Inferring Strings from Graphs and Arrays

Hideo Bannai[1], Shunsuke Inenaga[2],
Ayumi Shinohara[2,3], and Masayuki Takeda[2,3]

[1] Human Genome Center, Institute of Medical Science, University of Tokyo
4-6-1 Shirokanedai, Minato-ku, Tokyo 108-8639, Japan
bannai@ims.u-tokyo.ac.jp
[2] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
[3] PRESTO, Japan Science and Technology Corporation (JST)
{s-ine,ayumi,takeda}@i.kyushu-u.ac.jp

Abstract. This paper introduces a new problem of *inferring strings from graphs*, and *inferring strings from arrays*. Given a graph G or an array A, we infer a string that suits the graph, or the array, under some condition. Firstly, we solve the problem of finding a string w such that the *directed acyclic subsequence graph* (*DASG*) of w is isomorphic to a given graph G. Secondly, we consider *directed acyclic word graphs* (*DAWGs*) in terms of string inference. Finally, we consider the problem of finding a string w of a minimal size alphabet, such that the *suffix array* (*SA*) of w is identical to a given permutation $p = p_1, \ldots, p_n$ of integers $1, \ldots, n$. Each of our three algorithms solving the above problems runs in linear time with respect to the input size.

1 Introduction

To process *strings* efficiently, several kinds of data structures are often used. A typical form of such a structure is a *graph*, which is specialized for a certain purpose such as pattern matching [1]. For instance, *directed acyclic subsequence graphs* (*DASGs*) [2] are used for subsequence pattern matching, and *directed acyclic word graphs* (*DAWGs*) [3] are used for substring pattern matching. It is quite important to construct these graphs as fast as possible, processing the input strings. In fact, for any string, its DASG and DAWG can be built in linear time in the length of a given string. Thus, the input in this context is a string, and the output is a graph.

In this paper, we introduce a challenging problem that is a 'reversal' of the above, namely, a problem of *inferring strings from graphs*. That is, given a directed graph G, we infer a string that suits G under some condition. Firstly, we consider the problem of finding a string w such that the DASG of w is isomorphic to a given unlabeled graph G. We show a characterization theorem that gives if-and-only-if conditions so that a directed acyclic graph is isomorphic to a DASG. Our algorithm inferring a string w from G as a DASG is based on this theorem, and it will be shown to run in linear time in the size of G. Secondly, we consider DAWGs in terms of the string inference problem. We also give a linear-

B. Rovan and P. Vojtáš (Eds.): MFCS 2003, LNCS 2747, pp. 208–217, 2003.

time algorithm that finds a string w such that the DAWG of w is isomorphic to a given unlabeled graph G.

Another form of a data structure for string processing is an *array* of integers. A problem of inferring strings from arrays was first considered by Franěk *et al.* [4]. They proposed a method to check if an integer array is a *border array* for some string w. Border arrays are better known as *failure functions* [5]. They showed an on-line linear-time algorithm to verify if a given integer array is a border array for some string w on an unbounded size alphabet. Duval *et al.* [6] gave an on-line linear-time algorithm for a bounded size alphabet, to solve this problem.

On the other hand, in this paper we consider *suffix arrays* (*SAs*) [7] in the context of string inference. Namely, given a permutation $p = p_1, \dots, p_n$ of integers $1, \dots, n$, we infer a string w of a minimal size alphabet, such that the SA of w is identical to p. We present a linear time algorithm to infer string w from a given p.

1.1 Notations on Strings

Let Σ be a finite alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. 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. String u is said to be a *subsequence* of string w if u can be obtained by removing zero or more characters from w. The set of subsequences of a string w is denoted by $\mathit{Subseq}(w)$.

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 character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i:j] = \varepsilon$ for $j < i$.

For strings $w, u \in \Sigma^*$, we denote $w \equiv u$ if w is obtained from u by one-to-one character replacements. For a string w let Σ_w denote the set of the characters appearing in w.

1.2 Graphs

Let V be a finite set of *nodes*. An *edge* is defined to be an ordered pair of nodes. Let E be a finite set of edges. A *directed graph* G is defined to be a pair (V, E).

