Space-Efficient Online Computation of String Net Occurrences

Takuya Mieno
Department of Computer and Network Engineering, University of Electro-Communications, Chofu, Japan

Shunsuke Inenaga
Department of Informatics, Kyushu University, Fukuoka, Japan

Abstract

A substring u of a string T is said to be a *repeat* if u occurs at least twice in T. An occurrence $[i..j]$ of a repeat u in T is said to be a *net occurrence* if each of the substrings $aub = T[i-1..j+1]$, $au = T[i-1..j]$, and $ub = T[i..j+1]$ occurs exactly once in T. The occurrence $[i-1..j+1]$ of aub is said to be an *extended* net occurrence of u. Let T be an input string of length n over an alphabet of size σ, and let $\mathsf{ENO}(T)$ denote the set of extended net occurrences of repeats in T. Guo et al. [SPIRE 2024] presented an online algorithm which can report $\mathsf{ENO}(T[1..i])$ in $T[1..i]$ in $O(n\sigma^2)$ time, for each prefix $T[1..i]$ of T. Very recently, Inenaga [arXiv 2024] gave a faster online algorithm that can report $\mathsf{ENO}(T[1..i])$ in optimal $O(\#\mathsf{ENO}(T[1..i]))$ time for each prefix $T[1..i]$ of T, where $\#S$ denotes the cardinality of a set S. Both of the aforementioned data structures can be maintained in $O(n \log \sigma)$ time and occupy $O(n)$ space, where the $O(n)$-space requirement comes from the suffix tree data structure. In particular, Inenaga's recent algorithm is based on Weiner's right-to-left online suffix tree construction. In this paper, we show that one can modify Ukkonen's left-to-right online suffix tree construction algorithm in $O(n)$ space, so that $\mathsf{ENO}(T[1..i])$ can be reported in optimal $O(\#\mathsf{ENO}(T[1..i]))$ time for each prefix $T[1..i]$ of T. This is an improvement over Guo et al.'s method that is also based on Ukkonen's algorithm. Further, this leads us to the two following space-efficient alternatives:

- A *sliding-window* algorithm of $O(d)$ working space that can report $\mathsf{ENO}(T[i-d+1..i])$ in optimal $O(\#\mathsf{ENO}(T[i-d+1..i]))$ time for each sliding window $T[i-d+1..i]$ of size d in T.
- A *CDAWG*-based online algorithm of $O(\mathsf{e})$ working space that can report $\mathsf{ENO}(T[1..i])$ in optimal $O(\#\mathsf{ENO}(T[1..i]))$ time for each prefix $T[1..i]$ of T, where $\mathsf{e} < 2n$ is the number of edges in the CDAWG for T.

All of our proposed data structures can be maintained in $O(n \log \sigma)$ time for the input online string T. We also discuss that the extended net occurrences of repeats in T can be fully characterized in terms of the *minimal unique substrings* (*MUSs*) in T.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases string net occurrences, suffix trees, CDAWGs, maximal repeats, minimal unique substrings (MUSs)

Digital Object Identifier 10.4230/LIPIcs.CPM.2025.23

Funding *Takuya Mieno*: JSPS KAKENHI Grant Number JP24K20734.
Shunsuke Inenaga: JSPS KAKENHI Grant Numbers JP23K24808, JP23K18466, JP20H05964.

1 Introduction

Finding *repeats* in a string is a fundamental task of string processing that has applications in various fields including bioinformatics, data compression, and natural language processing. This paper focuses on the notion of *net occurrences* of a repeat in a string, which has attracted recent attention. Let u be a repeat in a string T such that u occurs at least twice in T. An occurrence $[i..j]$ of a repeat u in T is said to be a net occurrence of u if extending the occurrence to the left or to the right results in a unique occurrence, i.e., each

of $aub = T[i-1..j+1]$, $au = T[i-1..j]$, and $ub = T[i..j+1]$ occurs exactly once in T. Finding string net occurrences are motivated for Chinese language text processing [13, 14]. The occurrence $[i-1..j+1]$ of aub is said to be an *extended net occurrence* of a repeat u in T, and let $\mathsf{ENO}(T)$ denote the set of all extended net occurrences of repeats in T.

Guo et al. [5] were the first who considered the problem of computing (extended) net occurrences of repeats in a string from view points of string combinatorics and algorithmics. Guo et al. [5] showed a necessary and sufficient condition for a net occurrence of a repeat, which, since then, has played a core role in efficient computation of string net occurrences. For an input string T of length n, they gave an *offline* algorithm for computing $\mathsf{ENO}(T)$ in $O(n)$ time and space for integer alphabets of size polynomial in n, and in $O(n \log \sigma)$ time and $O(n)$ space for general ordered alphabets of size σ. Their offline method is based on the suffix array [15] and the Burrows-Wheeler transform [4]. Ohlebusch et al. gave another offline algorithm that works fast in practice [18].

Later, Guo et al. [6] proposed an *online* algorithm for computing all string net occurrences of repeats. Their algorithm maintains a data structure of $O(n)$ space that reports $\mathsf{ENO}(T[1..i])$ in $O(n\sigma^2)$ time for each prefix $T[1..i]$ of an online input string T of length n. Since their algorithm computes all the net occurrences upon a query, their algorithm requires at least $O(n^2\sigma^2)$ time to *maintain and update* the list of all (extended) net occurrences of repeats in an online string. Their algorithm is based on Ukkonen's left-to-right online suffix tree construction [22], that is enhanced with the suffix-extension data structure of Breslauer and Italiano [3].

Very recently, Inenaga [8] proposed a faster algorithm that can maintain $\mathsf{ENO}(T[1..i])$ for an online string $T[1..i]$ with growing $i = 1, ..., n$ in a total of $O(n \log \sigma)$ time and $O(n)$ space. Namely, this algorithm uses only amortized $O(\log \sigma)$ time to update $\mathsf{ENO}(T[1..i])$ to $\mathsf{ENO}(T[1..i+1])$. The proposed algorithm is based on Weiner's right-to-left online suffix tree construction [23] that is applied to the reversed input string, and can report all extended net occurrences of repeats in $T[1..i]$ in optimal $O(\#\mathsf{ENO}(T[1..i]))$ for each $1 \leq i \leq n$, where $\#S$ denotes the cardinality of a set S.

In this paper, we first show that Ukkonen's left-to-right online suffix tree construction algorithm can also be modified so that it can maintain and update $\mathsf{ENO}(T[1..i])$ in a total of $O(n \log \sigma)$ time with $O(n)$ space, and can report $\mathsf{ENO}(T[1..i])$ in optimal $O(\#\mathsf{ENO}(T[1..i]))$ time for each $i = 1, \ldots, n$. While this complexity of our Ukkonen-based method is the same as the previous Weiner-based method [8], our method enjoys the following merits:

(1) Our result shows that the arguably complicated suffix-extension data structure of Breslauer and Italiano is not necessary for online computation of string net occurrences with Ukkonen's algorithm.

