Efficient Computation of Substring Equivalence Classes with Suffix Arrays

Kazuyuki Narisawa[1], Shunsuke Inenaga[2], Hideo Bannai[1], and Masayuki Takeda[1,3]

[1] Department of Informatics, Kyushu University, Fukuoka 819-0395, Japan
[2] Department of Computer Science and Communication Engineering, Kyushu University, Fukuoka 819-0395, Japan
[3] SORST, Japan Science and Technology Agency (JST)
{k-nari, bannai, takeda}@i.kyushu-u.ac.jp
inenaga@c.csce.kyushu-u.ac.jp

Abstract. This paper considers enumeration of substring equivalence classes introduced by Blumer et al. [1]. They used the equivalence classes to define an index structure called compact directed acyclic word graphs (CDAWGs). In text analysis, considering these equivalence classes is useful since they group together redundant substrings with essentially identical occurrences. In this paper, we present how to enumerate those equivalence classes using suffix arrays. Our algorithm uses rank and lcp arrays for traversing the corresponding suffix trees, but does not need any other additional data structure. The algorithm runs in linear time in the length of the input string. We show experimental results comparing the running times and space consumptions of our algorithm, suffix tree and CDAWG based approaches.

1 Introduction

Finding distinct features from text data is an important approach for text analysis, with various applications in literary studies, genome studies, and spam detection [2]. In biological sequences and non-western languages such as Japanese and Chinese, word boundaries do not exist, and thus all substrings of the text are subject to analysis. However, a given text contains too many substrings to browse or analyze. A reasonable approach is to partition the set of substrings into equivalence classes under the equivalence relation of [1] so that an expert can examine the classes one by one [3]. This equivalence relation groups together substrings that correspond to essentially identical occurrences in the text. Such a partitioning is very beneficial for various text mining approaches whose mining criterion is based on occurrence frequencies, since each element in a given equivalence class will have the same occurrence frequency.

In this paper, we develop an efficient algorithm for enumerating the equivalence classes of a given string, as well as useful statistics such as frequency and size for each class. Although the number of equivalence classes in a string w of length n is at most $n+1$, the total number of elements in the equivalence classes is $O(n^2)$, that is, the number of substrings in w. However, each equivalence class can be expressed by a unique maximal (longest) element and multiple minimal elements. Further, these elements can be expressed by a pair of integers representing the beginning and end positions in the string. Thus, we consider these succinct expressions of the equivalence classes, which require only $O(n)$ space. The succinct expressions can easily be computed using the CDAWG data structure proposed by [1], which is an acyclic graph structure whose nodes correspond to the equivalence classes. Although CDAWGs can be constructed in $O(n)$ time and space [4], we present a more efficient algorithm based on suffix arrays.

In Section 3, we first describe an algorithm using suffix trees with suffix links (Algorithm 1), for computing the succinct expressions. Although suffix trees can also be constructed and represented in $O(n)$ time and space [5, 6], it has been shown that many algorithms on suffix trees can be efficiently simulated on suffix arrays [7] with the help of auxiliary arrays such as lcp and rank arrays [8, 9]. However, previous methods require extra time and space for maintaining suffix link information. In Section 4, we give an algorithm to simulate Algorithm 1 using the suffix, lcp and rank arrays (Algorithm 2). A key feature of this algorithm is that it does not require any extra data structure other than these arrays, making it quite space economical. Section 5 gives results of computational experiments of Algorithm 1, 2, and an algorithm using CDAWGs.

2 Preliminaries

2.1 Notations

Let Σ be a finite set of symbols, called an *alphabet*. An element of Σ^* is called a *string*. Strings x, y and z are said to be a *prefix*, *substring*, and *suffix* of the string $w = xyz$, respectively, and the string w is said to be a *superstring* of substring y. The sets of prefixes, substrings and suffixes of a string w are denoted by $\mathit{Prefix}(w)$, $\mathit{Substr}(w)$ and $\mathit{Suffix}(w)$, respectively. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. The i-th symbol of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. Also, let $w[i :] = w[i : |w|]$ for $1 \leq i \leq |w|$. For any string $w \in \Sigma^*$ and $x \in \mathit{Substr}(w)$, a *reference pair* of x w.r.t. w is a pair $\langle i, j \rangle$ such that $w[i : j] = x$. For any strings $x, y \in \Sigma^*$, the longest string in $\mathit{Prefix}(x) \cap \mathit{Prefix}(y)$ is called the *longest common prefix* (*LCP*) of x and y.

2.2 Equivalence Relations on Strings

In this subsection, we recall the equivalence relations introduced by Blumer et al. [10, 1], and then state their properties. Throughout this paper, we consider the equivalence classes of the input string w that ends with a distinct symbol \$ that does not appear anywhere else in w.

For any string $x \in \mathit{Substr}(w)$, let,

$$\mathit{BegPos}(x) = \{i \mid 1 \le i \le |w|, x = w[i : i + |x| - 1]\}, \text{ and}$$
$$\mathit{EndPos}(x) = \{i \mid 1 \le i \le |w|, x = w[i - |x| + 1 : i]\}.$$

For any string $y \notin \mathit{Substr}(w)$, let $\mathit{BegPos}(y) = \mathit{Endpos}(y) = \emptyset$.

Now we define two equivalence relations and classes based on *BegPos* and *EndPos*.