For an edge (u, v) of a directed graph G, u is called a *parent* of v, and v is called a *child* of u. Let $\mathit{Children}(u) = \{v \in V \mid (u, v) \in E\}$, and $\mathit{Parents}(v) = \{u \in V \mid (u, v) \in E\}$. Node u (v, respectively) is called the *head* (*tail*, respectively) of edge (u, v). An edge (u, v) is said to be an *out-going* edge of node u and an *in-coming* edge of node v. A node without any in-coming edges is said to be a *source* node of G. A node without any out-going edges is said to be a *sink* node of G.

In a directed graph G, the sequence of edges $(v_0, v_1), (v_1, v_2), \dots, (v_{n-1}, v_n)$ is called a *path*, and denoted by $\mathit{path}(v_0, v_n)$. The *length* of the path is defined to be the number of edges in the path, namely, n. If $v_0 = v_n$, the path is called a *cycle*. If G has no cycles, it is called a *directed acyclic graph* (*DAG*).

An edge of a *labeled* graph G is an ordered triple (u, a, v), where $u, v \in V$ and $a \in \Sigma$. A path $(v_0, a_1, v_1), (v_1, a_2, v_2), \ldots (v_{n-1}, a_n, v_n)$ is said to *spell out* string $a_1 a_2 \cdots a_n$. For a labeled graph G, let $s(G)$ be the graph obtained by removing all edge-labels from G. For two labeled graphs G and H, we write as $G \cong H$ if $s(G)$ is isomorphic to $s(H)$.

Recall the following basic facts on Graph Theory, which will be used in the sequel.

Lemma 1 (e.g. [8] pp. 6–10). *Checking if a given directed graph is acyclic can be done in linear time.*

Lemma 2 (e.g. [8] pp. 6–8). *Connected components of a given undirected graph can be computed in linear time.*

Without loss of generality, we consider in this paper, DAGs $G = (V, E)$ with exactly one source node and sink node, denoted *source* and *sink*, respectively. We also assume that for all nodes $v \in V$ (excluding *source* and *sink*), there exists both $path(source, v)$ and $path(v, sink)$. For nodes $u, v \in V$, let us define $pathLengths(u, v)$ as the multi-set of lengths of all paths from u to v, and let $depths(v) = pathLengths(source, v)$.

2 Inferring String from Graph as DASG

This section considers the problem of inferring a string from a given graph as an unlabeled DASG.

For a subsequence x of string $w \in \Sigma^*$, we consider the end-position of the leftmost occurrence of x in w and denote it by $LM_w(x)$, where $0 \le |x| \le LM_w(x) \le |w|$. We define an equivalence relation $\sim_w^{seq}$ on Σ^* by

$$x \sim_w^{seq} y \Leftrightarrow LM_w(x) = LM_w(y).$$

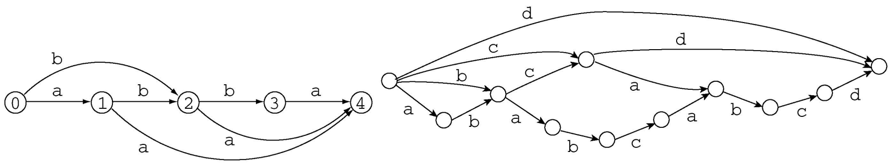
Let $[x]_w^{seq}$ denote the equivalence class of a string $x \in \Sigma^*$ under $\sim_w^{seq}$. The *directed acyclic subsequence graph* (*DASG*) of string $w \in \Sigma^*$, denoted by $DASG(w)$, is defined as follows:

Definition 1. *$DASG(w)$ is the DAG (V, E) such that*

$$V = \{[x]_w^{seq} \mid x \in Subseq(w)\},$$
$$E = \{([x]_w^{seq}, a, [xa]_w^{seq}) \mid x, xa \in Subseq(w) \text{ and } a \in \Sigma\}.$$

According to the above definition, each node of $DASG(w)$ can be associated with a position of w uniquely. When we indicate the position i of a node v of $DASG(w)$, we write as v_i.

Theorem 1 (Baeza-Yates [2]). *For any string $w \in \Sigma^*$, $DASG(w)$ is the smallest (partial) DFA that recognizes all subsequences of w.*



(a) (b)

Fig. 1. (a) $DASG(w)$ with $w =$ `abba` (b) $DAWG(w)$ with $w =$ `ababcabcd`

$DASG(w)$ with $w =$ `abba` is shown in Fig. 1 (a). Using $DASG(w)$, we can examine whether or not a given pattern $p \in \Sigma^*$ is a subsequence of w in $O(|p|)$ time [2]. Details of construction and applications of DASGs can be found in the literature [2].