(2) The new method can be extended to the *sliding suffix trees* [11, 21, 12] and the *compact directed acyclic word graphs* (*CDAWGs*) [2, 9].

The first point is a simplification and improvement over Guo et al.'s method [6] based on Ukkonen's construction. The second point leads us to the following space-efficient alternatives:

- A sliding-window algorithm of $O(d)$ working space that can be maintained in $O(n \log \sigma)$ time and can report $\mathsf{ENO}(T[i-d+1..i])$ in optimal $O(\#\mathsf{ENO}(T[i-d+1..i]))$ time for each sliding window $T[i-d+1..i]$ of size d in T.
- A CDAWG-based online algorithm of $O(\mathsf{e})$ working space that can be maintained in $O(n \log \sigma)$ time and can report $\mathsf{ENO}(T[1..i])$ in optimal $O(\#\mathsf{ENO}(T[1..i]))$ time for each prefix $T[1..i]$ of T, where e is the number of edges in the CDAWG for T.

We note that $\mathsf{e} < 2n$ always holds [2], and e can be as small as $O(\log n)$ for some highly repetitive strings [20, 19]. Finally, we also discuss that the extended net occurrences of repeats in a string T can be fully characterized with the *minimal unique substrings* (*MUSs*) [7] in T.

2 Preliminaries

Strings

Let Σ be an alphabet of size σ. An element of Σ is called a character. An element of $\Sigma^\star$ is called a string. The empty string ε is the string of length 0. If $T = pfs$ holds for strings T, p, f, and s, then p, f, and s are called a prefix of T, a substring of T, and a suffix of T, respectively. A prefix p (resp. a suffix s) of T is called a *proper* prefix (resp. a *proper* suffix) of T if $p \neq T$ (resp. $s \neq T$). For a string T, $|T|$ denotes the length of T. For a string T and an integer i with $1 \leq i \leq |T|$, $T[i]$ denotes the ith character of T. For a string T and integers i, j with $1 \leq i \leq j \leq |T|$, $T[i..j]$ denotes the substring of T starting at position i and ending at position j. For strings T and w, we say w *occurs* in T if $T[i..j] = w$ holds for some i, j. Also, if such i, j exist, we denote by $[i..j]$ the *occurrence* of $w = T[i..j]$ in T. Also, we denote by $occ_T(w)$ the set of occurrences of w in T, i.e., $occ_T(w) = \{[i..j] \mid T[i..j] = w\}$. For any set S, we denote by $\#S$ the cardinality of S. For convenience, we assume that the empty string ε occurs $|T| - 1$ times at the boundaries of consecutive characters in T, and denote these inner occurrences by $T[i+1..i] = \varepsilon$ for $1 \leq i < n$. We also assume that ε occurs before the first character and after the last character of T. Thus we have $\#occ_T(\varepsilon) = |T| + 1$. A string w is said to be *unique* in T if $\#occ_T(w) = 1$. Also, w is said to be *quasi-unique* in T if $1 \leq \#occ_T(w) \leq 2$. Further, w is said to be *repeating* in T if $\#occ_T(w) \geq 2$. We denote by $lrSuf(T)$ (resp. $lrPref(T)$) the longest repeating suffix (resp. prefix) of T. We denote by $sqSuf(T)$ (resp. $sqPref(T)$) the shortest quasi-unique suffix (resp. prefix) of T.

Maximal repeats and minimal unique substrings

A repeating substring of a string T is also called a *repeat* in T. A repeat u in T is said to be *left-branching* in T if there are at least two distinct characters $a, a' \in \Sigma$ such that $\#occ_T(au) \geq 1$ and $\#occ_T(a'u) \geq 1$. Symmetrically, a repeat u is said to be *right-branching* in T if there are at least two distinct characters $b, b' \in \Sigma$ such that $\#occ_T(ub) \geq 1$ and $\#occ_T(ub') \geq 1$. A repeat u of T is said to be *left-maximal* in T if u is a left-branching repeat of T or u is a prefix of T, and u is said to be *right-maximal* in T if u is a right-branching repeat in T or u is a suffix of T. A repeat u of T is said to be *maximal* in T if u is both a left-maximal repeat and a right-maximal repeat in T. Let $\mathsf{LB}(T)$, $\mathsf{RB}(T)$, $\mathsf{LM}(T)$, $\mathsf{RM}(T)$, and $\mathsf{M}(T)$ denote the sets of left-branching, right-branching, left-maximal, right-maximal, and maximal repeats in T, respectively. Note that $\mathsf{LB}(T) \subseteq \mathsf{LM}(T)$, $\mathsf{RB}(T) \subseteq \mathsf{RM}(T)$, and $\mathsf{M}(T) = \mathsf{LM}(T) \cap \mathsf{RM}(T)$ hold. A unique substring $u = T[i..j]$ of a string T is said to be a *minimal unique substring* (*MUS*) of T if each of the substrings $T[i+1..j]$ and $T[i..j-1]$ is a repeat in T. By definition, no MUS can be completely contained in another MUS in the string, and thus, there are at most n MUSs in any string T of length n. Let $\mathsf{MUS}(T)$ denote the set of occurrences of all MUSs in T.

In what follows, we fix a string T of length $n > 2$ arbitrarily.

(Extended) net occurrences

For a repeating substring P of T, its occurrence $[i..j]$ in T is said to be a *net occurrence* of P if $T[i-1..j]$ is unique in T, and $T[i..j+1]$ is unique in T. Let $\mathsf{NO}(T)$ be the set of net occurrences in T. We call $T[i-1..j+1]$ a net unique substring (NUS) if $[i..j] \in \mathsf{NO}(T)$. Let $\mathsf{NUS}(T)$ be the set of net unique substrings in T. Then, we call the occurrence $[i-1..j+1]$ of NUS $T[i-1..j+1]$ the *extended net occurrence* of the repeat $T[i..j]$. Let $\mathsf{ENO}(T)$ be the set of extended net occurrences in T. Clearly, there is a one-to-one correspondence between $\mathsf{NUS}(T)$ and $\mathsf{ENO}(T)$, i.e., $T[p..q] \in \mathsf{NUS}(T)$ iff $[p..q] \in \mathsf{ENO}(T)$. Note that $\#\mathsf{ENO}(T) = \#\mathsf{NO}(T)$ holds.

Data structures

The *suffix tree* [23] of a string T is a compacted trie that represents all suffixes of T. More formally, the suffix tree of T is a rooted tree such that (1) each edge is labeled by a non-empty substring of T, (2) the labels of the out-edges of the same node begin with distinct characters, and (3) every suffix of T is represented by a path from the root. If the path from the root to a node v spells out the substring w of T, then we say that the node v *represents* w. By representing each edge label x with a pair (i, j) of positions such that $x = T[i..j]$, the suffix tree can be stored in $O(n)$ space. For convenience, we identify each node with the string that the node represents. If av is a node of the suffix tree with $a \in \Sigma$ and $v \in \Sigma^\star$, then the *suffix link* of the node av points to the node v.