Definition 1. *The equivalence relations $\equiv_L$ and $\equiv_R$ on Σ^* are defined by:*

$$x \equiv_L y \Leftrightarrow \mathit{BegPos}(x) = \mathit{BegPos}(y), \text{ and}$$
$$x \equiv_R y \Leftrightarrow \mathit{EndPos}(x) = \mathit{EndPos}(y),$$

where x, y are any strings in Σ^*. The equivalence class of a string $x \in \Sigma^*$ with respect to $\equiv_L$ and $\equiv_R$ is denoted by $[x]_{\equiv_L}$ and $[x]_{\equiv_R}$, respectively.

Notice that any strings not in $\mathit{Substr}(w)$ form one equivalence class under $\equiv_L$, called the degenerate class. Similar arguments hold for $\equiv_R$. The above equivalence classes $[x]_{\equiv_L}$ and $[x]_{\equiv_R}$ correspond to the nodes of *suffix trees* [5] and *directed acyclic word graphs* (*DAWGs*) [10], respectively. For any string $x \in \mathit{Substr}(w)$, let $\overrightarrow{x}$ and $\overleftarrow{x}$ denote the unique longest member of $[x]_{\equiv_L}$ and $[x]_{\equiv_R}$, respectively. For any string $x \in \mathit{Substr}(w)$, let $\overleftrightarrow{x} = \alpha x \beta$ such that $\alpha, \beta \in \Sigma^*$ are the strings satisfying $\overleftarrow{x} = \alpha x$ and $\overrightarrow{x} = x\beta$.

Intuitively, $\overleftrightarrow{x} = \alpha x \beta$ means that:

– Every time x occurs in w, it is preceded by α and followed by β.
– Strings α and β are longest possible.

Note that $\overleftarrow{(\overrightarrow{x})} = \overrightarrow{(\overleftarrow{x})} = \overleftrightarrow{x}$.

Now we define another equivalence relation, whose equivalence classes correspond to the nodes of *compact directed acyclic word graphs* (*CDAWGs*) [1].

Definition 2. *For any string $x, y \in \Sigma^*$, we denote $x \equiv y$, if and only if*

1. *$x \notin \mathit{Substr}(w)$ and $y \notin \mathit{Substr}(w)$, or*
2. *$x, y \in \mathit{Substr}(w)$ and $\overleftrightarrow{x} = \overleftrightarrow{y}$.*

The equivalence class of a string x with respect to $\equiv$ is denoted by $[x]_\equiv$. For any $x \in \mathit{Substr}(w)$, the unique longest member $\overleftrightarrow{x}$ of $[x]_\equiv$ is called the representative *of the equivalence class.*

Now we consider a *succinct representation* of each non-degenerate equivalence class under $\equiv$. For any $x \in \mathit{Substr}(w)$, let $\mathit{Minimal}([x]_\equiv)$ denote the set of minimal elements of $[x]_\equiv$, that is,

$$\mathit{Minimal}([x]_\equiv) = \{y \in [x]_\equiv \mid\ z \in \mathit{Substr}(y) \text{ and } z \in [x]_\equiv \text{ implies } z = y\}.$$

Namely, $\mathit{Minimal}([x]_\equiv)$ is the set of strings y in $[x]_\equiv$ such that there is no string $z \in \mathit{Substr}(y) - \{y\}$ with $z \equiv x$.

The following lemma shows that the strings in every non-degenerate equivalence class $[x]_\equiv$ can be represented by a pair of its representative $\overleftrightarrow{x}$ and $\mathit{Minimal}([x]_\equiv)$.

Lemma 1 ([3]). *For any x in $Substr(w)$, let $y_1, \ldots, y_k$ be the elements of $Minimal([x]_\equiv)$. Then,*

$$[x]_\equiv = Pincer(y_1, \overleftrightarrow{x}) \cup \cdots \cup Pincer(y_k, \overleftrightarrow{x}),$$

where $Pincer(y_i, \overleftrightarrow{x})$ is the set of strings z such that $z \in Substr(\overleftrightarrow{x})$ and $y_i \in Substr(z)$.

Now, a succinct representation of a non-degenerate equivalence class $[x]_\equiv$ is a pair of $\overleftrightarrow{x}$ and $Minimal([x]_\equiv)$, where $\overleftrightarrow{x}$ and all strings in $Minimal([x]_\equiv)$ are represented by their reference pairs w.r.t. w. We have the following lemma about the total space requirement for the succinct representations of all the equivalence classes under $\equiv$.

Lemma 2. *A list of succinct representations of all non-degenerate equivalence classes under $\equiv$ requires only $O(|w|)$ space.*

Let the *size* of non-degenerate equivalence class $[x]_\equiv$ be the number of substrings that belong to $[x]_\equiv$, that is, $|[x]_\equiv|$. Let $Freq(x)$ denote the occurrence frequency of x in w. If $x \equiv y$, then $Freq(x) = Freq(y)$. Therefore, we consider the frequency of an equivalence class $[x]_\equiv$ and denote this by $Freq([x]_\equiv)$.

2.3 Data Structures

We use the following data structures in our algorithms.