Theorem 2. *A labeled DAG $G = (V, E)$ is $DASG(w)$ for some string w of length n, if and only if the following properties hold.*

1. **Path property** *There is a unique path of length n from source to sink.*
2. **Node number property** $|V| = n + 1$.
3. **Out-going edge labels property** *The labels of the out-going edges of each node v are mutually distinct.*
4. **In-coming edge labels property** *The labels of all in-coming edges of each node v are equal. Moreover, the integers assigned to the tails of these edges are consecutive.*
5. **Character positions property** *For any node $v_k \in V$, assume $Parents(v_k) \neq \emptyset$. Assume $v_i \in Parents(v_k)$ and $v_{i-1} \notin Parents(v_k)$ for some $1 \leq i < k$. If the in-coming edges of v_k are labeled by some character a, then edge (v_{i-1}, v_i) is also labeled by a.*

The path of Property 1 is the unique longest path of G, which spells out w. We call this path the *backbone* of G. The backbone of $DASG(w)$ can be expressed by sequence $(v_0, w[1], v_1), \ldots, (v_{n-1}, w[n], v_n)$.

Lemma 3. *For any two strings $u, w \in \Sigma^*$, $u \equiv w$ if and only if $DASG(u) \cong DASG(w)$.*

The above lemma means that, if an unlabeled DAG is isomorphic to the DASG of some string, the string is uniquely determined except for apparent one-to-one character replacements.

Theorem 3. *Given an unlabeled graph $G = (V, E)$, the string inference problem for DASGs can be solved in linear time.*

Proof. We describe a linear time algorithm which, when given unlabeled graph $G = (V, E)$, infers a string w where $s(DASG(w))$ is isomorphic to G. First, recall that the acyclicity test for given graph G is possible in linear time (Lemma 1). If it contains a cycle, we reject it and halt. While traversing G to test the acyclicity of G, we can also compute the length of the longest path from *source* to *sink* of G, and let n be the length. We at the same time count the number of nodes in G. If $|V| \neq n+1$, we reject it and halt. Then, we assign an integer i to each node v of G such that the length of the longest path from *source* to v is i. This corresponds to a topological sort of nodes in G, and it is known to be feasible in $O(|V|+|E|)$ time (e.g. [8] pp. 6–8).

After the above procedures, the algorithm starts from *sink* of G. Let w be a string of length n initialized with *nil* at each position. The variable *unlabeled* indicates the rightmost position of w where the character is not determined yet, and thus it is initially set to $n = |w|$. At step i, the node at position *unlabeled* is given a new character c_i. We then determine all the positions of the character c_i in w, by backward traversal of in-coming edges from *sink* towards *source*. To do so, we preprocess G after ordering the nodes topologically. At node v_i of G, for each $v_j \in \mathit{Children}(v_i)$ we insert v_i to the list maintained in v_j, corresponding to a reversed edge (v_j, v_i). Since there exists exactly $n+1$ nodes in G, the integers assigned to nodes in the backbone are sorted from 0 to n. Therefore, if we start from *source*, the list of reversed edges of every node is sorted in increasing order. Thus, given a node *node*, we can examine if the numbers assigned to nodes in $\mathit{Parents}(\mathit{node})$ are consecutive, in time linear in the number of elements in the list of the reversed edges of *node*. If they are consecutive, the next position where c_i appears in w corresponds to the smallest value in the set (the first element in the list), and the process is repeated for this node until we reach *source*. If, at any point, the elements in the set are not consecutive, we reject G and halt. This part is based on Properties 4 and 5 of Theorem 2. If, in this process, we encounter a position of w in which a character is already determined, we reject G and halt since if G is a DASG, for any position its character has to be *uniquely* determined. After we finish determining the positions of c_i in w, we decrement *unlabeled* until $w[\mathit{unlabeled}]$ is undetermined, or if we reach *source*. If $\mathit{unlabeled} \neq 0$ (if not *source*), then the process is repeated for a new character c_{i+1}. Otherwise, all the characters have been determined, and we output w. Since each edge is traversed (backwards) only once, and *unlabeled* is decremented at most n times, we can conclude that the whole algorithm runs in linear time with respect to the size of G. □

3 Inferring String from Graph as DAWG

This section considers the problem of inferring a string from a given graph as an unlabeled DAWG.