There are two versions of suffix trees, *implicit suffix trees* [22] (a.k.a. *Ukkonen trees*) and *explicit suffix trees* [23] (a.k.a *Weiner trees*). In the implicit suffix tree of string T, each repeating suffix s of T that has a unique right-extension $c \in \Sigma$ (namely, $\#occ_T(sc) \geq 1$ and $\#occ_T(sa) = 0$ for any $a \in \Sigma \setminus \{c\}$) is represented on an edge. On the other hand, each such repeating suffix is represented by a non-branching internal node in the explicit suffix tree of T. Let $\mathsf{STree}'(T)$ and $\mathsf{STree}(T)$ denote the implicit suffix tree and the explicit suffix tree of string T, respectively. The internal nodes of $\mathsf{STree}'(w)$ represent the right-branching repeats of T, while the internal nodes of $\mathsf{STree}(w)$ represent the right-maximal repeats of T. It is thus clear that $\mathsf{STree}'(T\$) = \mathsf{STree}(T\$)$ with a unique end-marker \$ that does not occur in T. Due to the nature of left-to-right online string processing, we will use the implicit suffix trees in our algorithms.

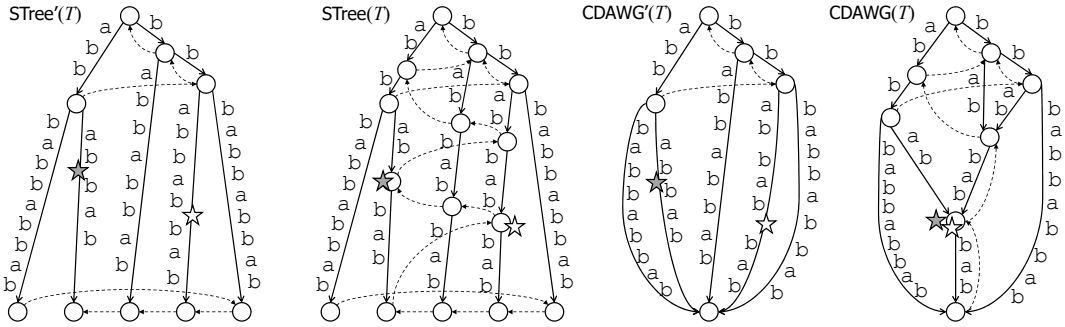

Figure 1 The implicit suffix tree $\mathsf{STree}'(T)$, the explicit suffix tree $\mathsf{STree}(T)$, the implicit CDAWG $\mathsf{CDAWG}'(T)$, and the explicit CDAWG $\mathsf{CDAWG}'(T)$ for string $T =$ `abbbabbabbab`. The broken arrows represent suffix links. The white and gray stars represent the loci of the longest repeating suffix `bbabbab` and shortest quasi-unique suffix `abbab` of T, respectively.

The *compact directed acyclic word graph* (*CDAWG*) [2] of a string T is the smallest edge-labeled DAG that represents all substrings of T. There are two versions of CDAWGs as well, *implicit CDAWGs* [9] and *explicit CDAWGs* [2]. The implicit CDAWG of string T, denoted $\mathsf{CDAWG}'(T)$, is the edge-labeled DAG that is obtained by merging all isomorphic subtrees of the implicit suffix tree $\mathsf{STree}'(T)$ which are connected by suffix links. On the other hand, the explicit CDAWG of T, denoted $\mathsf{CDAWG}(T)$, is the edge-labeled DAG that is obtained by merging all isomorphic subtrees of the explicit suffix tree $\mathsf{STree}(T)$ which are connected by suffix links. The internal nodes of $\mathsf{CDAWG}'(T)$ have a one-to-one correspondence with the left-maximal and right-branching repeats in T, while the internal nodes of $\mathsf{CDAWG}(T)$ have a one-to-one correspondence with the maximal repeats in T. More precisely, the longest string represented by an internal node in $\mathsf{CDAWG}'(T)$ is a left-maximal and right-branching repeat

in T, and the longest string represented by an internal node in $\mathsf{CDAWG}(T)$ is a maximal repeat in T. Thus, as in the case of suffix trees, $\mathsf{CDAWG}'(T\$) = \mathsf{CDAWG}(T\$)$ holds with a unique end-marker \$.

The length of a path in the implicit/explicit CDAWG is the total length of the labels of the edges in the path. An in-edge of a node v in the implicit/explicit CDAWG is said to be a *primary edge* if it belongs to the longest path from the source to v.

Due to the nature of left-to-right online string processing, we will use the implicit CDAWGs in our algorithm. Let $\mathsf{e}'(T)$ and $\mathsf{e}(T)$ denote the number of edges of $\mathsf{CDAWG}'(T)$ and $\mathsf{CDAWG}(T)$, respectively. The following lemma guarantees that the worst-case space complexity of our implicit-CDAWG based algorithm is linear in the size of explicit CDAWGs:

▶ **Lemma 1.** *For any string T, $\mathsf{e}'(T) \leq \mathsf{e}(T)$.*

Proof. Let V' and V be the sets of nodes of $\mathsf{CDAWG}'(T)$ and $\mathsf{CDAWG}(T)$, respectively. We identify each node of V' and of V with the longest string that the node represents. Then, $V' = \mathsf{LM}(T) \cap \mathsf{RB}(T)$ and $V = \mathsf{LM}(T) \cap \mathsf{RM}(T) = \mathsf{M}(T)$. Since $\mathsf{RB}(T) \subseteq \mathsf{RM}(T)$, we have that $V' \subseteq V$. Let $w \in \mathsf{LM}(T) \cap \mathsf{RB}(T)$, which implies that $w \in V'$ and $w \in V$. Let v' and v be the nodes that represent w in $\mathsf{CDAWG}'(T)$ and $\mathsf{CDAWG}(T)$, respectively. The out-degree $d(v')$ of node v' in $\mathsf{CDAWG}'(T)$, as well as the out-degree $d(v)$ of v in $\mathsf{CDAWG}(T)$, are equal to the number of right-extensions $c \in \Sigma$ such that $\#occ_T(wc) \geq 1$. Thus $\mathsf{e}'(T) = \sum_{v' \in V'} d(v') \leq \sum_{u \in V} d(u) = \mathsf{e}(T)$. ◀

3 Changes in net occurrences for online string

In this section, we show how the net unique substrings, equivalently the extended net occurrences in a string T can change when a character is appended. We will implicitly use the following fact throughout this section.

