Pointer-Machine Algorithms for Fully-Online Construction of Suffix Trees and DAWGs on Multiple Strings

Shunsuke Inenaga

Department of Informatics, Kyushu University, Japan
PRESTO, Japan Science and Technology Agency, Japan
inenaga@inf.kyushu-u.ac.jp

Abstract. We deal with the problem of maintaining the *suffix tree* indexing structure for a fully-online collection of strings, where a new character can be prepended to any string in the collection at any time. The only previously known algorithm for the problem, recently proposed by Takagi et al. [Algorithmica 82(5): 1346-1377 (2020)], runs in $O(N \log \sigma)$ time and $O(N)$ space *on the word RAM* model, where N denotes the total length of the strings and σ denotes the alphabet size. Their algorithm makes heavy use of the nearest marked ancestor (NMA) data structure on semi-dynamic trees, that can answer queries and supports insertion of nodes in $O(1)$ amortized time on the word RAM model. In this paper, we present a simpler fully-online right-to-left algorithm that builds the suffix tree for a given string collection in $O(N(\log \sigma + \log d))$ time and $O(N)$ space, where d is the maximum number of in-coming *Weiner links* to a node of the suffix tree. We note that d is bounded by the height of the suffix tree, which is further bounded by the length of the longest string in the collection. The advantage of this new algorithm is that it works on *the pointer machine model*, namely, it does not use the complicated NMA data structures that involve table look-ups. As a byproduct, we also obtain a pointer-machine algorithm for building the *directed acyclic word graph* (*DAWG*) for a fully-online left-to-right collection of strings, which runs in $O(N(\log \sigma + \log d))$ time and $O(N)$ space again without the aid of the NMA data structures.

1 Introduction

1.1 Suffix trees and DAWGs

Suffix trees are a fundamental string data structure with a myriad of applications [10]. The first efficient construction algorithm for suffix trees, proposed by Weiner [21], builds the suffix tree for a string in a right-to-left online manner, by updating the suffix tree each time a new character is prepended to the string. It runs in $O(n \log \sigma)$ time and $O(n)$ space, where n is the length of the string and σ is the alphabet size.

One of the most interesting features of Weiner's algorithm is a very close relationship to Blumer et al.'s algorithm [2] that builds the *directed acyclic word graph* (*DAWG*) in a left-to-right online manner, by updating the DAWG each time a new character is prepended to the string. It is well known (c.f. [4,5]) that the DAG of the Weiner links of the suffix tree of T is equivalent to the DAWG of the reversal $\overline{T}$ of T, or symmetrically, the suffix link tree of the DAWG of $\overline{T}$ is equivalent to the suffix tree of T. Thus, right-to-left online construction of suffix trees is essentially equivalent to left-to-right construction of DAWGs. This means that Blumer et al.'s DAWG construction algorithm also runs in $O(n \log \sigma)$ time and $O(n)$ space [2].

DAWGs also support efficient pattern matching queries, and have been applied to other important string problems such as local alignment [6], pattern matching with

Shunsuke Inenaga: Pointer-Machine Algorithms for Fully-Online Construction of Suffix Trees and DAWGs on Multiple Strings, pp. 111–124.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9 © Czech Technical University in Prague, Czech Republic

variable-length don't cares [14], dynamic dictionary matching [11], compact online Lempel-Ziv factorization [23], finding minimal absent words [8], and finding gapped repeats [18].

1.2 Fully online construction of suffix trees and DAWGs

Takagi et al. [17] initiated the generalized problem of maintaining the suffix tree for a collection of strings in a *fully-online manner*, where a new character can be prepended to any string in the collection at any time. This fully-online scenario arises in real-time database systems e.g. for sensor networks or trajectories. Takagi et al. showed that a direct application of Weiner's algorithm [21] to this fully-online setting requires one to visit $\Theta(N \min(K, \sqrt{N}))$ nodes, where N is the total length of the strings and K is the number of strings in the collection. Note that this leads to a worst-case $\Theta(N^{1.5} \log \sigma)$-time construction when $K = \Omega(\sqrt{N})$.

In their analysis, it was shown that Weiner's original algorithm applied to a fully-online string collection visits a total of $\Theta(N \min(K, \sqrt{N}))$ nodes. This means that the amortization argument of Weiner's algorithm for the number of nodes visited in the climbing process for inserting a new leaf, does not work for multiple strings in the fully-online setting. To overcome difficulty, Takagi et al. proved the three following statements: (1) By using σ nearest marked ancestor (NMA) structures [22], one can skip the last part of the climbing process; (2) All the σ NMA data structures can be stored in $O(n)$ space; (3) The number of nodes explicitly visited in the remaining part of each climbing process can be amortized to $O(1)$ per new added character. This led to their $O(N \log \sigma)$-time and $O(N)$-space fully-online right-to-left construction of the suffix tree for multiple strings.

Takagi et al. [17] also showed that Blumer et al.'s algorithm [2,3] applied to a fully-online left-to-right DAWG construction requires at least $\Theta(N \min(K, \sqrt{N}))$ work as well. They also showed how to maintain an *implicit* representation of the DAWG of $O(N)$ space which supports fully-online updates and simulates a DAWG edge traversal in $O(\log \sigma)$ time each. The key here was again the non-trivial use of the aforementioned σ NMA data structures over the suffix tree of the reversed strings.

As states above, Takagi et al.'s construction heavily relies on the use of the NMA data structures [22]. Although NMA data structures are useful and powerful, all known NMA data structures for (static and dynamic) trees that support $O(1)$ (amortized) time queries and updates [9,12,22] are quite involved, and they are valid only on the word RAM model as they use look-up tables that explicitly store the answers for small sub-problems. Hence, in general, it would be preferable if one could achieve similar efficiency without NMA data structures.

1.3 Our contribution

In this paper, we show how to maintain the suffix tree for a right-to-left fully-online string collection in $O(N(\log \sigma + \log d))$ time and $O(N)$ space, where d is the maximum number of in-coming *Weiner links* to a node of the suffix tree. Our construction does not use NMA data structures and works in the *pointer-machine model* [19,20], which is a simple computational model without address arithmetics. We note that d is bounded by the height of the suffix tree. Clearly, the height of the suffix tree is at most the maximum length of the strings. Hence, the d term can be dominated by the σ term when the strings are over integer alphabets of polynomial size in N, or when a

large number of strings of similar lengths are treated. To achieve the aforementioned bounds on the pointer-machine model, we reduce the problem of maintaining incoming Weiner links of nodes to the *ordered split-insert-find problem*, which maintains dynamic sets of sorted elements allowing for split and insert operations, and find queries, which can be solved in a total of $O(N \log d)$ time and $O(N)$ space.