Definition 2 (Crochemore [9]). *The* directed acyclic word graph *(*DAWG*) of $w \in \Sigma^*$ is the smallest (partial) DFA that recognizes all suffixes of w.*

The DAWG of $w \in \Sigma^*$ is denoted by $\mathit{DAWG}(w)$. $\mathit{DAWG}(w)$ with $w =$ `ababcabcd` is shown in Fig. 1 (b). Using $\mathit{DAWG}(w)$, we can examine whether or not a given pattern $p \in \Sigma^*$ is a substring of w in $O(|p|)$ time. Details of construction and applications of DAWGs can be found in the literature [3].

Lemma 4. *For any two strings $u, w \in \Sigma^*$, $u \equiv w$ if and only if $DAWG(u) \cong DAWG(w)$.*

The above lemma means that, if an unlabeled DAG is isomorphic to the DAWG of some string, the string is uniquely determined except for apparent one-to-one character replacements.

We assume that any string w terminates with a special delimiter symbol \$ which does not appear in prefixes. Then the suffixes of w are all recognized at *sink* of $DAWG(w)$, spelled out from *source*. Note that, on such an assumption, $DAWG(w)$ is the smallest DFA recognizing all substrings of w. It is not difficult to see that a DAWG will have the following properties.

Theorem 4. *If a labeled DAG G is $DAWG(w)$ for some string w of length n, then the following properties hold.*

1. **Length property** *For each length $i = 1, \ldots, n$, there is a unique path from source to sink of length i, where n is the length of the longest path.*
2. **In-coming edge labels property** *The labels of all in-coming edges of each node v are equal.*
3. **Suffix property** *Let $u_i = u_i[1]u_i[2] \ldots u_i[i]$ be the labels of a path of length i from source to sink. Then $u_i[i-j] = w[n-j]$ for each $j = 0, \ldots, i-1$.*

The above theorem gives necessary properties for a DAG to be a DAWG. Therefore, if a DAG G does not satisfy a property of the above theorem, then we can immediately decide that G is not isomorphic to any DAWG.

A naïve way to check the length property would take $O(n^2)$ time since the total lengths of all the paths is $\Sigma_{i=1}^{n} i$, but we here introduce how to confirm the length property in linear time. The length property claims that $depths(sink) = \{1, 2, \ldots, n\}$ holds, where n is the length of the longest path in G from *source* to *sink*. The next lemma is a stronger version of the length property, which holds for any node.

Lemma 5. *Let w be an arbitrary string of length n. For any node v in $DAWG(w)$, the multi-set $depths(v)$ consists of distinct consecutive integers, that is, $depths(v) = \{i, i+1, \ldots, j\}$ for some $1 \le i \le j \le n$.*

Lemma 6. *Length property can be verified in linear time with respect to the total number of edges in the graph.*

Proof. If a given G forms $DAWG(w)$ for some string w, by Lemma 5, at each node v, the multi-set $depths(v)$ consists of distinct consecutive integers. Thus $depths(v) = \{i, i+1, \ldots, j\}$ can be represented by the pair $\langle i, j \rangle$ of the minimum i and the maximum j. Starting from *source*, we traverse all nodes in a depth-first manner, where all in-coming edges of a node must have been traversed to go deeper. If a node v has only one parent node u, then $depths(v)$ is simply $\langle i+1, j+1 \rangle$ where $depths(u) = \langle i, j \rangle$. If a node v has $k > 1$ parent nodes u_1, …, u_k, we do as follows. Let $\langle i_1, j_1 \rangle = depths(u_1)$, …, $\langle i_k, j_k \rangle = depths(u_k)$. By Lemma 5, $depths(v) = \langle i_1 + 1, j_1 + 1 \rangle \cup \cdots \cup \langle i_k + 1, j_k + 1 \rangle$ must be equal to