Definition 3 (Suffix Trees and Suffix Link Trees). *For any string w, the suffix tree of w, denoted $ST(w)$, is an edge-labeled tree structure (V, E) such that*

$$V = \{x \mid x = \overrightarrow{x}, x \in Substr(w)\}, \text{ and}$$
$$E = \{(x, \beta, x\beta) \mid x, x\beta \in V, \beta \in \Sigma^+,\ a = \beta[1],\ \overrightarrow{xa} = x\beta\},$$

where the second component β of each edge $(x, \beta, x\beta)$ in E is its label, and the suffix link tree of w, denoted $SLT(w)$, is a tree structure (V, E_ℓ) such that

$$E_\ell = \{(ax, x) \mid x, ax \in V, a \in \Sigma\}.$$

It is well known that $ST(w)$ with $SLT(w)$ can be computed in linear time and space [11, 6].

The root node of $ST(w)$ and $SLT(w)$ is associated with $\varepsilon = \overrightarrow{\varepsilon}$. Since the end-marker \$ is unique in w, every nonempty suffix of strings in w corresponds to a leaf of $ST(w)$, and only such a leaf exists in $ST(w)$. Therefore, each leaf can be identified by the beginning position of the corresponding suffix of w. The values $Freq(x)$ for all nodes $x \in V$ of $ST(w)$ can be computed in linear time and space by a post-order traversal on $ST(w)$.

For any node $x\beta$ with incoming edge $(x, \beta, x\beta)$ of $ST(w)$, let $Paths(x\beta) = \{x\beta' \mid \beta' \in Prefix(\beta) - \{\varepsilon\}\}$. Note that $Paths(x\beta) = [x\beta]_{\equiv_L}$, and therefore $\overrightarrow{z} = x\beta$ for any $z \in Paths(x\beta)$. It is easy to see that $|Paths(x\beta)| = |x\beta| - |x| = |\beta|$.

For any node x of $ST(w)$ such that $x \neq \varepsilon$, let $Parent(x)$ denote the parent of x.

Definition 4 (Suffix Arrays). *The* suffix array *[7] SA of any string w is an array of length $|w|$ such that $SA[i] = j$, where $w[j:]$ is the i-th lexicographically smallest suffix of w.*

The suffix array of string w can be computed in linear time from $ST(w)$, by arranging the out-going edges of any node of $ST(w)$ in the lexicographically increasing order of the first symbols of the edge labels. This way all the leaves of $ST(w)$ are sorted in the lexicographically increasing order, and they correspond to SA of w. Linear-time direct construction of SA has also been extensively studied [12–14].

Definition 5 (Rank and LCP Arrays). *The* rank *and* lcp *arrays of any string w are arrays of length $|w|$ such that $rank[SA[i]] = i$, and $lcp[i]$ is the length of the longest common prefix of $w[SA[i-1]:]$ and $w[SA[i]:]$ for $2 \le i \le |w|$, and $lcp[1] = -1$.*

Given SA of string w, the *rank* and *lcp* arrays of w can be computed in linear time and space [8].

3 Computing Equivalence Classes under ≡ Using Suffix Trees

In this section we present a suffix tree based algorithm to compute a succinct representation of each non-degenerate equivalence class, together with its size and frequency. This algorithm will be the basis of our algorithm of Section 4, which uses suffix arrays instead of trees.

The following lemma states how to check the equivalence relation $\equiv$ between two substrings using $ST(w)$.

Lemma 3. *For any $x, y \in Substr(w)$, $x \equiv y$ if and only if $Freq(\overrightarrow{x}) = Freq(\overrightarrow{y})$ and $\overrightarrow{x} \in Suffix(\overrightarrow{y})$ or vise versa.*

Proof. The case that $\overrightarrow{x} = \overrightarrow{y}$ is trivial. We consider the case that $\overrightarrow{x} \neq \overrightarrow{y}$. Assume w.l.o.g. that $|\overrightarrow{x}| < |\overrightarrow{y}|$.

Assume $x \equiv y$. Then we have $\overleftarrow{(\overrightarrow{x})} = \overleftrightarrow{x} = \overleftrightarrow{y} = \overleftarrow{(\overrightarrow{y})}$, which implies that $EndPos(\overrightarrow{x}) = EndPos(\overrightarrow{y})$. Thus we have $Freq(\overrightarrow{x}) = Freq(\overrightarrow{y})$. Since $|\overrightarrow{x}| < |\overrightarrow{y}|$, $\overrightarrow{x} \in Suffix(\overrightarrow{y})$.

Now assume $Freq(\overrightarrow{x}) = Freq(\overrightarrow{y})$ and $\overrightarrow{x} \in Suffix(\overrightarrow{y})$. Since $\overrightarrow{x} \in Suffix(\overrightarrow{y})$, we have $EndPos(\overrightarrow{x}) \supseteq EndPos(\overrightarrow{y})$. Moreover, since $Freq(\overrightarrow{x}) = Freq(\overrightarrow{y})$, we get $EndPos(\overrightarrow{x}) = EndPos(\overrightarrow{y})$. Hence $\overleftrightarrow{x} = \overleftarrow{(\overrightarrow{x})} = \overleftarrow{(\overrightarrow{y})} = \overleftrightarrow{y}$. □

Lemma 4. *For any node $x \in V$ of $ST(w)$ such that $\overleftrightarrow{x} = x$, let $\ell = \max\{i \mid Freq(x[i:]) = Freq(x)\}$. Then, $[x]_{\equiv} = \bigcup_{y \in [x]_{\equiv_R}} [y]_{\equiv_L} = \bigcup_{i=1}^{\ell} Paths(x[i:])$.*

Proof. By Lemma3. □

For each node x of $ST(w)$, $|\mathit{Paths}(x)| = |\mathit{Parent}(x)| - |x|$ and it can be precomputed by a post-order traversal on $ST(w)$. Thus, by the above lemma, the size of each non-degenerate equivalence class can be computed by a post-order traversal on $SLT(w)$.