As a byproduct of the above result, we also obtain the *first* non-trivial algorithm that maintains an *explicit* representation of the DAWG for fully-online left-to-right multiple strings, which runs in $O(N(\log \sigma + \log d))$ time and $O(N)$ space. By an explicit representation, we mean that every edge of the DAWG is implemented as a pointer. This DAWG construction does not require complicated table look-ups and thus also works on the pointer machine model.

2 Preliminaries

2.1 String notations

Let Σ be a general ordered alphabet. Any element of Σ^* is called a *string*. For any string T, let $|T|$ denote its length. Let ε be the empty string, namely, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma \setminus \{\varepsilon\}$. If $T = XYZ$, then X, Y, and Z are called a *prefix*, a *substring*, and a *suffix* of T, respectively. For any $1 \leq i \leq j \leq |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j in T. For any $1 \leq i \leq |T|$, let $T[i]$ denote the ith character of T. For any string T, let $\mathsf{Suffix}(T)$ denote the set of suffixes of T, and for any set $\mathcal{T}$ of strings, let $\mathsf{Suffix}(\mathcal{T})$ denote the set of suffixes of all strings in $\mathcal{T}$. Namely, $\mathsf{Suffix}(\mathcal{T}) = \bigcup_{T \in \mathcal{T}} \mathsf{Suffix}(T)$. For any string T, let $\overline{T}$ denote the reversed string of T, i.e., $\overline{T} = T[|T|] \cdots T[1]$. For any set $\mathcal{T}$ of strings, let $\overline{\mathcal{T}} = \{\overline{T} \mid T \in \mathcal{T}\}$.

2.2 Suffix trees and DAWGs for multiple strings

For ease of description, we assume that each string T_i in the collection $\mathcal{T}$ terminates with a unique character $\$_i$ that does not appear elsewhere in $\mathcal{T}$. However, our algorithms work without $\$_i$ symbols at the right end of strings as well.

A compacted trie is a rooted tree such that (1) each edge is labeled by a non-empty string, (2) each internal node is branching, and (3) the string labels of the out-going edges of each node begin with mutually distinct characters. The *suffix tree* [21] for a text collection $\mathcal{T}$, denoted $\mathsf{STree}(\mathcal{T})$, is a compacted trie which represents $\mathsf{Suffix}(\mathcal{T})$. The *string depth* of a node v of $\mathsf{Suffix}(\mathcal{T})$ is the length of the substring that is represented by v. We sometimes identify node v with the substring it represents. The suffix tree for a single string T is denoted $\mathsf{STree}(T)$.

$\mathsf{STree}(\mathcal{T})$ has at most $2N-1$ nodes and thus $2N-2$ nodes, since every internal node of $\mathsf{STree}(\mathcal{T})$ is branching and there are N leaves in $\mathsf{STree}(\mathcal{T})$. By representing each edge label x with a triple $\langle k, i, j \rangle$ of integers such that $x = T_k[i..j]$, $\mathsf{STree}(\mathcal{T})$ can be stored in $O(N)$ space.

We define the *suffix link* of each non-root node av of $\mathsf{STree}(\mathcal{T})$ with $a \in \Sigma$ and $v \in \Sigma^*$, by $\mathsf{slink}(av) = v$. For each explicit node v and $a \in \Sigma$, we also define the reversed suffix link (a.k.a. *Weiner link*) by $\mathsf{W_link}_a(v) = avx$, where $x \in \Sigma^*$ is the shortest string such that avx is a node of $\mathsf{STree}(\mathcal{T})$. $\mathsf{W_link}_a(v)$ is undefined if av is not a substring of strings in $\mathcal{T}$. A Weiner link $\mathsf{W_link}_a(v) = avx$ is said to be *hard* if $x = \varepsilon$, and *soft* if $x \in \Sigma^+$.

See the left diagram of Figure 1 for an example of $\mathsf{STree}(\mathcal{T})$ and Weiner links.

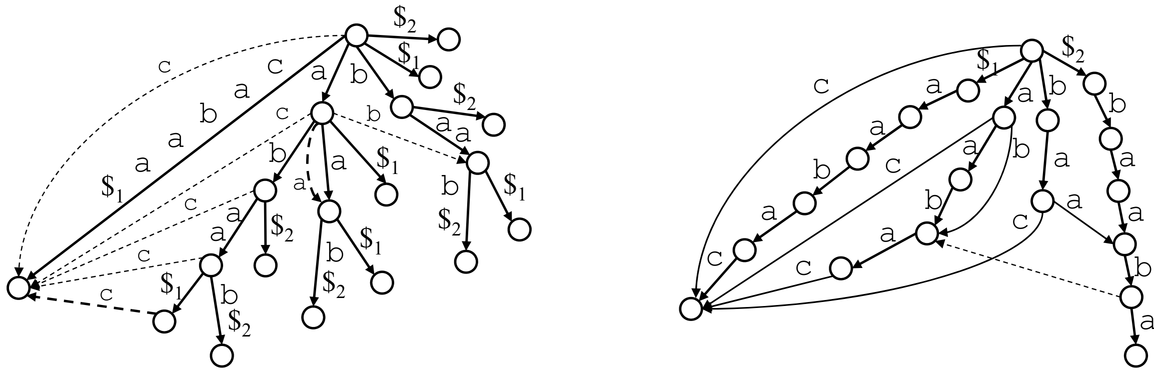

Figure 1. Left: $\mathsf{STree}(\mathcal{T})$ for $\mathcal{T} = \{\mathtt{cabaa}\$_1, \mathtt{abaab}\$_2\}$. The bold dashed arrows represent hard Weiner links, while the narrow dashed arrows represent soft Weiner links. Not all Weiner links are shown for simplicity. Right: $\mathsf{DAWG}(\mathcal{S})$ for $\mathcal{S} = \overline{\mathcal{T}} = \{\$_1\mathtt{aabac}, \$_2\mathtt{baaba}\}$. The dashed arrow represents a suffix link. Not all suffix links are shown for simplicity.

