stack and executed later, according to increasing insertion point lengths. The low-level details of this de-amortization stack are similar to Section 4, and thus are omitted. This gives the following theorem and corollary, which are the analogues of Theorem 4.1 and Corollary 4.1.

Theorem 5.6. *We can adapt Ukkonen's left-to-right on-line suffix tree algorithm over constant size alphabets to take up to $O(\log\log n)$ time processing each input symbol and spend $O(n\log\log n)$ time in total.*

Corollary 5.7. *An algorithm may access the suffix tree constructed for the left-to-right extended text and its internal invisible M-links and W-links. The suffix tree and the suffix links will be available "just in time", if the algorithm executes the delayed update tasks from the de-amortization stack, provided that such algorithm traverses the suffix tree starting from the root, one symbol deeper in each step.*

6. Conclusion

In this paper we have contributed a further step towards the plausible *real-time* construction of suffix trees by designing on-line algorithms that spend only $O(\log\log n)$ time processing each input symbol and take $O(n\log\log n)$ time in total. We have presented both a right-to-left algorithm based on Weiner's approach, and a left-to-right algorithm based on Ukkonen's approach. In their work, Inenaga [33] and Maaß [41] combined Weiner's [49] right-to-left algorithm and Ukkonen's [48] left-to-right algorithm to obtain bi-directional on-line linear-time suffix tree and affix tree [47] construction algorithms that may extend the *same text* on either end. We omit the details, which are out of the scope of this paper, but remark that by combining our adaptations of Weiner's and Ukkonen's algorithms with Inenaga's and Maaß' observations, it is also possible to symbiotically construct in quasi real-time the suffix tree of a bi-directionally extended text and of the reverse text and consequently, the affix tree.

There are several open questions related to this work in general and in particular to the fringe marked ancestor problem. First and foremost, is it possible to compute the suffix tree insertion points, or the longest previous factors, on-line in real time, or at least in $o(\log\log n)$ worst-case time? Alstrup, Husfeldt and Rauhe [1] also give a colored version of their nearest marked ancestor data structure. The fringe marked ancestor problem might also be similarly generalized to efficiently handle marks in various colors while sharing the same skeleton tree, what could be useful when the alphabet size is larger, since the total number of colored marks is still linear by Lemma 3.4. We are also interested in other uses for the new fringe marked ancestor data structure.

We presented near real-time algorithms for the construction of suffix trees and their invisible suffix links. These algorithms also build the Directed Acyclic Word Graph (DAWG) [8,9,14,16,18,40]: the DAWG of a text shares the same set of nodes with the suffix tree of the reverse text and its transitions are precisely the W-links of that suffix tree. We did not consider extending our results to the direct construction of the Compact Directed Acyclic Word Graph (CDAWG) [7,9,19,34], which may be an interesting related problem.

In some circumstances, it might also be useful to proceed with the suffix tree construction and the same time, use the available suffix tree in some search, a situation that requires suitably careful definitions and was considered, for example, by Amir and Nor [2]. Our de-amortized construction might be useful in such settings, but we must replace the basic stack used in this paper to delay the suffix tree updates, with a generalized stack-like data structure that allows a second sweep in the middle to clear up the delayed construction tasks, while also simultaneously adding to the stack newly delayed tasks resulting from later suffix tree construction steps.

Finally, there exist off-line linear-time algorithms for suffix tree and suffix array construction over larger integer alphabets [25,26,35–37,45] (DAWG and Aho–Corasick Automata by reductions [11,24]). We are curious whether linear-time on-line suffix tree construction algorithms exist over large integer alphabets.

Acknowledgements

We thank the anonymous referees for their suggestions and Amir Ben-Amram, Johannes Fischer, Roberto Grossi, Gadi Landau, Maxime Crochemore and Oren Weimann for discussions about this work.