In what follows, we show how to check whether or not a given node x in the suffix link tree traversal is the representative of the equivalence class under $\equiv$, namely, whether or not $x = \overleftrightarrow{x}$.

Lemma 5. *For any node $x \in V$ of $ST(w)$, $x = \overleftrightarrow{x}$ if and only if $\mathit{Freq}(ax) < \mathit{Freq}(x)$ for any $a \in \Sigma$ such that $ax \in V$.*

Proof. By Lemma 3. □

The following two lemmas follow from Lemma 5.

Lemma 6. *For any leaf node $x \in V$ of $SLT(w)$, $x = \overleftrightarrow{x}$.*

Proof. Since x is also a node of $ST(w)$, $x = \overrightarrow{x}$. We show that for any symbol $a \in \Sigma$, $\mathit{Freq}(ax) < \mathit{Freq}(x)$, and therefore, $x = \overleftarrow{x}$. Since x is a leaf of $SLT(w)$, we have $ax \notin V$ for any $a \in \Sigma$, for which there are the two following cases:

1. $ax \notin \mathit{Substr}(w)$. Then, $\mathit{Freq}(ax) = 0$ while $\mathit{Freq}(x) > 0$.
2. $ax \in \mathit{Substr}(w)$. Consider $\beta \in \Sigma^+$ such that $\overrightarrow{ax} = ax\beta \in V$. Then, we have that $\mathit{Freq}(ax) = \mathit{Freq}(ax\beta) \leq \mathit{Freq}(x\beta) < \mathit{Freq}(x)$.

In both cases we have $\mathit{Freq}(ax) < \mathit{Freq}(x)$, and hence $x = \overleftrightarrow{x}$ from Lemma 5. □

Lemma 7. *For any internal node $x \in V$ of $SLT(w)$ and for any $a \in \Sigma$ with $ax \in V$, $x = \overleftrightarrow{x}$ if and only if $\mathit{Freq}(ax) \neq \mathit{Freq}(x)$.*

Proof. ($\Rightarrow$) Since $x = \overleftrightarrow{x}$, we have $\mathit{Freq}(bx) \neq \mathit{Freq}(x)$ for any $b \in \Sigma$. ($\Leftarrow$) Since $\mathit{Freq}(x) \geq \sum_{b \in \Sigma} \mathit{Freq}(bx)$ and $\mathit{Freq}(x) > \mathit{Freq}(ax) > 0$, we have $\mathit{Freq}(x) > \mathit{Freq}(bx)$ for any $b \in \Sigma$. □

We have the following lemma concerning the minimal members of the non-degenerate equivalence classes.

Lemma 8. *For any nodes $x, ax \in V$ of $ST(w)$ with $a \in \Sigma$, and let yb be the shortest member in $\mathit{Paths}(ax)$ where $y \in \Sigma^*$ and $b \in \Sigma$. Then, $yb \in \mathit{Minimal}([ax]_\equiv)$ if and only if (1) $|\mathit{Paths}(ax)| > |\mathit{Paths}(x)|$ or (2) $\mathit{Freq}(ax) < \mathit{Freq}(x)$.*

Proof. It is clear when $y = \varepsilon$. We consider the case where $y \neq \varepsilon$. Let $y = ay'$ for $y \in \Sigma^*$.

($\Rightarrow$) Assume $ay'b \in \mathit{Minimal}([ax]_\equiv)$, which implies $ay' \notin [ax]_\equiv$ and $y'b \notin [ax]_\equiv$. First, consider the case where $x \notin [ax]_\equiv$. Then clearly (2) holds. For the case where $x \in [ax]_\equiv$, $\overrightarrow{y'b} \neq x$ since $y'b \notin [ax]_\equiv = [x]_\equiv$, and there exists a node corresponding to $\overrightarrow{y'b}$ on the path from y' to x. Therefore (1) holds.

($\Leftarrow$) Since $ay'b$ is the shortest member in $\mathit{Paths}(ax)$, $ay' \notin [ax]_\equiv$. It remains to show $y'b \notin [ax]_\equiv$. If we assume (2), $\mathit{Freq}(ax) < \mathit{Freq}(x) \leq \mathit{Freq}(y'b)$ since $y'b$