The *directed acyclic word graph* (*DAWG* for short) [2,3] of a text collection $\mathcal{S}$, denoted $\mathsf{DAWG}(\mathcal{S})$, is a (partial) DFA which represents $\mathsf{Suffix}(\mathcal{S})$. It is proven in [3] that $\mathsf{DAWG}(\mathcal{S})$ has at most $2N - 1$ nodes and $3N - 4$ edges for $N \geq 3$. Since each DAWG edge is labeled by a single character, $\mathsf{DAWG}(\mathcal{S})$ can be stored with $O(N)$ space. The DAWG for a single string S is denoted $\mathsf{DAWG}(S)$.

A node of $\mathsf{DAWG}(\mathcal{S})$ corresponds to the substrings in $\mathcal{S}$ which share the same set of ending positions in $\mathcal{S}$. Thus, for each node, there is a unique longest string represented by that node. For any node v of $\mathsf{DAWG}(\mathcal{S})$, let $long(v)$ denote the longest string represented by v. An edge (u, a, v) in the DAWG is called *primary* if $|long(u)| + 1 = |long(v)|$, and is called *secondary* otherwise. For each node v of $\mathsf{DAWG}(\mathcal{S})$ with $|long(v)| \geq 1$, let $\mathsf{slink}(v) = y$, where y is the longest suffix of $long(v)$ which is not represented by v.

Suppose $S = \overline{\mathcal{T}}$. It is known (c.f. [2,3,5]) that there is a node v in $\mathsf{STree}(\mathcal{T})$ iff there is a node x in $\mathsf{DAWG}(\mathcal{S})$ such that $long(x) = \overline{v}$. Also, the hard Weiner links and the soft Weiner links of $\mathsf{STree}(\mathcal{T})$ coincide with the primary edges and the secondary edges of $\mathsf{DAWG}(\mathcal{S})$, respectively. In a symmetric view, the reversed suffix links of $\mathsf{DAWG}(\mathcal{S})$ coincide with the suffix tree $\mathsf{STree}(\mathcal{T})$ for $\mathcal{T}$.

See Figure 1 for some concrete examples of the aforementioned symmetry. For instance, the nodes $\mathtt{abaa}$ and $\mathtt{baa}$ of $\mathsf{STree}(\mathcal{T})$ correspond to the nodes of $\mathsf{DAWG}(\mathcal{S})$ whose longest strings are $\overline{\mathtt{abaa}} = \mathtt{aaba}$ and $\overline{\mathtt{baa}} = \mathtt{aab}$, respectively. Observe that both $\mathsf{STree}(\mathcal{T})$ and $\mathsf{DAWG}(\mathcal{S})$ have 19 nodes each. The Weiner links of $\mathsf{STree}(\mathcal{T})$ labeled by character $\mathtt{c}$ correspond to the out-going edges of $\mathsf{DAWG}(\mathcal{S})$ labeled by $\mathtt{c}$. To see another example, the three Weiner links from node $\mathtt{a}$ in $\mathsf{STree}(\mathcal{T})$ labeled $\mathtt{a}$, $\mathtt{b}$, and $\mathtt{c}$ correspond to the three out-going edges of node $\{\mathtt{a}\}$ of $\mathsf{DAWG}(\mathcal{S})$ labeled $\mathtt{a}$, $\mathtt{b}$, and $\mathtt{c}$, respectively. For the symmetric view, focus on the suffix link of the node $\{\$_2\mathtt{baab}, \mathtt{baab}\}$ of $\mathsf{DAWG}(\mathcal{S})$ to the node $\{\mathtt{aab}, \mathtt{ab}\}$. This suffix link reversed corresponds to the edge labeled $\mathtt{b}\$_2$ from the node $\mathtt{baa}$ to the node $\mathtt{baab}\$_2$ in $\mathsf{STree}(\mathcal{T})$.

We now see that the two following tasks are essentially equivalent:

(A) Building $\mathsf{STree}(\mathcal{T})$ for a fully-online right-to-left text collection $\mathcal{T}$, using hard and soft Weiner links.

(B) Building $\mathsf{DAWG}(\mathcal{S})$ for a fully-online left-to-right text collection $\mathcal{S}$, using suffix links.

2.3 Pointer machines

A pointer machine [19,20] is an abstract model of computation such that the state of computation is stored as a directed graph, where each node can contain a constant amount of data (e.g. integers, symbols) and a constant number of pointers (i.e. out-going edges to other nodes). The instructions supported by the pointer machine model are basically creating new nodes and pointers, manipulating data, and performing comparisons. The crucial restriction in the pointer machine model, which distinguishes it from the word RAM model, is that pointer machines cannot perform address arithmetics, namely, memory access must be performed only by an explicit reference to a pointer. While the pointer machine model is apparently weaker than the word RAM model that supports address arithmetics and unit-cost bit-wise operations, the pointer machine model serves as a good basis for modeling linked structures such as trees and graphs, which are exactly our targets in this paper. In addition, pointer-machines are powerful enough to simulate list-processing based languages such as LISP and Prolog (and their variants), which have recurrently gained attention.

3 Brief reviews on previous algorithms

To understand why and how our new algorithms to be presented in Section 4 work efficiently, let us briefly recall the previous related algorithms.

3.1 Weiner's algorithm and Blumer et al.'s algorithm for a single string

First, we briefly review how Weiner's algorithm for a single string T adds a new leaf to the suffix tree when a new character a is prepended to T. Our description of Weiner's algorithm slightly differs from the original one, in that we use both hard and soft Weiner links while Weiner's original algorithm uses hard Weiner links only and it instead maintains Boolean vectors indicating the existence of soft Weiner links.