$\langle i_{\min}+1, j_{\max}+1\rangle$, where $i_{\min} = \min\{i_1, \ldots, i_k\}$ and $j_{\max} = \max\{j_1, \ldots, j_k\}$. (Remark that the union operation is taken over multi-sets.) This can be verified by sorting the pairs $\langle i_1, j_1\rangle$, ..., $\langle i_k, j_k\rangle$ with respect to the first component in increasing order into $\langle i'_1, j'_1\rangle$, ..., $\langle i'_k, j'_k\rangle$, $(i'_1 < \cdots < i'_k)$ and checking that $j'_1 + 1 = i'_2$, ..., $j'_{k-1} + 1 = i'_k$. The sorting and verification can be done in $O(k)$ time at each node with a radix sort and skip count trick, provided that we prepare an array of size n before the traversal, and reuse it.

If $\mathit{depths}(\mathit{sink}) = \langle 1, n\rangle$ finally, the length property holds. The running time is linear with respect to the number of edges, since each edge is only processed once as out-going, and once as in-coming. □

Theorem 5. *Given an unlabeled graph $G = (V, E)$, the string inference problem for DAWGs can be solved in linear time.*

Proof (sketch). We describe a linear time algorithm which, when given unlabeled graph $G = (V, E)$, infers a string w where $s(\mathit{DAWG}(w))$ is isomorphic to G. The algorithm is correct, provided that there exists such a string for G. Invalid inputs can be rejected with linear time sanity checks, after the inference.

Initially, we check the acyclicity of the graph in linear time (Lemma 1), and find *source* and *sink*. Using the algorithm of Lemma 6, we verify the length property in linear time. At the same time, we can mark at each node, its deepest parent, that is, the parent on the longest path from *source*. Notice that Property 2 of Theorem 4 allows us to label the nodes instead of the edges. From Definition 2, it is easy to see that the labels of out-going edges from *source* are distinct and should comprise the alphabet Σ_w, and therefore we assign distinct labels to nodes in $\mathit{Children}(\mathit{source})$ (the label for *sink* can be set to '\$').

The algorithm then resembles a simple breadth-first traversal from *sink*, going up to *source*. For any set N of nodes, let $\mathit{Parents}(N) = \bigcup_{u \in N} \mathit{Parents}(u)$. Starting with $N_0 = \{\mathit{sink}\}$, at step i, we will consider labeling a set $N_{i+1} \subseteq \mathit{Parents}(N_i)$ of nodes whose construction is defined below. Nodes may have multiple paths of different lengths to the *sink*, and it is marked *visited* when it is first considered in the set. N_{i+1} is constructed by including all unvisited nodes, as well as a single deepest visited node (if any), in $\mathit{Parents}(N_i)$ (*sink* is also disregarded since it cannot have a label). With this construction, we will later see that at least one node in N_{i+1} will have already been labeled, and therefore from Property 3 of Theorem 4, all other nodes in N_{i+1} can be given the same label. When there are no more unvisited nodes, we infer the resulting string w, which is spelled out by longest path from *source* to *sink*. The linear run time of the algorithm is straightforward, since it is essentially a linear time breadth-first traversal of the DAG with one extra width at most (notice that redundant traversals from visited nodes can be avoided by using only the deepest parent node marked at each node), and the depth of the traversal is at most the length of the longest path from *source* to *sink*.

The claim that N_{i+1} will contain at least one labeled node for all i is justified as follows. If N_{i+1} contains a node marked visited, we can use this node since the label of nodes are always inferred when they are marked visited. If N_{i+1} does not contain a visited node, it is not difficult to see from its construction that this implies that N_{i+1} represents the set of *all* nodes which have a path of length $i+1$ to the *sink*. Then, from the length property, we can see that at least one of these nodes is labeled in the initial distinct labeling of $Children(source)$.

If G was not a valid structure for a DAWG, $s(DAWG(w))$ may not be isomorphic to G. However, G is labeled at the end of the inference algorithm, and we can check if the labeled G and $DAWG(w)$ are congruent or not in linear time. This is done by first creating $DAWG(w)$ from w in linear time [3], checking the number of nodes and edges, and then doing a simultaneous linear-time traversal on $DAWG(w)$ and labeled G. For each pair of nodes which have the same path from *source* in both graphs, the labels of the out-going edges are compared. □

The inclusion of a single deepest visited node (if any) when constructing N_{i+1} from $Parents(N_i)$ is the key to the linear time algorithm, because including all visited nodes in $Parents(N_i)$ would result in quadratic running time, while not including any visited nodes would result in failure of inferring the string for some inputs.