▶ **Fact 2.** *For any strings T, w, and character c, $\#occ_T(w) \leq \#occ_{Tc}(w)$ holds.*

Thus, if w is repeating in T, then it must be repeating in Tc. Further, if w is unique in Tc and is not a suffix of Tc, then w must be unique in T.

Next lemma characterizes net unique substrings to be deleted when a character c is appended to string T.

▶ **Lemma 3.** *Suppose that $aub \in \mathsf{NUS}(T) \setminus \mathsf{NUS}(Tc)$ is a net unique substring in T where $a, b, c \in \Sigma$ and $u \in \Sigma^\star$. Then, (i) $au = sqSuf(Tc)$ and $\#occ_{Tc}(au) = 2$ or (ii) $ub = lrSuf(Tc)$ and $\#occ_{Tc}(ub) = 2$ hold.*

Proof. Since $aub \in \mathsf{NUS}(T)$, au is unique in T, ub is unique in T, and u is repeating in T. Also, since $aub \notin \mathsf{NUS}(Tc)$, au or ub becomes repeating in Tc. See also Figure 2. Note that au and ub cannot be repeating in Tc, because if we assume they become repeating in Tc simultaneously, then both au and ub are suffixes of Tc and this implies $au = ub$, which contradicts that au is unique in T.

If au is repeating in Tc, then au is a suffix of Tc. Since au is unique in T, $\#occ_{Tc}(au) = 2$ holds. Also, since u is repeating in T, $\#occ_{Tc}(u) \geq 3$ holds. These imply that $au = sqSuf(Tc)$.

If ub is repeating in Tc, then ub is a suffix of Tc. Since ub is unique in T, $\#occ_{Tc}(ub) = 2$ holds. Now let $a' = T[|T| - |u|]$ be the preceding character of the suffix ub of Tc. If we assume that $a'ub$ is repeating in Tc, then the other occurrence $a'ub$ matches the occurrence of aub since ub is unique in T. This implies that $a' = a$, which contradicts au is unique in T. Thus, $ub = lrSuf(Tc)$. ◀

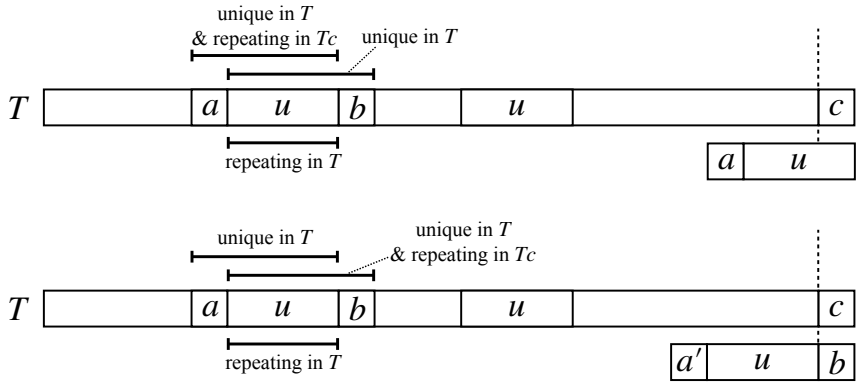

Figure 2 Illustration for Lemma 3. The upper part depicts the case where au is repeating in Tc. The lower part depicts the case where ub is repeating in Tc.

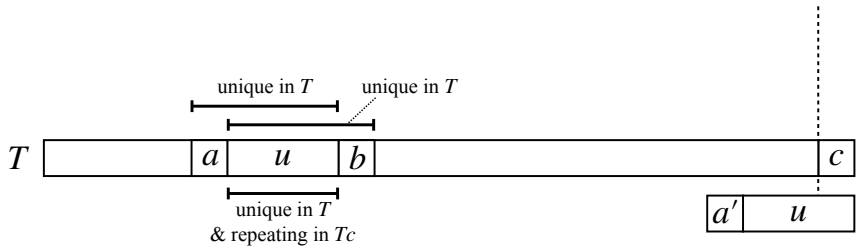

Figure 3 Illustration for Lemma 4. Suffix u of Tc is repeating in Tc but longer suffix $a'u$ of Tc is unique in Tc.

▶ **Lemma 4.** *Suppose that* $aub \in \mathsf{NUS}(Tc) \setminus \mathsf{NUS}(T)$ *is a net unique substring in* Tc *such that* aub *is not a suffix of* Tc *where* $a, b, c \in \Sigma$ *and* $u \in \Sigma^\star$. *Then,* $u = lrSuf(Tc)$ *and* $\#occ_{Tc}(u) = 2$ *hold.*

Proof. Since $aub \in \mathsf{NUS}(Tc)$, au is unique in Tc, ub is unique in Tc, and u is repeating in Tc. Also, since $aub \notin \mathsf{NUS}(T)$, u is unique in T. Namely, u occurs in Tc as a suffix and $\#occ_{Tc}(u) = 2$. See also Figure 3. Since au is unique in Tc, the preceding character a' of $u = T[|Tc| - |u| + 1..|Tc|]$ is not equal to a, and thus, $a'u$ is unique in Tc. Therefore, $u = lrSuf(Tc)$ holds. ◀

▶ **Lemma 5.** *Suppose that* $auc \in \mathsf{NUS}(Tc) \setminus \mathsf{NUS}(T)$ *is a net unique substring in* Tc *such that* auc *is a suffix of* Tc *where* $a, c \in \Sigma$ *and* $u \in \Sigma^\star$. *Then, either (i)* $u = lrSuf(T)$ *or (ii)* $u = c^{exp}$ *and* $\#occ_T(u) = 1$ *where* $exp = \max\{e \mid c^e$ *is a suffix of* $T\}$.

Proof. Since $auc \in \mathsf{NUS}(Tc)$, au is unique in Tc, uc is unique in Tc, and u is repeating in Tc. There are two cases with respect to the number of occurrences of u in T. See also Figure 4.

If u is repeating in T, $u = lrSuf(T)$ since au is unique in T. If u is unique in T, then appending c causes u to be repeating in T. This implies that $u = u[2..|u|]c$, i.e., $u = c^{|u|}$. Also, since $uc = c^{|u|+1}$ is unique in Tc, $a \neq c$ holds. Thus $|u| = exp = \max\{e \mid c^e$ is a suffix of $T\}$. ◀

As for the differences between $\mathsf{NUS}(T)$ and $\mathsf{NUS}(cT)$, the three following lemmas also hold by symmetry:

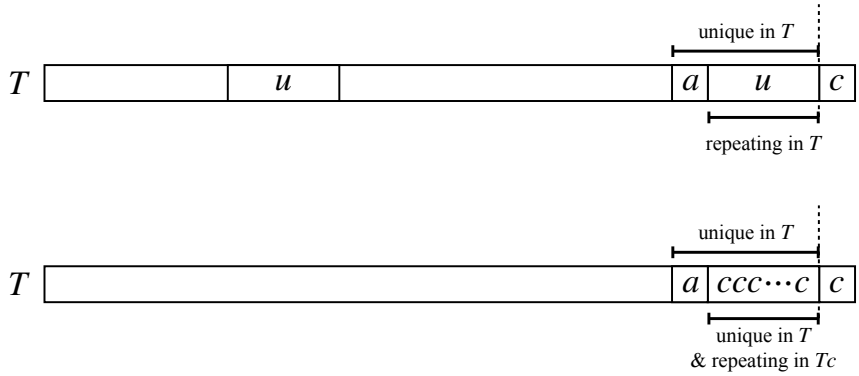

Figure 4 Illustration for Lemma 5. The upper part depicts the case where u is repeating in T. The lower part depicts the case where u is unique in T.