Suppose we have already constructed $\mathsf{STree}(T)$ with hard and soft Weiner links. Let ℓ be the leaf that represents T. Given a new character a, Weiner's algorithm climbs up the path from the leaf ℓ until encountering the deepest ancestor v of ℓ that has a Weiner link $\mathsf{W_link}_a(v)$ defined. If there is no such ancestor of ℓ above, then a new leaf representing aT is inserted from the root r of the suffix tree. Otherwise, the algorithm follows the Weiner link $\mathsf{W_link}_a(v)$ and arrives at its target node $u = \mathsf{W_link}_a(v)$. There are two sub-cases:

(1) If $\mathsf{W_link}_a(v)$ is a hard Weiner link, then a new leaf $\hat{\ell}$ representing aT is inserted from u.
(2) If $\mathsf{W_link}_a(v)$ is a soft Weiner link, then the algorithm splits the incoming edge of u into two edges by inserting a new node y as a new parent of u such that $|y| = |v| + 1$ (See also Figure 2). A new leaf representing aT is inserted from this new internal node y. We also copy each *out-going* Weiner link $\mathsf{W_link}_c(u)$ from u with a character c as an out-going Weiner link $\mathsf{W_link}_c(y)$ from y so that their target nodes are the same (i.e. $\mathsf{W_link}_c(u) = \mathsf{W_link}_c(y)$). See also Figure 3. Then, a new *hard* Weiner link is created from v to y with label a, in other words, an old soft Weiner link $\mathsf{W_link}_a(v) = u$ is *redirected* to a new hard Weiner link $\mathsf{W_link}_a(v) = y$. In addition, all the old soft Weiner links of ancestors z of v such that $\mathsf{W_link}_a(z) = u$ in $\mathsf{STree}(T)$ have to be redirected to new soft Weiner links $\mathsf{W_link}_a(z) = y$ in $\mathsf{STree}(aT)$. These redirections can be done by keeping climbing

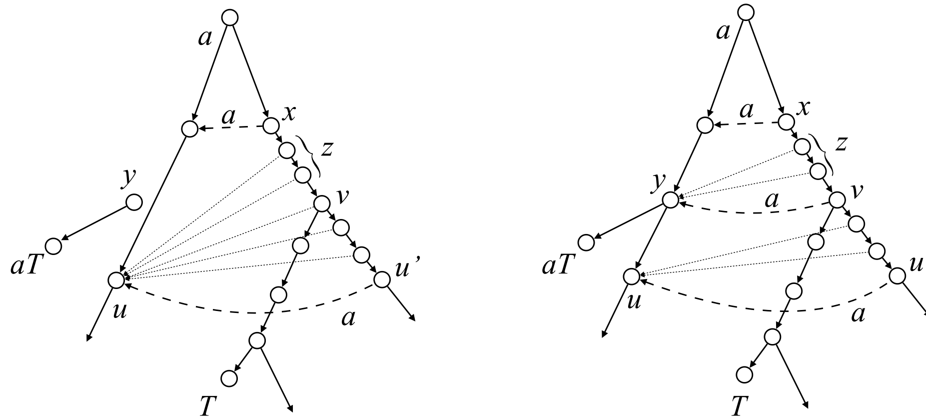

Figure 2. Left: Illustration for $\mathsf{STree}(T)$ of Case (2) before inserting the new leaf representing aT. Right: Illustration for $\mathsf{STree}(aT)$ of Case (2) after inserting the new leaf representing aT. In both diagrams, thick dashed arrows represent hard Winer links, and narrow dashed arrows represent soft Weiner links. All these Winer links are labeled by a. Also, new Weiner links labeled a are created from the nodes between the leaf for T and v to the new leaf for aT (not shown in this diagram).

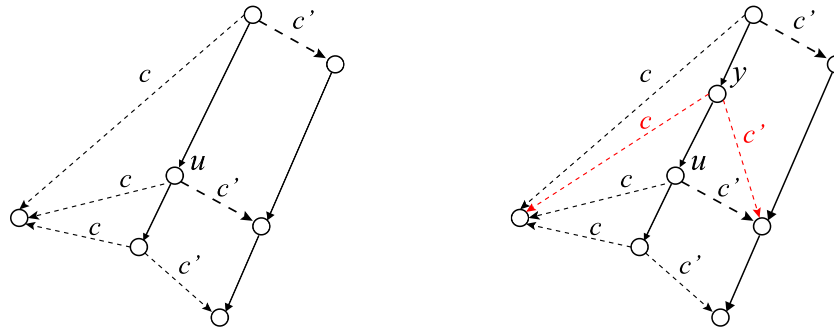

Figure 3. Illustration of the copy process of the out-going Weiner links of u to its new parent y in Case (2). Left: Out-going Weiner links of node u before the update. Right: Each out-going Winer link of node u is copied to its new parent y, represented by a red dashed arrow.

up the path from v until finding the deepest node x that has a hard Weiner link with character a pointing to the parent of u in $\mathsf{STree}(T)$.

In both Cases (1) and (2) above, new soft Weiner links $\mathsf{W_link}_a(x) = \hat{\ell}$ are created from every node x in the path from ℓ to the child of v.

The running time analysis of the above algorithm has three phases.

(a) In both Cases (1) and (2), the number of nodes from leaf ℓ for T to v is bounded by the number of newly created soft Weiner links. This is amortized $O(1)$ per new character since the resulting suffix tree has a total of $O(n)$ soft Weiner links [2], where $n = |T|$.
(b) In Case (2), the number of out-going Weiner links copied from u to y is bounded by the number of newly created Weiner links, which is also amortized $O(1)$ per new character by the same argument as (a).
(c) In Case (2), the number redirected soft Weiner links is bounded by the number of nodes from v to x. The analysis by Weiner [21] shows that this number of nodes from v to x can be amortized $O(1)$.

Wrapping up (a), (b), and (c), the total numbers of visited nodes, created Weiner links, and redirected Weiner links through constructing $\mathsf{STree}(T)$ by prepending n characters are $O(n)$. Thus Weiner's algorithm constructs $\mathsf{STree}(T)$ in a right-to-left online manner in $O(n \log \sigma)$ time with $O(n)$ space, where the $\log \sigma$ term comes from the cost of maintaining Weiner links of each node in the lexicographically sorted order by e.g. a standard balanced binary search tree.

Since this algorithm correctly maintains all (hard and soft) Weiner links, it builds $\mathsf{DAWG}(S)$ for the reversed string $S = \overline{T}$ in a left-to-right manner, in $O(n \log \sigma)$ time with $O(n)$ space. In other words, this version of Weiner's algorithm is equivalent to Blumer et al.'s DAWG online construction algorithm.

We remark that the aforementioned version of Weiner's algorithm, and equivalently Blumer et al.'s algorithm, work on the pointer machine model as they do not use address arithmetics nor table look-ups.