Algorithm 1: Algorithm for computing a succinct representation, the size and frequency of each non-degenerate equivalence class using suffix trees

```
Input: ST(w),SLT(w) : suffix tree and suffix link tree of w
Output: a succinct representation, the size and frequency of each
        non-degenerate equivalence class
1  foreach node v ∈ V in post-order of ST(w) do
2  |   calculate and store the values $Freq(\mathsf{v})$ and $|Paths(\mathsf{v})|$;
3  size := 0; freq := 0;
4  foreach node v ∈ V in post-order of SLT(w) do
5  |   if v is a leaf of SLT(w) or freq $\neq Freq(\mathsf{v})$ then
6  |   |   if size $\neq 0$ then
7  |   |   |   report size as the size of $[\mathsf{rep\_v}]_\equiv$;
8  |   |   |   minimal := minimal $\cup \{\langle i,j\rangle\}$ s.t. $w[i:j]$ is the shortest string in
   |   |   |   $Paths(\mathsf{old\_v})$;
9  |   |   |   report $(\langle i,j\rangle, \mathsf{minimal})$ as a succinct representation of $[\mathsf{rep\_v}]_\equiv$,
   |   |   |   where $w[i:j] = \mathsf{rep\_v}$;
10 |   |   |   minimal := $\emptyset$;
11 |   |   freq := $Freq(\mathsf{v})$; report freq as the frequency of $[\mathsf{rep\_v}]_\equiv$;
12 |   |   size := $|Paths(\mathsf{v})|$; len := $|Paths(\mathsf{v})|$; old_v := v; rep_v := v;
13 |   else
14 |   |   if len $> |Paths(\mathsf{v})|$ then
15 |   |   |   minimal := minimal $\cup \{\langle i,j\rangle\}$ s.t. $w[i:j]$ is the shortest string in
   |   |   |   $Paths(\mathsf{old\_v})$;
16 |   |   size := size + $|Paths(\mathsf{v})|$; len := $|Paths(\mathsf{v})|$; old_v := v;
17 |   end
18 end
```

is a prefix of x. Therefore, we have $y'b \notin [ax]_\equiv$ because $Freq(y'b) \neq Freq(ax)$. Next, assume (1) when (2) does not hold, that is, $Freq(y'b) = Freq(ax)$. Then, $\overrightarrow{y'b} \neq x$ or else, $|Paths(ax)| = |Paths(x)|$. Therefore, $y'b \notin [x]_\equiv = [ax]_\equiv$. □

A pseudo-code of the algorithm to compute a succinct representation of each non-degenerate equivalence class together with its size and frequency is shown as Algorithm 1. The above arguments lead to the following theorem.

Theorem 1. *Given $ST(w)$ and $SLT(w)$, Algorithm 1 computes succinct representations of all non-degenerate equivalence classes under $\equiv$, together with their sizes and frequencies in linear time.*

4 Computing Equivalence Classes under ≡ Using Suffix Array

In this section, we develop an algorithm that simulates Algorithm 1 using suffix arrays. Our algorithm is based on the algorithm by Kasai et al. [8] which simulates a post-order traversal on suffix trees with *SA*, *rank* and *lcp* arrays. A key feature of our algorithm is that it does not require any extra data structure other than the suffix, rank and lcp arrays, making it quite space economical.

For any string $x \in \mathit{Substr}(w)$, let

$$\mathit{Lbeg}(x) = \mathit{SA}[\min\{\mathit{rank}[i] \mid i \in \mathit{BegPos}(x)\}] \text{ and}$$
$$\mathit{Rbeg}(x) = \mathit{SA}[\max\{\mathit{rank}[i] \mid i \in \mathit{BegPos}(x)\}].$$

Recall that Algorithm 1 traverses $\mathit{SLT}(w)$. Our suffix array based algorithm simulates traversal on $ST(w)$, and when reaching any node x such that $x = \overleftrightarrow{x}$, it simulates suffix link tree traversal until reaching node $y \in \mathit{Suffix}(x)$ such that $y \not\equiv x$.

The next lemma states that for any node x of $ST(w)$, $\mathit{Freq}(x)$ is constant time computable using *rank* array.

Lemma 9. *For any node $x \in V$ of $ST(w)$,*

$$\mathit{Freq}(x) = \mathit{rank}[\mathit{Rbeg}(x)] - \mathit{rank}[\mathit{Lbeg}(x)] + 1.$$

When reaching any node x such that $x = \overleftrightarrow{x}$ in the post-order traversal on $ST(w)$, we compute a succinct representation of $[x]_\equiv$ due to Lemma 3. Examination of whether $x = \overleftrightarrow{x}$ can be done in constant time according to the following lemma.

Lemma 10. *For any node $x \in V$ of $ST(w)$, let $l = \mathit{Lbeg}(x)$ and $r = \mathit{Rbeg}(x)$. We have $x = \overleftrightarrow{x}$ if and only if at least one of the following holds: (1) $l-1 = 0$ or $r-1 = 0$, (2) $w[l-1] \neq w[r-1]$, or (3) $\mathit{rank}[r]-\mathit{rank}[l] \neq \mathit{rank}[r-1]-\mathit{rank}[l-1]$.*

Proof. ($\Rightarrow$) First, let us assume $x = \overleftrightarrow{x}$. If (1) and (2) do not hold, that is, $l-1 \neq 0$, $r-1 \neq 0$ and $w[l-1] = w[r-1] = a$, then $\mathit{Freq}(x) > \mathit{Freq}(ax)$ due to Lemma 3. This implies $\mathit{rank}[r] - \mathit{rank}[l] > \mathit{rank}[\mathit{Rbeg}(ax)] - \mathit{rank}[\mathit{Lbeg}(ax)] \geq \mathit{rank}[r-1] - \mathit{rank}[l-1]$ from Lemma 9, showing (3).

($\Leftarrow$) To show the reverse, we have only to show $\overleftarrow{x}$ since x is a node of $ST(w)$, and therefore $x = \overrightarrow{x}$. First, we show (1) $\Rightarrow x = \overleftrightarrow{x}$. If $l = 1$, then $|x| \in \mathit{EndPos}(x)$ while $|x| \notin \mathit{EndPos}(ax)$ for any $a \in \Sigma$, implying $x = \overleftarrow{x}$. The same applies for $r = 1$.