▶ **Lemma 6.** *Suppose that* $aub \in \mathsf{NUS}(T) \setminus \mathsf{NUS}(cT)$ *is a net unique substring in* T *where* $a, b, c \in \Sigma$ *and* $u \in \Sigma^\star$. *Then, (i)* $au = sqPref(cT)$ *and* $\#occ_{cT}(au) = 2$ *or (ii)* $ub = lrPref(cT)$ *and* $\#occ_{cT}(ub) = 2$ *hold.*

▶ **Lemma 7.** *Suppose that* $aub \in \mathsf{NUS}(cT) \setminus \mathsf{NUS}(T)$ *is a net unique substring in* cT *such that* aub *is not a prefix of* cT *where* $a, b, c \in \Sigma$ *and* $u \in \Sigma^\star$. *Then,* $u = lrPref(cT)$ *and* $\#occ_{cT}(u) = 2$ *hold.*

▶ **Lemma 8.** *Suppose that* $auc \in \mathsf{NUS}(cT) \setminus \mathsf{NUS}(T)$ *is a net unique substring in* cT *such that* auc *is a prefix of* cT *where* $a, c \in \Sigma$ *and* $u \in \Sigma^\star$. *Then, either (i)* $u = lrPref(T)$ *or (ii)* $u = c^{exp}$ *and* $\#occ_T(u) = 1$ *where* $exp = \max\{e \mid c^e$ *is a prefix of* $T\}$.

4 Algorithms

In this section, we present our online/sliding algorithms for computing extended net occurrences of repeats for a given string.

4.1 Online algorithm based on implicit suffix trees

By Lemmas 3, 4 and 5, we can compute $\mathsf{ENO}(Tc)$ from $\mathsf{ENO}(T)$ with Algorithm 1 since $[p..q] \in \mathsf{ENO}(T)$ iff $T[p..q] \in \mathsf{NUS}(T)$. We encode each element $[i..j] \in \mathsf{ENO}(T)$ by a pair (i, j) so that $\mathsf{ENO}(T)$ can be stored in $O(\#\mathsf{ENO}(T))$ space. Note that $\#\mathsf{ENO}(Tc) - \#\mathsf{ENO}(T) \neq -2$ while $\#(\mathsf{ENO}(Tc) \setminus \mathsf{ENO}(T))$ can be 2. See lines 9–15 of Algorithm 1. The size of E decreases by 1 if we enter line 12, however, the size increases by 1 at line 14. Thus $-1 \leq \#\mathsf{ENO}(Tc) - \#\mathsf{ENO}(T) \leq 2$.

From Algorithm 1, we obtain the following theorem:

▶ **Theorem 9.** *Given string* T *of length* n, $E = \mathsf{ENO}(T)$, *and character* c, *we can compute* $\mathsf{ENO}(Tc)$ *in* $O(t(n))$ *time using* $O(s(n))$ *space if each of the following operations can be executed in* $t(n)$ *time within* $s(n)$ *space:*

1. *Determine if* $\#occ_{Tc}(sqSuf(Tc)) = 2$.
2. *Find the non-suffix occurrence of* $sqSuf(Tc)$ *in* Tc *when* $\#occ_{Tc}(sqSuf(Tc)) = 2$.
3. *Determine if* $\#occ_{Tc}(lrSuf(Tc)) = 2$.
4. *Find the non-suffix occurrence of* $lrSuf(Tc)$ *in* Tc *when* $\#occ_{Tc}(lrSuf(Tc)) = 2$.
5. *Compute the lengths of* $lrSuf(T)$ *and* $lrSuf(Tc)$.

Algorithm 1 Updating extended net occurrences when a character is appended.

Require: $E = \mathsf{ENO}(T)$
Ensure: $E = \mathsf{ENO}(Tc)$
1: **procedure** AppendChar(String T, character c, set E)
2: **if** $\#occ_{Tc}(sqSuf(Tc)) = 2$ **then**
3: $(i, j) \leftarrow$ the non-suffix occurrence of $sqSuf(Tc)$ in Tc.
4: **if** $(i, j+1) \in E$ **then**
5: $E \leftarrow E \setminus \{(i, j+1)\}$ ▷ by (i) of Lemma 3
6: **end if**
7: **end if**
8: **if** $\#occ_{Tc}(lrSuf(Tc)) = 2$ **then**
9: $(i', j') \leftarrow$ the non-suffix occurrence of $lrSuf(Tc)$ in Tc.
10: **if** $i' > 1$ **then**
11: **if** $(i'-1, j') \in E$ **then**
12: $E \leftarrow E \setminus \{(i'-1, j')\}$ ▷ by (ii) of Lemma 3
13: **end if**
14: $E \leftarrow E \cup \{(i'-1, j'+1)\}$ ▷ by Lemma 4
15: **end if**
16: **end if**
17: **if** $|lrSuf(Tc)| \leq |lrSuf(T)|$ **then** ▷ $\iff \#occ_{Tc}(lrSuf(T)c) = 1$
18: $E \leftarrow E \cup \{(|T| - |lrSuf(T)|, |Tc|)\}$ ▷ by (i) of Lemma 5
19: **else if** $\#occ_T(c^{exp}) = 1$ where $exp = \max\{e \mid c^e \text{ is a suffix of } T\}$ **then**
20: $E \leftarrow E \cup \{(|T| - exp, |Tc|)\}$ ▷ by (ii) of Lemma 5
21: **end if**
22: **end procedure**

6. *Compute* $exp = \max\{e \mid c^e$ *is a suffix of* $T\}$ *and determine if* $\#occ_T(c^{exp}) = 1$.
7. *Determine if* $(i, j) \in E$ *for given pair* (i, j).
8. *Insert an element* (i, j) *into* E *for given pair* (i, j).
9. *Delete an element* (i, j) *from* E *if* $(i, j) \in E$ *for given pair* (i, j).