3.2 Takagi et al.'s algorithm for multiple strings on the word RAM

When Weiner's algorithm is applied to fully-online right-to-left construction of $\mathsf{STree}(\mathcal{T})$, the amortization in Analysis (c) does not work. Namely, it was shown by Takagi et al. [17] that the number of redirected soft Weiner links is $\Theta(N \min(K, \sqrt{N}))$ in the fully-online setting for multiple K strings. A simpler upper bound $O(NK)$ immediately follows from an observation that the insertion of a new leaf for a string T_i in $\mathcal{T}$ may also increase the depths of the leaves for all the other $K-1$ strings $T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_K$ in $\mathcal{T}$. Takagi et al. then obtained the aforementioned improved $O(N \min(K, \sqrt{N}))$ upper bound, and presented a lower bound instance that indeed requires $\Omega(N \min(K, \sqrt{N}))$ work. It should also be noted that the original version of Weiner's algorithm that only maintains Boolean indicators for the existence of soft Weiner links, must also visit $\Theta(N \min(K, \sqrt{N}))$ nodes [17].

Takagi et al. gave a neat way to overcome this difficulty by using the nearest marked ancestor (NMA) data structure [22] for a rooted tree. This NMA data structure allows for making unmarked nodes, splitting edges, inserting new leaves, and answering NMA queries in $O(1)$ amortized time each, in the word RAM model of machine word size $\Omega(\log N)$. Takagi et al. showed how to skip the nodes between v to x in $O(1)$ amortized time using a single NMA query on the NMA data structure associated to a given character a that is prepended to T. They also showed how to store σ NMA data structures for all σ distinct characters in $O(N)$ total space. Since the amortization argument (c) is no more needed by the use of the NMA data structures, and since the analyses (a) and (b) still hold for fully-online multiple strings, the total number of visited nodes was reduced to $O(N)$ in their algorithm. This led to their construction in $O(N \log \sigma)$ time and $O(N)$ space, in the word RAM model.

Takagi et al.'s $\Theta(N \min(K, \sqrt{N}))$ bound also applies to the number of visited nodes and that of redirected secondary edges of $\mathsf{DAWG}(\mathcal{S})$ for multiple strings in the fully-online setting. Instead, they showed how to simulate secondary edge traversals of $\mathsf{DAWG}(\mathcal{S})$ in $O(\log \sigma)$ amortized time each, using the aforementioned NMA structures. We remark that their data structure is only an implicit representation of $\mathsf{DAWG}(\mathcal{S})$ in the sense that the secondary edges are not explicitly stored.

4 Simple fully-online constructions of suffix trees and DAWGs on the pointer-machine model

In this section, we present our new algorithms for fully-online construction of suffix trees and DAWGs for multiple strings, which work on the pointer-machine model.

4.1 Right-to-left suffix tree construction

In this section, we present our new algorithm that constructs the suffix tree for a fully-online right-to-left string collection.

Consider a collection $\mathcal{T}' = \{T_1, \ldots, T_K\}$ of K strings. Suppose that we have built $\mathsf{STree}(\mathcal{T}')$ and that for each string $T_i \in \mathcal{T}'$ we know the leaf ℓ_i that represents T_i.

In our fully-online setting, any new character from Σ can be prepended to any string in the current string collection $\mathcal{T}$. Suppose that a new character $a \in \Sigma$ is prepended to a string T in the collection $\mathcal{T}'$, and let $\mathcal{T} = (\mathcal{T}' \setminus \{T\}) \cup \{aT\}$ be the collection after the update. Our task is to update $\mathsf{STree}(\mathcal{T}')$ to $\mathsf{STree}(\mathcal{T})$.

Our approach is to reduce the sub-problem of redirecting Weiner links to the *ordered split-insert-find problem* that operates on ordered sets over dynamic universe of elements, and supports the following operations and queries efficiently:

- Make-set, which creates a new list that consists only of a single element;
- Split, which splits a given set into two disjoint sets, such that the elements in one set are all smaller than those in the other set;
- Insert, which inserts a new single element into a given set;
- Find, which reports the name of the set that a given element belongs to.

Recall our description of Weiner's algorithm in Section 3.1 and see Figure 2. Consider the set of in-coming Weiner links of node u before updates (the left diagram of Figure 2), and assume that these Weiner links are sorted by the length of the origin nodes. After arriving at the node v in the climbing up process from the leaf for T, we take the Weiner link with character a and arrive at node u. Then we access the set of in-coming Weiner-links of u by a find query. When we create a new internal node y as the parent of the new leaf for aT, we split this set into two sets, one as the set of in-coming Weiner links of y, and the other as the set of in-coming Weiner links of u (see the right diagram of Figure 2). This can be maintained by a single call of a split operation.

Now we pay our attention to the copying process of Weiner links described in Figure 3. Observe that each newly copied Weiner link can be inserted by a single find operation and a single insert operation into the set of in-coming Weiner links of $\mathsf{W_link}_u(c)$ for each character c where $\mathsf{W_link}_u(c)$ is defined.

Now we prove the next lemma:

Lemma 1. *Let f denote the operation and query time of a linear-space algorithm for the ordered split-insert-find problem. Then, we can build the suffix tree for a fully-online right-to-left string collection of total length N in a total of $O(N(f + \log \sigma))$ time and $O(N)$ space.*

Proof. The number of split operations is clearly bounded by the number of leaves, which is N. Since the number of Weiner links is at most $3N - 4$, the number of insert operations is also bounded by $3N - 4$. The number of find queries is thus bounded by $N + 3N - 4 = 4N - 4$. By using a linear-space split-insert-find data structure, we

can maintain the set of in-coming Weiner links for all nodes in a total of $O(Nf)$ time with $O(N)$ space.

Given a new character a to prepend to a string T, we climb up the path from the leaf for T and find the deepest ancestor v of the leaf for which $\mathsf{W_link}_a(v)$ is defined. This can be checked in $O(\log \sigma)$ time at each visited node, by using a balanced search tree. Since we do not climb up the nodes z (see Figure 2) for which the soft Weiner links with a are redirected, we can use the same analysis (a) as in the case of a single string. This results in that the number of visited nodes in our algorithm is $O(N)$. Hence we use $O(N \log \sigma)$ total time for finding the deepest node which has a Weiner link for the prepended character a.

Overall, our algorithm uses $O(N(f + \log \sigma))$ time and $O(N)$ space. □

Our ordered split-insert-find problem is a special case of the union-split-find problem on ordered sets, since each insert operation can be trivially simulated by make-set and union operations. Link-cut trees of Sleator and Tarjan [15] for a dynamic forest support make-tree, link, cut operations and find-root queries in $O(\log d)$ time each. Since link-cut trees can be used to path-trees, make-set, insert, split, and find in the ordered split-insert-find problem can be supported in $O(\log d)$ time each. Since link-cut trees work on the pointer machine model, this leads to a pointer-machine algorithm for our fully-online right-to-left construction of the suffix tree for multiple strings with $f = O(\log d)$. Here, in our context, d denotes the maximum number of in-coming Weiner links to a node of the suffix tree.