Next, we show (2) $\Rightarrow x = \overleftrightarrow{x}$ when (1) does not hold, that is, $l-1 \neq 0$ and $r-1 \neq 0$. Since $w[l-1] \neq w[r-1]$, we have that $l-1+|x| \in \mathit{EndPos}(x)$ while $l-1+|x| \notin \mathit{EndPos}(w[r-1]x)$ and $r-1+|x| \in \mathit{EndPos}(w[r-1]x)$. Therefore, $\mathit{Freq}(x) > \mathit{Freq}(w[r-1]x) > 0$, and since $\mathit{Freq}(x) \geq \sum_{a\in\Sigma} \mathit{Freq}(ax)$, we have $\mathit{Freq}(ax) < \mathit{Freq}(x)$ for all $a \in \Sigma$, thus implying $x = \overleftarrow{x}$.

Finally, we show (3) $\Rightarrow x = \overleftrightarrow{x}$ when (1) and (2) do not hold, that is, $l-1 \neq 0$, $r-1 \neq 0$ and $w[l-1] = w[r-1] = a$. (3) implies that $\mathit{rank}[r] - \mathit{rank}[l] > \mathit{rank}[\mathit{Rbeg}(ax)] - \mathit{rank}[\mathit{Lbeg}(ax)] \geq \mathit{rank}[r-1] - \mathit{rank}[l-1]$, and from Lemma 9 we have that $\mathit{Freq}(x) > \mathit{Freq}(ax) > 0$. Therefore $x \not\equiv ax$ from Lemma 3, implying $x = \overleftarrow{x}$.

Therefore, we have $x = \overleftrightarrow{x}$ if we assume at least one of (1)–(3). $\square$

Now we consider to check whether or not $ax \equiv x$ for any nodes ax, x of $ST(w)$, where $a \in \Sigma$ and $x \in \Sigma^*$. By definition, it is clear that $\mathit{Lbeg}(ax) + 1 \in \mathit{BegPos}(x)$ and $\mathit{Rbeg}(ax) + 1 \in \mathit{BegPos}(x)$. However, note that $\mathit{Lbeg}(ax) + 1 = \mathit{Lbeg}(x)$ does *not* always hold (same for Rbeg).

To check if $ax \equiv x$, we need to know whether or not $\mathit{Lbeg}(ax) + 1 = \mathit{Lbeg}(x)$, and it can be done by the following lemma:

Lemma 11. *For any nodes $ax, x \in V$ of $ST(w)$ such that $a \in \Sigma$ and $x \in \Sigma^*$, let $l = \mathit{Lbeg}(ax)$. Then, $\mathit{lcp}[\mathit{rank}[l+1]] < |ax| - 1$ if and only if $\mathit{Lbeg}(x) = l + 1$.*

Proof. If $\mathit{Lbeg}(x) = l + 1$, then clearly $\mathit{lcp}[\mathit{rank}[l+1]] < |x| = |ax| - 1$.

Now, assume on the contrary that $\mathit{Lbeg}(x) \neq l + 1$. Then, $\mathit{rank}[\mathit{Lbeg}(x)] < \mathit{rank}[l+1]$, and since $w[\mathit{Lbeg}(x) :]$ and $w[l+1 :]$ share x as a prefix, we have $\mathit{lcp}[\mathit{rank}[l+1]] \geq |x| = |ax| - 1$ which is a contradiction. □

The following lemma can be shown in a similar way to the above lemma:

Lemma 12. *For any nodes $ax, x \in V$ of $ST(w)$ such that $a \in \Sigma$ and $x \in \Sigma^*$, let $r = \mathit{Rbeg}(ax)$. Then, $\mathit{lcp}[\mathit{rank}[r+1]+1] < |ax| - 1$, if and only if $\mathit{Rbeg}(x) = r + 1$.*

Now we have the following lemma on which our examination of equivalence relation is based.

Lemma 13. *For any nodes $ax, x \in V$ of $ST(w)$ such that $a \in \Sigma$ and $x \in \Sigma^*$, we have $ax \equiv x$ if and only if*

(1) $|ax| - 1 > \mathit{lcp}[\mathit{rank}[l+1]]$,
(2) $|ax| - 1 > \mathit{lcp}[\mathit{rank}[r+1]+1]$, and
(3) $\mathit{rank}[r] - \mathit{rank}[l] = \mathit{rank}[r+1] - \mathit{rank}[l+1]$,

where $l = \mathit{Lbeg}(ax)$ and $r = \mathit{Rbeg}(ax)$.

Proof. ($\Rightarrow$) Assume $ax \equiv x$. Then, $\mathit{Freq}(ax) = \mathit{Freq}(x)$ and thus we have $\mathit{rank}[\mathit{Rbeg}(ax)] - \mathit{rank}[\mathit{Lbeg}(ax)] = \mathit{rank}[\mathit{Rbeg}(x)] - \mathit{rank}[\mathit{Lbeg}(x)]$ from Lemma 9. From $\mathit{Freq}(ax) = \mathit{Freq}(x)$, we have $\mathit{Rbeg}(x) = \mathit{Rbeg}(ax) + 1 = r + 1$ and $\mathit{Lbeg}(x) = \mathit{Lbeg}(ax) + 1 = l + 1$. By Lemma 11 and Lemma 12 we get $|ax| - 1 > \mathit{lcp}[\mathit{rank}[l+1]]$ and $|ax| - 1 > \mathit{lcp}[\mathit{rank}[r+1]+1]$.