References

[1] S. Alstrup, T. Husfeldt, T. Rauhe, Marked ancestor problems, in: FOCS, 1998, pp. 534–544.
[2] A. Amir, I. Nor, Real-time indexing over fixed finite alphabets, in: S.-H. Teng (Ed.), SODA, SIAM, 2008, pp. 1086–1095.
[3] A. Amir, M. Farach, R. Idury, J.L. Poutré, A. Schäffer, Improved dynamic dictionary-matching, Inform. and Comput. 119 (1995) 258–282.
[4] A. Amir, G. Franceschini, R. Grossi, T. Kopelowitz, M. Lewenstein, N. Lewenstein, Managing unbounded-length keys in comparison-driven data structures with applications to on-line indexing, Manuscript, 2011.
[5] A. Amir, T. Kopelowitz, M. Lewenstein, N. Lewenstein, Towards real-time suffix tree construction, in: M.P. Consens, G. Navarro (Eds.), SPIRE, in: Lecture Notes in Computer Science, vol. 3772, Springer, 2005, pp. 67–78.
[6] A. Amir, G.M. Landau, E. Ukkonen, Online timestamped text indexing, Inform. Process. Lett. 82 (5) (2002) 253–259.
[7] A. Apostolico, S. Lonardi, A speed-up for the commute between subword trees and DAWGs, Inform. Process. Lett. 83 (3) (2002) 159–161.
[8] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, J. Seiferas, The smallest automaton recognizing the subwords of a text, Theoret. Comput. Sci. 40 (1985) 31–55.
[9] A. Blumer, J. Blumer, D. Haussler, R. McConnel, A. Ehrenfeucht, Complete inverted files for efficient text retrieval and analysis, J. ACM 34 (3) (1987) 578–595.
[10] D. Breslauer, The suffix tree of a tree and minimizing sequential transducers, Theoret. Comput. Sci. 191 (1–2) (1998) 131–144.
[11] D. Breslauer, R. Hariharan, Optimal parallel construction of minimal suffix and factor automata, Parallel Process. Lett. 6 (1) (1996) 35–44.
[12] D. Breslauer, G.F. Italiano, Near real-time suffix tree construction via the fringe marked ancestor problem, in: 18th Intern. Symp. on String Processing and Information Retrieval, 2011, pp. 156–167.
[13] D. Breslauer, G.F. Italiano, On suffix extensions in suffix trees, in: 18th Intern. Symp. on String Processing and Information Retrieval, 2011, pp. 301–312.
[14] M. Chen, J. Seiferas, Efficient and elegant subword-tree construction, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, in: NATO ASI Series F, vol. 12, Springer-Verlag, Berlin, Germany, 1985, pp. 97–107.
[15] R. Cole, R. Hariharan, Dynamic LCA queries on trees, SIAM J. Comput. 34 (4) (2005) 894–923.
[16] M. Crochemore, Transducers and repetitions, Theoret. Comput. Sci. 12 (1986) 63–86.
[17] M. Crochemore, L. Ilie, Computing longest previous factor in linear time and applications, Inform. Process. Lett. 106 (2) (2008) 75–80.
[18] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[19] M. Crochemore, R. Vérin, Direct construction of compact directed acyclic word graphs, in: A. Apostolico, J. Hein (Eds.), CPM, in: Lecture Notes in Computer Science, vol. 1264, Springer, 1997, pp. 116–129.
[20] M. Crochemore, L. Ilie, C.S. Iliopoulos, M. Kubica, W. Rytter, T. Walen, LPF computation revisited, in: J. Fiala, J. Kratochvíl, M. Miller (Eds.), IWOCA, in: Lecture Notes in Computer Science, vol. 5874, Springer, 2009, pp. 158–169.
[21] M. Crochemore, C.S. Iliopoulos, M. Kubica, W. Rytter, T. Walen, Efficient algorithms for two extensions of LPF table: the power of suffix arrays, in: J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, B. Rumpe (Eds.), SOFSEM, in: Lecture Notes in Computer Science, vol. 5901, Springer, 2010, pp. 296–307.
[22] M. Crochemore, S.J. Puglisi, G. Tischler, Computing rightmost positions of longest previous factors, Manuscript, 2011.
[23] P.F. Dietz, R. Raman, Persistence, amortization and randomization, in: SODA, 1991, pp. 78–88.
[24] S. Dori, G.M. Landau, Construction of Aho Corasick automaton in linear time for integer alphabets, Inform. Process. Lett. 98 (2) (2006) 66–72.
[25] M. Farach, Optimal suffix tree construction with large alphabets, in: FOCS, 1997, pp. 137–143.
[26] M. Farach-Colton, P. Ferragina, S. Muthukrishnan, On the sorting-complexity of suffix tree construction, J. ACM 47 (6) (2000) 987–1011.
[27] P. Ferragina, I. Nitto, R. Venturini, On the bit-complexity of Lempel–Ziv compression, in: C. Mathieu (Ed.), SODA, SIAM, 2009, pp. 768–777.
[28] G. Franceschini, R. Grossi, A general technique for managing strings in comparison-driven data structures, in: J. Díaz, J. Karhumäki, A. Lepistö, D. Sannella (Eds.), ICALP, in: Lecture Notes in Computer Science, vol. 3142, Springer, 2004, pp. 606–617.
[29] Z. Galil, Open problems in stringology, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, in: NATO ASI Series F, vol. 12, Springer-Verlag, Berlin, Germany, 1984, pp. 1–8.
[30] R. Giegerich, S. Kurtz, From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction, Algorithmica 19 (3) (1997) 331–353.
[31] R. Grossi, G.F. Italiano, Efficient techniques for maintaining multidimensional keys in linked data structures, in: J. Wiedermann, P. van Emde Boas, M. Nielsen (Eds.), ICALP, in: Lecture Notes in Computer Science, vol. 1644, Springer, 1999, pp. 372–381.
[32] D. Gusfield, Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology, Cambridge University Press, 1997.
[33] S. Inenaga, Bidirectional construction of suffix trees, Nordic J. Comput. 10 (1) (2003) 52.
[34] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, G. Pavesi, On-line construction of compact directed acyclic word graphs, Discrete Appl. Math. 146 (2) (2005) 156–179.