A potential drawback of using link-cut trees is that in order to achieve $O(\log d)$-time operations and queries, link-cut trees use some auxiliary data structures such as splay trees [16] as its building block. Yet, in what follows, we show that our ordered split-insert-find problem can be solved by a simpler balanced tree, AVL-trees [1], retaining $O(N(\log \sigma + \log d))$-time and $O(N)$-space complexities.

Theorem 2. *There is an AVL-tree based pointer-machine algorithm that builds the suffix tree for a fully-online right-to-left multiple strings of total length N in $O(N(\log \sigma + \log d))$ time with $O(N)$ space, where d is the maximum number of in-coming Weiner links to a suffix tree node and σ is the alphabet size.*

Proof. For each node u of the suffix tree $\mathsf{STree}(\mathcal{T}')$ before update, let $S(u) = \{|x| \mid \mathsf{W_link}_a(x) = u\}$ where $a = u[1]$, namely, $S(u)$ is the set of the string depths of the origin nodes of the in-coming Weiner links of u. We maintain an AVL tree for $S(u)$ with the node u, so that each in-coming Weiner link for u points to the corresponding node in the AVL tree for $S(u)$. The root of the AVL tree is always linked to the suffix tree node u, and each time another node in the AVL tree becomes the new root as a result of node rotations, we adjust the link so that it points to u from the new root of the AVL tree.

This way, a find query for a given Weiner link is reduced to accessing the root of the AVL tree that contains the given Weiner link, which can be done in $O(\log S(u)) \subseteq O(\log d)$ time.

Inserting a new element to $S(u)$ can also be done in $O(\log S(u)) \subseteq O(\log d)$ time.

Given an integer k, let S_1 and S_2 denote the subset of $S(u)$ such that any element in S_1 is not larger than k, any element in S_2 is larger than k, and $S_1 \cup S_2 = S(u)$. It is well known that we can split the AVL tree for $S(u)$ into two AVL trees for S_1 and for S_2 in $O(\log S(u)) \subseteq O(\log d)$ time (c.f. [13]). In our context, k is the string depth of the deepest node v that is a Weiner link with character a in the upward path from

the leaf for T. This allows us to maintain $S_1 = S(y)$ and $S_2 = S(u)$ in $O(\log d)$ time in the updated suffix tree $\mathsf{STree}(\mathcal{T})$.

When we create the in-coming Weiner links labeled a to the new leaf $\hat{\ell}$ for aT, we first perform a make-set operation which builds an AVL tree consisting only of the root. If we naïvely insert each in-coming Weiner link to the AVL tree one by one, then it takes a total of $O(N \log d)$ time. However, we can actually perform this process in $O(N)$ total time even on the pointer machine model: Since we climb up the path from the leaf ℓ for T, the in-coming Weiner links are already sorted in decreasing order of the string depths of the origin nodes. We create a family of maximal complete binary trees of size $2^h - 1$ each, arranged in decreasing order of h. This can be done as follows: Initially set $r \leftarrow |S(\hat{\ell})|$. We then greedily take the largest h such that $2^h - 1 \leq r$, and then update $r \leftarrow r - (2^h - 1)$ and search for the next largest h and so on. These trees can be easily created in $O(|S(\hat{\ell})|)$ total time by a simple linear scan over the sorted list of the in-coming Weiner links. Since the heights h of these complete binary search trees are monotonically decreasing, and since all of these binary search trees are AVL trees, one can merge all of them into a single AVL tree in time linear in the height of the final AVL tree (c.f. [13]), which is bounded by $O(h) = O(\log S(\hat{\ell}))$. Thus, we can construct the initial AVL tree for the in-coming Weiner links of each new leaf $\hat{\ell}$ in $O(|S(\hat{\ell})|)$ time. Since the total number of Weiner links is $O(N)$, we can construct the initial AVL trees for the in-coming Weiner links of all new leaves in $O(N)$ total time.

Overall, our algorithm works in $O(N(\log \sigma + \log d))$ time with $O(N)$ space. □

4.2 Left-to-right DAWG construction

The next theorem immediately follows from Theorem 2.

Theorem 3. *There is an AVL-tree based pointer-machine algorithm that builds an explicit representation of the DAWG for a fully-online left-to-right multiple strings of total length N in $O(N(\log \sigma + \log d))$ time with $O(N)$ space, where d is the maximum number of in-coming edges of a DAWG node and σ is the alphabet size. This representation of the DAWG allows each edge traversal in $O(\log \sigma + \log d)$ time.*

Proof. The correctness and the complexity of construction are immediate from Theorem 2.

Given a character a and a node v in the DAWG, we first find the out-going edge of v labeled a in $O(\log \sigma)$ time. If it does not exist, we terminate. Otherwise, we take this a-edge and arrive at the corresponding node in the AVL tree for the destination node u for this a-edge. We then perform a find query on the AVL tree and obtain u in $O(\log d)$ time. □

We emphasize that Theorem 3 gives the *first* non-trivial algorithm that builds an explicit representation of the DAWG for fully-online multiple strings. Recall that a direct application of Blumer et al.'s algorithm to the case of fully-online K multiple strings requires to visit $\Theta(N \min(K, \sqrt{N}))$ nodes in the DAWG, which leads to $O(N \min(K, \sqrt{N}) \log \sigma) = O(N^{1.5} \log \sigma)$-time construction for $K = \Theta(\sqrt{N})$.

It should be noted that after all the N characters have been processed, it is easy to modify, in $O(N)$ time in an offline manner, this representation of the DAWG so that each edge traversal takes $O(\log \sigma)$ time.

4.3 On optimality of our algorithms

It is known that sorting a length-N sequence of σ distinct characters is an obvious lower bound for building the suffix tree [7] or alternatively the DAWG. This is because, when we build the suffix tree or the DAWG where the out-going edges of each node are sorted in the lexicographical order, then we can obtain a sorted list of characters at their root. Thus, $\Omega(N \log \sigma)$ is a comparison-based model lower bound for building the suffix tree or the DAWG. Since Takagi et al.'s $O(N \log \sigma)$-time algorithm [17] works only on the word RAM model, in which faster integer sorting algorithms exist, it would be interesting to explore some cases where our $O(N(\log \sigma + \log d))$-time algorithms for a weaker model of computation can perform in optimal $O(N \log \sigma)$ time.