($\Leftarrow$) From Lemma 11, if $|ax| - 1 > \mathit{lcp}[\mathit{rank}[l+1]]$, then $\mathit{Lbeg}(x) = l+1$. From Lemma 12, if $|ax| - 1 > \mathit{lcp}[\mathit{rank}[r+1]+1]$, then $\mathit{Rbeg}(x) = r + 1$. Therefore, if $\mathit{rank}[r] - \mathit{rank}[l] = \mathit{rank}[r+1] - \mathit{rank}[l+1]$, then $\mathit{Freq}(ax) = \mathit{Freq}(x)$ by Lemma 9. Consequently, we get $ax \equiv x$ from Lemma 3. □

Next, we consider how to compute $|\mathit{Paths}(x)| = |x| - |\mathit{Parent}(x)|$. When $x = \overleftrightarrow{x}$, we know $|x|$ and $|\mathit{Parent}(x)|$ which are computed in post-order traversal on $ST(w)$ simulated by the algorithm of [8]. When $x \neq \overleftrightarrow{x}$, namely, when x has been reached in suffix link traversal simulation, we have that $|x| = |ax| - 1$ where ax is the node reached immediately before x in the suffix link tree traversal simulation. We have the following lemma for computation of $|\mathit{Parent}(x)|$.

Algorithm 2: Algorithm for computing a succinct representation, the size and frequency of each non-degenerate equivalence class using suffix, lcp and rank arrays

```
Input: $SA[1:|\mathsf{w}|]$, $lcp[1:|\mathsf{w}|]$, $rank[1:|\mathsf{w}|]$ : suffix, lcp and rank arrays of string w
Output: a succinct representation, the size and frequency of each
        non-degenerate equivalence class
1  Stack initialization (Left, Height) := $(-1,-1)$;
2  for $i = 1,\dots,n$ do
3      Lnew := $i-1$; Hnew := $lcp[i]$; Left := Stack.Left; Height := Stack.Height;
4      while Height > Hnew do
5          Pop Stack;
6          if Stack.Height > Hnew then parent := Stack.Height;
7          else parent = Hnew;
8          L := Left; R := $i-1$; freq := R − L + 1; rlen := Height;
9          if $(SA[\mathsf{L}] \neq 1)\&(SA[\mathsf{R}] \neq 1)$ then
10             BL := $rank[SA[\mathsf{L}]-1]$; BR := $rank[SA[\mathsf{R}]-1]$;
11         if (BR−BL+1≠freq) or (w[BL]≠w[BR]) or $(SA[\mathsf{L}]=1)$ or $(SA[\mathsf{R}]=1)$ then
12             Let x = w$[SA[\mathsf{L}] : SA[\mathsf{L}] + \mathsf{rlen} - 1]$;
13             report freq as the frequency of $[\mathsf{x}]_\equiv$;
14             size := rlen − parent; mlen := rlen − parent; len := rlen; minimal := $\emptyset$;
15             FL := $rank[SA[\mathsf{L}]+1]$; FR := $rank[SA[\mathsf{R}]+1]$; BL := L; BR := R;
16             while (len−1 > $lcp$[FL])&(len−1 > $lcp$[FR+1])&(FR−FL+1 = freq) do
17                 if $lcp$[FL] ≥ $lcp$[FR + 1] then parent := $lcp$[FL];
18                 else parent := $lcp$[FR + 1];
19                 len := len − 1; size := size + len − parent;
20                 if mlen > len − parent then
21                     minimal := minimal ∪ {⟨$SA$[BL], $SA$[BL] + parent⟩};
22                 BL := FL; BR := FR;
23                 if ($SA$[FL] + 1 ≥ |w|) or ($SA$[FR] + 1 ≥ |w|) then break;
24                 FL := $rank[SA[\mathsf{FL}]+1]$; FR := $rank[SA[\mathsf{FR}]+1]$; mlen := len−parent;
25             report size as the size of $[\mathsf{x}]_\equiv$;
26             minimal := minimal ∪ {⟨$SA$[BL], $SA$[BL] + parent⟩};
27             report(⟨$SA$[L], $SA$[L] + rlen − 1⟩, minimal) as a succinct
               representation of $[\mathsf{x}]_\equiv$;
28         Lnew := Left; Left := Stack.Left; Height := Stack.Height;
29     if Height < Hnew then Push(Lnew, Hnew) to Stack;
30     Push($i$, |w| − $SA[i]$) to Stack;
31 end
```

Lemma 14. *For any node $x \in V$ of $ST(w)$, let $l = Lbeg(x)$ and $r = Rbeg(x)$. Then, $|Parent(x)| = \max\{lcp[rank[l]], lcp[rank[r]+1]\}$.*

Proof. For all $1 \leq i < rank[l]$, the length of the longest common prefix of $w[SA[i]:]$ and x is at most $lcp[rank[l]]$. Similarly for $rank[r] < j \leq |w|$, the length of the longest common prefix of $w[SA[j]:]$ and x is at most $lcp[rank[r]+1]$. Also, for all $rank[l] \leq k \leq rank[r]$, the longest common prefix of $w[SA[k]:]$ and x is $|x|$, and therefore $lcp[k] \geq |x|$ for all $rank[l] < k \leq rank[r]$. This implies that