Proof. Look at Algorithm 1. Line 2 uses operation 1, Line 3 uses operation 2, Lines 4 and 11 use operation 7, Lines 5 and 12 use operation 9, Line 8 uses operation 3, Line 9 uses operation 4, Lines 14, 18, and 20 use operation 8, Line 17 uses operation 5, and Line 19 uses operation 6. Thus, Algorithm 1 consists of the above nine operations and basic arithmetic operations. ◀

Note that the correctness of Theorem 9 does not depend on the data structure used. The next lemma holds if we utilize the *implicit suffix tree* by Ukkonen [22].

▶ **Lemma 10.** *Based on Ukkonen's left-to-right online suffix tree construction [22], we can design a data structure* $\mathcal{D}_T$ *of size* $O(|T|)$ *that supports all nine operations of Theorem 9 in constant time. The data structure* $\mathcal{D}_T$ *can be updated to* $\mathcal{D}_{Tc}$ *in amortized* $O(\log \sigma)$ *time where* c *is a character.*

Proof. We employ an implicit suffix tree [22] of online string T enhanced with the *active point*, which represents $lrSuf(T)$, and the *secondary active point*, which represents $sqSuf(T)$ as in [16]. According to [16], such an enhanced implicit suffix tree can support operations 1–6 in $O(1)$ time. For the readers of this paper, we briefly explain how to perform those operations efficiently below:

- Operation 5 is obvious since we maintain the active point for every step.
- Operation 3 can be easily done in constant time by checking whether the active point locates on an edge towards a leaf or not.
- Operation 1 can be done in constant time by using the (secondary) active points (due to Lemma 1 of [16]).
- Operations 2 and 4 can be done by looking at the leaves under the (secondary) active points. For instance, look at the implicit suffix tree $\mathsf{STree}'(T)$ depicted in Figure 1. The secondary active point (the gray star), which represents the shortest quasi-unique suffix $s = \texttt{abbab}$, is on an edge towards a leaf. Further, the leaf under the secondary active point represents the suffix $\texttt{abbabbab}$ of T starting at position 5. Thus, s occurs at position 5, which is the non-suffix occurrence of s. Similarly, the non-suffix occurrence of the longest repeating suffix $\texttt{bbabbab}$ is position 3 since the leaf under the active point (the white star) represents the suffix of T starting at position 3.
- Value *exp* defined in operation 6 can be easily maintained independent of the suffix tree.

As for operations 7–9, we implement set $E = \mathsf{ENO}(T)$ as a set of occurrences where each element $(i, j) \in E$ is connected to the corresponding locus of the suffix tree. Since $T[i..j]$ is unique in T, the locus of $T[i..j]$ is either the leaf corresponding to unique suffix $T[i..|T|]$ or on the edge towards the leaf. Thus we can perform operations 7–9 in constant time via the leaves of the suffix tree, for given (i, j), which represents some unique substring of T.

Finally, the data structure can be maintained in amortized $O(\log \sigma)$ time: Basically the amortized analysis is due to [22]. The secondary active point, which was originally proposed in [16], can also be maintained in a similar manner to the active point, and thus the amortized analysis for the secondary active point is almost the same as that for the active point in [22] (see [16] for the complete proof). ◀

By wrapping up the above discussions, we obtain the following theorem:

▶ **Theorem 11.** *We can compute the set of extended net occurrences of string T of length n given in an online manner in a total of $O(n \log \sigma)$ time using $O(n)$ space.*

4.2 Sliding-window algorithm based on implicit suffix trees

By applying symmetric arguments of Theorem 9, we can design a sliding-window algorithm.

▶ **Lemma 12.** *There exists a data structure $\mathcal{D}_T$ of size $O(|T|)$ that supports all nine operations of Theorem 9 in addition to their symmetric operations listed below in constant time.*

1. *Determine if $\#occ_T(sqPref(T)) = 2$.*
2. *Find the non-prefix occurrence of $sqPref(T)$ in T when $\#occ_T(sqPref(T)) = 2$.*
3. *Determine if $\#occ_T(lrPref(T)) = 2$.*
4. *Find the non-prefix occurrence of $lrPref(T)$ in T when $\#occ_T(lrPref(T)) = 2$.*
5. *Compute the lengths of $lrPref(T[2..n])$ and $lrPref(T)$.*
6. *Compute $exp' = \max\{e \mid c^e$ is a prefix of $T[2..n]\}$ and determine if $\#occ_{T[2..n]}(c^{exp'}) = 1$ where $c = T[1]$.*

The data structure $\mathcal{D}_T$ can be updated to either $\mathcal{D}_{T[2..|T|]}$ or $\mathcal{D}_{Tc}$ in amortized $O(\log \sigma)$ time where c is a character.

Proof. The sliding suffix tree data structure of [16] supports all the operations in amortized $O(\log \sigma)$ time using $O(d)$ space. ◀

In case we perform the deletion of the leftmost character and the addition of the rightmost character simultaneously, then our algorithm works for a sliding-window of fixed size d. On the other hand, our scheme is also applicable to a sliding-window of variable size. Thus we have the following:

▶ **Theorem 13.** *We can maintain the set of extended net occurrences for a sliding window over string T of length n in a total of $O(n \log \sigma)$ time using $O(d)$ working space where d is the maximum size of the window.*

4.3 Online algorithm based on implicit CDAWGs

The next lemma is an adaptation of Lemma 10 which uses implicit CDAWGs in place of implicit suffix trees:

▶ **Lemma 14.** *Based on the left-to-right online CDAWG construction [9], we can design a data structure $\mathcal{C}_T$ of size $O(\mathsf{e}(T))$ that supports all nine operations of Theorem 9 in constant time. The data structure $\mathcal{C}_T$ can be updated to $\mathcal{C}_{Tc}$ in amortized $O(\log \sigma)$ time where c is a character.*

Proof. Since the online implicit CDAWG construction algorithm [9] is based on Ukkonen's implicit suffix tree construction, it also maintains the active point that indicates the locus corresponding to $lrSuf(T)$. While the locus can correspond to multiple substrings of T (as the CDAWG is a DAG), we can retrieve $|lrSuf(T)|$ in $O(1)$ time by storing, in each node v of $\mathsf{CDAWG}'(T)$, the length of the maximal repeat corresponding to v. This is because the path that spells out $lrSuf(T)$ from the source consists only of the primary edges (see [9] for more details). Since edge label x is represented by an integer pair (p, q) such that $x = T[p..q]$, we can obtain the non-suffix occurrence (i', j') of $lrSuf(T)$ in $O(1)$ time (Line 9 in Algorithm 1).