It is clear that the maximum number d of in-coming Weiner links to a node is bounded by the total length N of the strings. Hence, in case of integer alphabets of size $\sigma = N^{O(1)}$, our algorithms run in optimal $O(N \log \sigma) = O(N \log N)$ time.

For the case of smaller alphabet size $\sigma = \mathrm{polylog}(N)$, the next lemma can be useful:

Lemma 4. *The maximum number d of in-coming Weiner links is less than the height of the suffix tree.*

Proof. For any node u in the suffix tree, all in-coming Weiner links to u is labeled by the same character a, which is the first character of the substring represented by u. Therefore, all in-coming Weiner links to u are from the nodes in the path between the root and the node $u[2..|u|]$. □

We note that the height of the suffix tree for multiple strings is bounded by the length of the longest string in the collection. In many applications such as time series from sensor data, it would be natural to assume that all the K strings in the collection have similar lengths. Hence, when the collection consists of $K = N/\mathrm{polylog}(N)$ strings of length $\mathrm{polylog}(N)$ each, we have $d = \mathrm{polylog}(N)$. In such cases, our algorithms run in optimal $O(N \log \sigma) = O(N \log \log N)$ time.

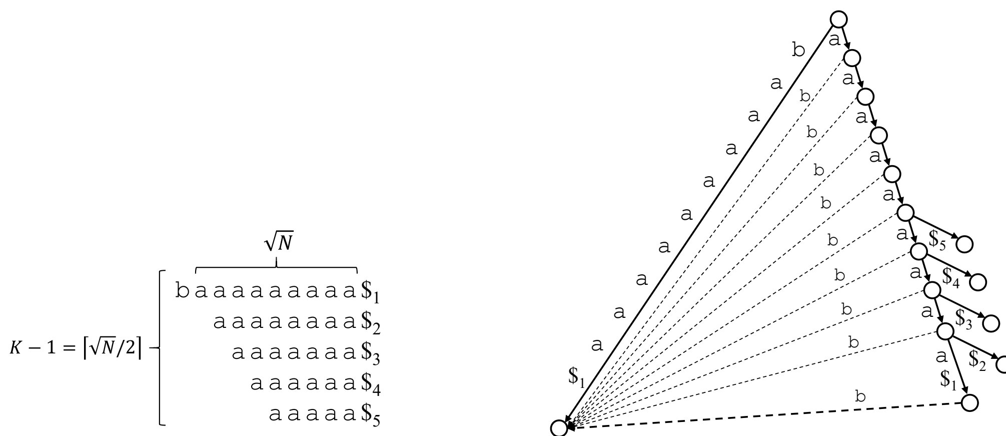

Figure 4. Left: The $K-1 = \lceil \sqrt{N}/2 \rceil$ strings where character b has been prepended only to the first string T_1. Right: The corresponding part of the suffix tree. Dashed arrows represent Weiner links with character b.

The next lemma shows some instance over a binary alphabet of size $\sigma = 2$, which requires a certain amount of work for the splitting process.

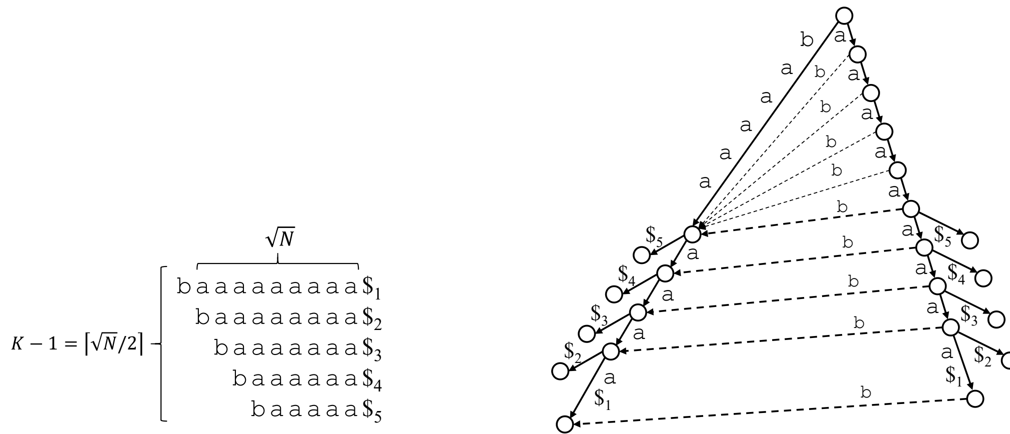

Figure 5. Left: The $K-1 = \lceil\sqrt{N}/2\rceil$ strings where character $\mathtt{b}$ has been prepended to all of them. Right: The corresponding part of the suffix tree after the updates. Each time a new leaf is created, $\Theta(\sqrt{N})$ in-coming Weiner links were involved in a split operation on the AVL tree and it takes $O(\log N)$ time.

Lemma 5. *There exist a set of fully-online multiple strings over a binary alphabet such that the node split procedure of our algorithms takes $O(\sqrt{N}\log N)$ time.*

Proof. Let $K = 1 + \lceil\sqrt{N}/2\rceil$.

For the time being, we assume that each string T_i is terminated with a unique symbol $\$_i$. Consider a subset $\{T_1, \ldots, T_{K-1}\}$ of $K-1 = \lceil\sqrt{N}/2\rceil$ strings such that for each $1 \leq i \leq K-1$, $T_i = \mathtt{a}^{\sqrt{N}-i+1}\$_i$. We then prepend the other character $\mathtt{b}$ from the binary alphabet $\{\mathtt{a}, \mathtt{b}\}$ to each T_i in increasing order of $i = 1, \ldots, K-1$. For $i = 1$, $\sqrt{N}$ Weiner links to the new leaf for $\mathtt{b}T_1 = \mathtt{ba}^{\sqrt{N}}\$_1$, each labeled $\mathtt{b}$, are created. See Figure 4 for illustration of this step.

Then, for each $i = 2, \ldots, K-1$, inserting a new leaf for $\mathtt{b}T_i$ requires an insertion of a new internal node as the parent of the new leaf. This splits the set of in-coming Weiner links into two sets: one is a singleton consisting of the Winer link from node $\mathtt{a}^{\sqrt{N}-i+1}$, and the other consists of the Weiner links from the shallower nodes. Each of these $K-2$ split operations can be done by a simple deletion operation on the corresponding AVL tree, using $O(\log\sqrt{N}) = O(\log N)$ time each. See Figure 5 for illustration.