[35] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (6) (2006) 918–936.
[36] D.K. Kim, J.S. Sim, H. Park, K. Park, Constructing suffix arrays in linear time, J. Discrete Algorithms 3 (2–4) (2005) 126–142.
[37] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, J. Discrete Algorithms 3 (2–4) (2005) 143–156.
[38] T. Kopelowitz, From off-line to on-line indexing data-structures, PhD thesis, Dept. of Computer Science, Bar-Ilan University, 2011.
[39] S.R. Kosaraju, Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version), in: STOC, 1994, pp. 310–316.
[40] M. Lothaire, Applied Combinatorics on Words, Cambridge University Press, 2005.
[41] M.G. Maaß, Linear bidirectional on-line construction of affix trees, Algorithmica 37 (1) (2003) 43–74.
[42] U. Manber, E.W. Myers, Suffix arrays: A new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[43] E. McCreight, A space economical suffix tree construction algorithm, J. ACM 23 (1976) 262–272.
[44] J. Seiferas, Subword trees, Manuscript, undated.
[45] T. Shibuya, Constructing the suffix tree of a tree with a large alphabet, in: A. Aggarwal, C.P. Rangan (Eds.), ISAAC, in: Lecture Notes in Computer Science, vol. 1741, Springer, 1999, pp. 225–236.
[46] A. Slisenko, Detection of periodicities and string-matching in real time, J. Sov. Math. (1983) 1316–1386.
[47] J. Stoye, Affix trees, Master's thesis, Technische Fakultat, Universitat Bielefeld, Bielefeld, Germany, 2000, Report 2000-04.
[48] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[49] P. Weiner, Linear pattern matching algorithms, in: Proc. 14th Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[50] J. Westbrook, Fast incremental planarity testing, in: Proc. 19th International Colloquium on Automata, Languages, and Programming, in: Lecture Notes in Computer Science, vol. 623, Springer-Verlag, Berlin, Germany, 1992, pp. 342–353.