The secondary active point that indicates the locus for $sqSuf(T)$ can also be maintained on the implicit $\mathsf{CDAWG}'(T)$ by adapting the algorithm from [16]. Let y be the suffix of T that is one-character shorter than $sqSuf(T)$. By definition, y is the longest suffix of T such that $\#occ_T(y) \geq 3$. Given the locus P for y on $\mathsf{CDAWG}'(T)$, one can check in $O(1)$ time whether the substrings corresponding to P occur at least 3 times, by checking the number of paths from P to the sink and checking if the active point is in the subgraph under P. Also, by definition, y is the longest string represented by the locus P. This tells us the length of $sqSuf(T)$ as well. Thus, we can also maintain the secondary active point in a similar manner to the active point on the implicit CDAWG in $O(\log \sigma)$ amortized time per character, and we can obtain the non-suffix occurrence (i, j) of $sqSuf(T)$ in $O(1)$ time (Line 3 in Algorithm 1).

The $O(\mathsf{e}(T))$-space requirement follows from Lemma 1. ◀

▶ **Theorem 15.** *We can compute the set of extended net occurrences of string T of length n given in an online manner in a total of $O(n \log \sigma)$ time using $O(\mathsf{e}(T))$ working space.*

Proof. The correctness and the time complexity follows from the above discussions.

It is shown in [9] that the function $\mathsf{e}'(T[1..i])$ is monotonically non-decreasing for any online string $T[1..i]$ with increasing $i = 1, \ldots, n$. Together with Lemma 1, we have $\mathsf{e}'(T[1..i]) \leq \mathsf{e}'(T) \leq \mathsf{e}(T)$ for any $1 \leq i \leq n$, which leads to an $O(\mathsf{e}(T))$-space bound. ◀

5 Relating extended net occurrences and MUSs

In this section, we give a full characterization of the extended net occurrences of repeats in T in terms of minimal unique substrings (MUSs) in T. See also Figure 5 for illustration.

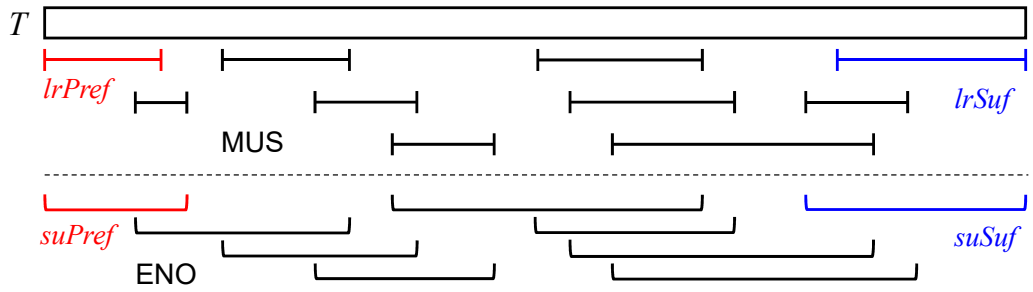

Figure 5 Illustration for Lemma 16 and Lemma 17.

▶ **Lemma 16.** *Let* $[i-1..h], [k..j+1] \in \mathsf{MUS}(T)$ *be the intervals that represent consecutive MUSs in* T*, namely, there is no element* $[s..t]$ *in* $\mathsf{MUS}(T)$ *such that* $i \leq s < k$ *and* $h < t \leq j$*. Then,* $[i..j]$ *is a net occurrence for repeat* $u = T[i..j]$*.*

Proof. Observe that any unique substring of T must contain a MUS. Since $T[i..j]$ does not contain a MUS, $u = T[i..j]$ is a repeat in T. Let $a = T[i-1]$ and $b = T[j+1]$. Then, $\#occ_T(aub) = \#occ_T(au) = \#occ_T(ub) = 1$ since any of $aub = T[i-1..j-1]$, $au = T[i-1..j]$, and $ub = T[i..j+1]$ contains a MUS. ◀

A consequence of Lemma 16 is that a net occurrence $[i..j]$ cannot be contained in another net occurrence $[i'..j']$. This is because a MUS cannot be contained in another MUS. Another consequence is that two consecutive extended net occurrences in $\mathsf{ENO}(T)$ are overlapping.

Below we show the reversed version of Lemma 16:

▶ **Lemma 17.** *Let* $\mathsf{L} = \mathsf{ENO}(T) \cup \{[1..p] \mid p = |lrPref(T)| + 1\} \cup \{[q..n] \mid q = n - |lrSuf(T)|\}$*. Let* $[h..j], [i..k] \in \mathsf{L}$ *be consecutive elements in* L*, namely, there is no element* $[s..t]$ *in* L *such that* $h < s < i$ *and* $j < t < k$*. Then,* $[i..j] \in \mathsf{MUS}(T)$*.*

Proof. By the definition of the extended net occurrences, there is a MUS $[x..j]$ with $x \geq h+1$ that ends at j since $T[h+1..j]$ is unique and $T[h+1..j-1]$ is repeating in T. Similarly, there is a MUS $[i..y]$ with $y \leq k-1$ that starts at i. Here, for the sake of contradiction, we assume $i \neq x$. If $i < x$, there are at least two MUSs within range $[i, j] \subset [h, k]$. If $i > x$, there are at least two MUSs within range $[x, y] \subset [h, k]$. In both cases, there exists some net occurrence within range $[h, k]$ by Lemma 16, which contradicts that $[h..j]$ and $[i..k]$ are consecutive elements in L. Thus $i = x$ holds. Similarly, we can prove $j = y$, hence $[i..j] \in \mathsf{MUS}(T)$.

Consider the case where $h = 1$ and $j = p$, namely $T[1..p]$ is the *shortest unique prefix* (*suPref*) of T, and $T[i..k]$ is the leftmost extended net occurrence in T. Again by the definition of the extended net occurrences, there is a MUS that begins at position i. Let $T[1..p] = ub$ where $u \in \Sigma^\star$ and $b \in \Sigma$. Then, $u = lrPref(T)$. Since ub is unique and since $u = lrPref(T)$, there must exist a MUS that ends at position p (see Figure 5.) Using a similar argument as above, it can be proven that these two MUSs are the same. Thus $[i..p] \in \mathsf{MUS}(T)$. The case where $i = q$ and $k = n$ is symmetric. ◀

Consequently, the next theorem follows from Lemma 16 and Lemma 17.

▶ **Theorem 18.** *For any string* T*,*

(1) $\#\mathsf{ENO}(T) = \#\mathsf{MUS}(T) - 1$*.*

(2) $\mathsf{ENO}(T)$ *can be obtained from the sorted* $\mathsf{MUS}(T)$ *in optimal* $O(\#\mathsf{ENO}(T))$ *time.*

(3) $\mathsf{MUS}(T)$ *can be obtained from the sorted* $\mathsf{ENO}(T)$*,* $|lrPref(T)|$*, and* $|lrSuf(T)|$ *in optimal* $O(\#\mathsf{ENO}(T))$ *time.*

We also have the following corollary for space-efficient computation of MUSs:

▶ **Corollary 19.** *We can maintain the set of all MUSs of a string T of length n given in an online manner in a total of $O(n \log \sigma)$ time using $O(\mathsf{e}(T))$ working space, where $\mathsf{e}(T)$ denotes the size of* $\mathsf{CDAWG}(T)$.