Observe also that the same analysis holds even if we remove the terminal symbol $\$_i$ from each string T_i (in this case, there is a non-branching internal node for each T_i and we start the climbing up process from this internal node).

The total length of these $K-1$ strings is approximately $3N/8$. We can arbitrarily choose the last string T_K of length approximately $5N/8$ so that it does not affect the above split operations (e.g., a unary string $a^{5N/8}$ or $b^{5N/8}$ would suffice).

Thus, there exists an instance over a binary alphabet for which the node split operations require $O(\sqrt{N}\log N)$ total time. □

Since $\sqrt{N}\log N = o(N)$, the $\sqrt{N}\log N$ term is always dominated by the $N\log\sigma$ term. It is left open whether there exists a set of strings with $\Theta(N)$ character additions, each of which requires splitting a set that involves $N^{O(1)}$ in-coming Weiner links. If such an instance exists, then our algorithm must take $\Theta(N\log N)$ time in the worst case.

5 Conclusions and future work

In this paper we considered the problem of maintaining the suffix tree and the DAWG indexing structures for a collection of multiple strings that are updated in a fully-online manner, where a new character can be added to the left end or the right end of any string in the collection, respectively. Our contributions are simple pointer-machine algorithms that work in $O(N(\log\sigma + \log d))$ time and $O(N)$ space, where N is the total length of the strings, σ is the alphabet size, and d is the maximum number of in-coming Weiner links of a node in the suffix tree. The key idea was to reduce the sub-problem of re-directing in-coming Weiner links to the ordered split-insert-find problem, which we solved in $O(\log d)$ time by AVL trees. We also discussed the cases where our $O(N(\log\sigma + \log d))$-time solution is optimal.

A major open question regarding the proposed algorithms is whether there exists an instance over a small alphabet which contains $\Theta(N)$ positions each of which requires $\Theta(\log N)$ time for the split operation, or requires $\Theta(N)$ insertions each taking $\Theta(\log N)$ time. If such instances exist, then the running time of our algorithms may be worse than the optimal $O(N\log\sigma)$ for small σ. So far, we have only found an instance with $\sigma = 2$ that takes sub-linear $O(\sqrt{N}\log N)$ total time for split operations.

Acknowledgements

This work is supported by JSPS KAKENHI Grant Number JP17H01697 and JST PRESTO Grant Number JPMJPR1922.

References

1. G. Adelson-Velsky and E. Landis: *An algorithm for the organization of information.* Proceedings of the USSR Academy of Sciences (in Russian), 146 1962, pp. 263–266, English translation by Myron J. Ricci in Soviet Mathematics - Doklady, 3:1259–1263, 1962.
2. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas: *The smallest automaton recognizing the subwords of a text.* Theoretical Computer Science, 40 1985, pp. 31–55.
3. A. Blumer, J. Blumer, D. Haussler, R. Mcconnell, and A. Ehrenfeucht: *Complete inverted files for efficient text retrieval and analysis.* J. ACM, 34(3) 1987, pp. 578–595.
4. M. T. Chen and J. Seiferas: *Efficient and elegant subword-tree construction*, in Combinatorial Algorithms on Words, vol. 12 of NATO ASI Series, 1985, pp. 97–107.
5. M. Crochemore and W. Rytter: *Text Algorithms*, Oxford University Press, 1994.
6. H. H. Do and W. Sung: *Compressed directed acyclic word graph with application in local alignment.* Algorithmica, 67(2) 2013, pp. 125–141.
7. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan: *On the sorting-complexity of suffix tree construction.* J. ACM, 47(6) 2000, pp. 987–1011.
8. Y. Fujishige, Y. Tsujimaru, S. Inenaga, H. Bannai, and M. Takeda: *Computing DAWGs and minimal absent words in linear time for integer alphabets*, in MFCS 2016, 2016, pp. 38:1–38:14.
9. H. N. Gabow and R. E. Tarjan: *A linear-time algorithm for a special case of disjoint set union.* Journal of Computer and System Sciences, 30 1985, pp. 209–221.
10. D. Gusfield and J. Stoye: *Linear time algorithms for finding and representing all the tandem repeats in a string.* J. Comput. Syst. Sci., 69(4) 2004, pp. 525–546.
11. D. Hendrian, S. Inenaga, R. Yoshinaka, and A. Shinohara: *Efficient dynamic dictionary matching with DAWGs and AC-automata.* Theor. Comput. Sci., 792 2019, pp. 161–172.
12. H. Imai and T. Asano: *Dynamic orthogonal segment intersection search.* J. Algorithms, 8(1) 1987, pp. 1–18.

13. D. Knuth: *The Art Of Computer Programming, vol. 3: Sorting And Searching, Second Edition*, Addison-Wesley, 1998.
14. G. Kucherov and M. Rusinowitch: *Matching a set of strings with variable length don't cares*. Theor. Comput. Sci., 178(1-2) 1997, pp. 129–154.
15. D. D. Sleator and R. E. Tarjan: *A data structure for dynamic trees*. J. Comput. Syst. Sci., 26(3) 1983, pp. 362–391.
16. D. D. Sleator and R. E. Tarjan: *Self-adjusting binary search trees*. J. ACM, 32(3) 1985, pp. 652–686.
17. T. Takagi, S. Inenaga, H. Arimura, D. Breslauer, and D. Hendrian: *Fully-online suffix tree and directed acyclic word graph construction for multiple texts*. Algorithmica, 82(5) 2020, pp. 1346–1377.
18. Y. Tanimura, Y. Fujishige, T. I, S. Inenaga, H. Bannai, and M. Takeda: *A faster algorithm for computing maximal α-gapped repeats in a string*, in SPIRE 2015, 2015, pp. 124–136.
19. R. E. Tarjan: *A class of algorithms which require nonlinear time to maintain disjoint sets*. J. Comput. Syst. Sci., 18(2) 1979, pp. 110–127.
20. R. E. Tarjan: *Data structures and network algorithms*, vol. 44 of CBMS-NSF regional conference series in applied mathematics, SIAM, 1983.
21. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.
22. J. Westbrook: *Fast incremental planarity testing*, in ICALP 1992, 1992, pp. 342–353.
23. J. Yamamoto, T. I, H. Bannai, S. Inenaga, and M. Takeda: *Faster compact on-line Lempel-Ziv factorization*, in STACS 2014, 2014, pp. 675–686.