$|Parent(x)|$ is equal to either $lcp[rank[l]]$ or $lcp[rank[r]+1]$ and hence the lemma follows. □

A pseudo-code of the algorithm is shown in Algorithm 2. The **for** and **while** loops on line 2 and line 4 simulate a post-order traversal on $ST(w)$ using SA, *rank* and *lcp* arrays, and it takes linear time due to [8]. Checking whether or not $x = \overleftrightarrow{x}$ for any node x reached in the post-order traversal on $ST(w)$, is done in line 11 due to Lemma 10. Thus, we go into the **while** loop on line 16 only when $x = \overleftrightarrow{x}$, and this **while** loop continues until reaching $y \in Suffix(x)$ such that $y \not\equiv x$ due to Lemma 13. It is clear that all calculations in the **while** loop can be done in constant time.

Theorem 2. *Given SA, rank and lcp arrays of string w, Algorithm 2 computes succinct representations of all non-degenerate equivalence classes under ≡, together with their sizes and frequencies in linear time.*

5 Experimental Results

We performed preliminary experiments on corpora [15, 16], to compare practical time and space requirements of suffix tree, CDAWG, and suffix array based approaches to compute a succinct representation of for each non-degenerate equivalence class under ≡, together with its size and frequency.

We constructed suffix trees using Ukkonen's algorithm [6], and ran Algorithm 1. CDAWGs were constructed using the CDAWG construction algorithm of [4]. We computed suffix arrays using the qsufsort program by [17]. All the experiments were conducted on a RedHat Linux desktop computer with a 2.8 GHz Pentium 4 processor and 1 GB of memory.

Table 1 shows the running time and memory usage of the algorithms for each data structure. The enumeration column shows the time efficiency of the algorithms computing succinct representations of all equivalence classes together with their sizes and frequencies. For all the corpora the suffix array approach was the fastest. In addition, the suffix array algorithm uses the least memory space for all the corpora.

References

1. Blumer, A., Blumer, J., Haussler, D., Mcconnell, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. J. ACM 34(3) (1987) 578–595
2. Narisawa, K., Bannai, H., Hatano, K., Takeda, M.: Unsupervised spam detection based on string alienness measures. Technical report, Department of Informatics, Kyushu University (2007)
3. Takeda, M., Matsumoto, T., Fukuda, T., Nanri, I.: Discovering characteristic expressions in literary works. Theoretical Computer Science 292(2) (2003) 525–546
4. Inenaga, S., Hoshinoa, H., Shinohara, A., Takeda, M., Arikawa, S., Mauri, G., Pavesi, G.: On-line construction of compact directed acyclic word graphs. Discrete Applied Mathematics **146**(2) (2005) 156–179

Table 1. The comparison of the computation time and memory space for suffix trees, CDAWGs and suffix arrays.

corpora data name	data size (Mbytes)	data structure	Time (seconds)			memory (Mbytes)
			construction	enumeration	total	
cantrby/plrabn12	0.47	Suffix Tree	0.95	0.21	1.16	21.446
		CDAWG	0.97	0.18	1.15	9.278
		Suffix Array	**0.43**	**0.14**	**0.57**	**5.392**
ProteinCorpus/sc	2.8	Suffix Tree	12.08	1.43	13.51	121.877
		CDAWG	12.76	1.12	13.88	69.648
		Suffix Array	**3.08**	**0.63**	**3.71**	**33.192**
large/bible.txt	3.9	Suffix Tree	7.33	2.23	9.56	191.869
		CDAWG	6.68	1.62	8.30	56.255
		Suffix Array	**4.71**	**1.50**	**6.21**	**46.319**
large/E.coli	4.5	Suffix Tree	8.17	2.91	11.08	232.467
		CDAWG	8.58	2.31	10.89	139.802
		Suffix Array	**5.95**	**1.46**	**7.41**	**53.086**

5. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th IEEE Annual Symp. on Switching and Automata Theory. (1973) 1–11
6. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3) (1995) 249–260
7. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Computing **22**(5) (1993) 935–948
8. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Proc. of CPM'01. Volume 2089 of LNCS., Springer-Verlag (2001) 181–192
9. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms **2**(1) (2004) 53–86
10. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.: The smallest automaton recognizing the subwords of a text. Theoretical Computer Science **40** (1985) 31–55
11. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2) (1976) 262–272
12. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proc. ICALP'03. Volume 2719 of LNCS., Springer-Verlag (2003) 943–955
13. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Proc. CPM'03. Volume 2676 of LNCS. (2003) 186–199
14. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Proc. CPM'03. Volume 2676 of LNCS. (2003) 200–210
15. Arnold, R., Bell, T.: A corpus for the evaluation of lossless compression algorithms. In: Proc. DCC '97. (1997) 201–210 `http://corpus.canterbury.ac.nz/`.
16. Nevill-Manning, C., Witten, I.: Protein is incompressible. In: Proc. DCC '99. (1999) 257–266 `http://www.data-compression.info/Corpora/ProteinCorpus/index.htm`.
17. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden (1999) `http://www.larsson.dogma.net/qsufsort.c`.