4 Inferring String from Suffix Array

A *suffix array SA* of a string w of length n is a permutation $p = p_1, \ldots, p_n$ of the integers $1, \ldots, n$, which represents the lexicographic ordering of the suffixes $w[p_i : n]$. Details of construction and applications of suffix arrays can be found in the literature [7].

Opposed to the string inference problem for DASGs and DAWGs, the inferred string cannot be determined uniquely (with respect to $\equiv$). For example, for a given suffix array $p = p_1, \ldots, p_n$, we can easily create a string $w = w[1] \ldots w[n]$ with an alphabet of size n, where $w[i]$ is set to the character with the p_ith lexicographic order in the alphabet. Therefore, we define the string inference problem for suffix arrays as: given a permutation $p = p_1 \ldots p_n$ of integers $1, \ldots, n$, construct a string w with a *minimal* alphabet size, whose suffix array $SA(w) = p$.

The only condition that a permutation $p = p_1 \ldots p_n$ must satisfy for it to represent a suffix array of string w is, for all $i \in 1, \ldots n-1$, $w[p_i : n] \leq_{lex} w[p_{i+1} : n]$, where $\leq_{lex}$ represents the lexicographic relation over strings. From the suffix array, we are provided with the lexicographic ordering of each of the characters in the string, that is, $w[p_1] \leq_{lex} \cdots \leq_{lex} w[p_n]$. Let I denote the set of integers where $i \in I$ indicates $w[p_i] <_{lex} w[p_{i+1}]$, that is, $w[p_i]$ is lexicographically strictly less than $w[p_{i+1}]$. A strict inequality $w[p_i] <_{lex} w[p_{i+1}]$ implies that the characters of $w[p_i]$ and $w[p_{i+1}]$ are different, and therefore increases the alphabet size. If $w[p_1] <_{lex} \cdots <_{lex} w[p_n]$, that is, if $I = \{1, \ldots, n-1\}$, then this is the same as in the previous example where we obtain the trivial string of alphabet size n. If $I = \phi$, this indicates a single character alphabet where the only possible suffix array $p = p_1, \ldots, p_n = n, \ldots, 1$. Our problem is to find the smallest I where p still holds as a suffix array for some string w with alphabet size $|I| + 1$.

Theorem 6. *Given a permutation* $p = p_1, \ldots, p_n$ *of integers* $1, \ldots, n$*, the string inference problem for SAs can be solved in linear time.*

Proof. We give a linear time algorithm to find the smallest I defined as above. The algorithm itself is very simple: for all $i = 1, \ldots, n-1$, if $w[p_i + 1] \not\leq_{lex} w[p_{i+1}+1]$ then $i \in I$ ($w[n+1]$ is defined to be first in the lexicographic ordering of the alphabet). The validity of the algorithm is shown below.

Define a mapping from a position j in the string, to its lexicographic order k, that is $r_1, \ldots, r_n$ so that $r_j = k$ such that $p_k = j$. For $i \in 1, \ldots n-1$, consider the two suffixes $w[p_i : n] \leq_{lex} w[p_{i+1} : n]$. Notice that, if there exists j s.t. $w[p_i + j] \not\leq_{lex} w[p_{i+1} + j]$, then there must $\exists k < j$ s.t. $w[p_i + k] <_{lex} w[p_{i+1} + k]$.

Suppose for some i, $w[p_i + j] \not\leq_{lex} w[p_{i+1} + j]$ with some $j \geq 1$, and $w[p_i + k] \leq_{lex} w[p_{i+1} + k]$ with all $0 \leq k < j$. If $j = 1$, this indicates that $w[p_i] <_{lex} w[p_{i+1}]$ must hold in order for the lexicographic order of the suffixes $w[p_i : n] \leq_{lex} w[p_{i+1} : n]$ to hold, and i must be included in I. If $j \geq 2$, we show that such conditions are covered by the conditions satisfied with $j = 1$ for a different i.