Proof. By combining Theorem 15 and Lemmas 16 and 17, we obtain the corollary except for computation of $|lrPref(T)|$. This can easily be maintained in the implicit CDAWG as follows. Let z be the node of $\mathsf{CDAWG}'(T)$ from which the primary edge to the sink stems out. We identify z with the maximal repeat that the node represents. If the active point does not exist on this primary edge, then $|z| = |lrPref(T)|$. If the active point lies on this primary edge leading to the sink, then $|z| + k = |lrPref(T)|$, where k is the offset of the active point on the primary edge from the node z. ◀

For any string T, $\#\mathsf{MUS}(T) \leq \mathsf{e}(T)$ holds [10]. Together with Theorem 18, we obtain:

▶ **Corollary 20.** *For any string T, $\#\mathsf{ENO}(T) < \mathsf{e}(T)$ holds.*

6 Conclusions and open questions

In this paper we presented how Ukkonen's left-to-right online suffix tree construction can be used for online computation of string net frequency. Our main contributions are space-efficient algorithms for computing string net occurrences, one works in $O(d)$ space in the sliding model for window-length d, and the other works in $O(\mathsf{e}(T))$ space where $\mathsf{e}(T)$ denotes the size of the CDAWG of the input string T. Both of our methods run in $O(n \log \sigma)$ time and can report all (extended) net occurrences of repeats in the current string in output-optimal time. We also showed that computing the sorted list of extended net occurrences of repeats in a string T is equivalent to computing the sorted list of minimal unique substrings (MUSs) in T.

An intriguing open question is whether one can efficiently compute the extended net occurrences of repeats within $O(\mathsf{r})$ space, where r denotes the number of equal-character runs in the BWT of the input string. It is known that $\mathsf{r} \leq \mathsf{e}$ holds for any string [1]. The *R-enum* algorithm of Nishimoto and Tabei [17] is able to compute the set of MUSs in $O(n \log \log_\omega(n/\mathsf{r}))$ time with $O(\mathsf{r})$ space, where ω denotes the machine word size of the word RAM model. However, it is unclear whether their algorithm can output a list of MUSs arranged in the sorted order of the beginning positions within $O(\mathsf{r})$ space.

References

1 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *CPM 2015*, volume 9133 of *Lecture Notes in Computer Science*, pages 26–39. Springer, 2015. doi:10.1007/978-3-319-19929-0_3.

2 Anselm Blumer, Janet Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987. doi:10.1145/28869.28873.

3 Dany Breslauer and Giuseppe F. Italiano. On suffix extensions in suffix trees. *Theor. Comput. Sci.*, 457:27–34, 2012. doi:10.1016/J.TCS.2012.07.018.

4 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, DIGITAL System Research Center, 1994.

5 Peaker Guo, Patrick Eades, Anthony Wirth, and Justin Zobel. Exploiting new properties of string net frequency for efficient computation. In *CPM 2024*, pages 16:1–16:16, 2024. doi:10.4230/LIPICS.CPM.2024.16.

6 Peaker Guo, Seeun William Umboh, Anthony Wirth, and Justin Zobel. Online computation of string net frequency. In *SPIRE 2024*, volume 14899 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2024. doi:10.1007/978-3-031-72200-4_12.

7 Lucian Ilie and William F. Smyth. Minimum unique substrings and maximum repeats. *Fundam. Informaticae*, 110(1-4):183–195, 2011. doi:10.3233/FI-2011-536.

8 Shunsuke Inenaga. Faster and simpler online computation of string net frequency. *CoRR*, abs/2410.06837, 2024. doi:10.48550/arXiv.2410.06837.

9 Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Giancarlo Mauri, and Giulio Pavesi. On-line construction of compact directed acyclic word graphs. *Discret. Appl. Math.*, 146(2):156–179, 2005. doi:10.1016/J.DAM.2004.04.012.

10 Shunsuke Inenaga, Takuya Mieno, Hiroki Arimura, Mitsuru Funakoshi, and Yuta Fujishige. Computing minimal absent words and extended bispecial factors with CDAWG space. In *IWOCA 2024*, volume 14764 of *Lecture Notes in Computer Science*, pages 327–340. Springer, 2024. doi:10.1007/978-3-031-63021-7_25.

11 N. Jesper Larsson. Extended application of suffix trees to data compression. In *DCC 1996*, pages 190–199. IEEE Computer Society, 1996. doi:10.1109/DCC.1996.488324.

12 Laurentius Leonard, Shunsuke Inenaga, Hideo Bannai, and Takuya Mieno. Sliding suffix trees simplified. *CoRR*, abs/2307.01412, 2023. doi:10.48550/arXiv.2307.01412.

13 Yih-Jeng Lin and Ming-Shing Yu. Extracting Chinese frequent strings without dictionary from a Chinese corpus, its applications. *J. Inf. Sci. Eng.*, 17(5):805–824, 2001. URL: http://www.iis.sinica.edu.tw/page/jise/2001/200109_07.html.

14 Yih-Jeng Lin and Ming-Shing Yu. The properties and further applications of Chinese frequent strings. In *International Journal of Computational Linguistics & Chinese Language Processing, Volume 9, Number 1, February 2004: Special Issue on Selected Papers from ROCLING XV*, pages 113–128, 2004.

15 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

16 Takuya Mieno, Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing minimal unique substrings for a sliding window. *Algorithmica*, 84(3):670–693, 2022. doi:10.1007/S00453-021-00864-1.

17 Takaaki Nishimoto and Yasuo Tabei. R-enum: Enumeration of characteristic substrings in bwt-runs bounded space. In *CPM 2021*, volume 191 of *LIPIcs*, pages 21:1–21:21, 2021. doi:10.4230/LIPICS.CPM.2021.21.

18 Enno Ohlebusch, Thomas Büchler, and Jannik Olbrich. Faster computation of Chinese frequent strings and their net frequencies. In *SPIRE 2024*, volume 14899 of *Lecture Notes in Computer Science*, pages 249–256. Springer, 2024. doi:10.1007/978-3-031-72200-4_19.

19 Jakub Radoszewski and Wojciech Rytter. On the structure of compacted subword graphs of Thue-Morse words and their applications. *J. Discrete Algorithms*, 11:15–24, 2012. doi:10.1016/J.JDA.2011.01.001.

20 Wojciech Rytter. The structure of subword graphs and suffix trees of Fibonacci words. *Theor. Comput. Sci.*, 363(2):211–223, 2006. doi:10.1016/J.TCS.2006.07.025.

21 Martin Senft. Suffix tree for a sliding window: An overview. In *WDS 2005*, volume 5, pages 41–46, 2005.

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

23 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.