Suppose for some i, $w[p_i + j] \not\leq_{lex} w[p_{i+1} + j]$ with some $j \geq 2$, and $w[p_i + k] \leq_{lex} w[p_{i+1} + k]$ with all $0 \leq k < j$. Since $w[p_i + j - 1] \leq_{lex} w[p_{i+1} + j - 1]$, we have their lexicographic order $r_{p_i+j-1} < r_{p_{i+1}+j-1}$. For convenience, denote these as r' and r'' respectively, that is, $r' < r''$. Since we have $w[p_i + j] \not\leq_{lex} w[p_{i+1}+j]$, it follows that $w[p_{r'}+1] \not\leq_{lex} w[p_{r''}+1]$. Therefore, there must exist i' ($r' \leq i' < r''$) such that $w[p_{i'}+1] \not\leq_{lex} w[p_{i'+1}+1]$ (and thus $w[p_{i'}] <_{lex} w[p_{i'+1}]$), and it should belong to I. However, this condition will be found by the algorithm which only considers the case for $j = 1$. □

For a given permutation $p = p_1, \ldots, p_n$, let $k(p)$ represent the size of the minimal alphabet that w can consist of, for which $SA(w) = p$. Interestingly, the number of permutations p of length n where $k(p) = k$, is given by the Eulerian number $\left\langle {n \atop k} \right\rangle$ [10]. This is because Eulerian numbers can be interpreted as the number of permutations of length n which have k *ascents* (*descents*), in the permutation. This is exactly the condition we check for in Theorem 6.

5 Conclusions and Open Problems

In this paper we introduced a new challenging problem named *string inference*, where we infer strings from given graphs or arrays. We gave linear-time algorithms to solve the problem for DASGs and DAWGs. We also extended this scheme to arrays, and gave an algorithm that infers a string from a given suffix array in linear time.

One interesting open problem is whether inferring a string from a given *factor oracle* [11] can be done in linear time. The factor oracle of a string w is a DFA that 'at least' accepts $Substr(w)$, but possibly accepts some subsequences of w as well. Factor oracles therefore can be regarded as an 'intermediate' data structure between DAWGs and DASGs. To infer a string from a given unlabeled DAG as a factor oracle, we shall need to know what the language accepted by the factor oracle of w is, but it is still unknown. Therefore, the formal definition of factor oracles is awaited, and it would be a part of our future work as well.

We are also interested in string inference from *suffix trees* [12]. The suffix tree of string w is a tree structure that represents $Substr(w)$. The point is that its edge labels are *strings* (multiple characters). Also, the *compact DAWG* (*CDAWG*) [13] of w is a DAG recognizing $Substr(w)$ with string edge labels. Therefore, to infer a string from a suffix tree or CDAWG, we need to infer edge labels as strings but their lengths are *not* given beforehand. We expect that some kinds of *word equations* will be involved in this problem, and thus this class of the string inference problem should be far more complex than those we have solved in this paper.

References

1. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2002)
2. Baeza-Yates, R.A.: Searching subsequences (note). Theoretical Computer Science **78** (1991) 363–376
3. 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
4. Franěk, F., Gao, S., Lu, W., Ryan, P.J., Smyth, W.F., Sun, Y., Yang, L.: Verifying a border array in linear time. J. Comb. Math. Comb. Comput. (2002) 223–236
5. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The design and analysis of computer algorithms. Addison-Wesley (1974)
6. Duval, J.P., Lecroq, T., Lefevre, A.: Border array on bounded alphabet. In: Proc. The Prague Stringology Conference '02 (PSC'02), Czech Technical University (2002) 28–35
7. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Compt. **22** (1993) 935–948
8. Atallah, M.J., ed.: Algorithms and Theory of Computation Handbook. CRC Press (1998) ISBN:0-8493-2649-4.
9. Crochemore, M.: Transducers and repetitions. Theoretical Computer Science **45** (1986) 63–86
10. Graham, R., Knuth, D., Patashnik, O.: Concrete Mathematics (2nd edition). Addison Wesley (1994)
11. Allauzen, C., Crochemore, M., Raffinot, M.: Factor oracle: A new structure for pattern matching. In: Proc. 26th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'99). Volume 1725 of LNCS., Springer-Verlag (1999) 291–306
12. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th Annual Symposium on Switching and Automata Theory. (1973) 1–11
13. Blumer, A., Blumer, J., Haussler, D., McConnell, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. Journal of the ACM **34** (1987) 578–595