A Dynamic Data Structure for Reverse
Lexicographically Sorted Prefixes

Hidetoshi Yokoo*

Department of Computer Science, Gunma University
Kiryu 376-8515, Japan
yokoo@cs.gunma-u.ac.jp

Abstract. This paper proposes a simple data structure, called a prefix list, which maintains all
prefixes of a string in reverse lexicographic order. It can be on-line incrementally constructed in
time and space linear in the string length. It is strongly related to suffix trees and suffix arrays,
and may share applications with these existing structures. A suffix array can be built via the

corresponding prefix list in linear time. Particular applications of the prefix list lie in source-coding
problems that require on-line right-to-left string matching. We apply the prefix list to on-line
estimation of source entropy and to context-based symbol-ranking text compression algorithms.

1 Introduction

We propose a simple data structure, called the prefix list, which can store all
prefixes of a string in reverse lexicographic order. A prefix list can be on-line
incrementally constructed in time and space linear in the string length. We can
apply it to string matching problems and to data compression algorithms.

The proposed data structure is deeply related to such index structures as
suffix trees [4], [10], [12] and suffix arrays [8], [6]. The suffix array for a text is
an array of integers which represent lexicographic orders of all suffixes of
the text. It was proposed as a space-efficient alternative to the more ubiquitous
suffix tree. Whether we use suffix trees or suffix arrays, we usually suppose a
text to be static and fixed in the sense that we preprocess it to accept multiple
queries afterwards. In particular, a suffix array must be constructed from scratch
even if a bit of modification is added to the text. In some actual situations, we
must answer index-based string-matching problems while incrementally reading
a text. A suffix tree, which can be constructed in an on-line manner, serves as
a strong tool in such situations. However, since strings are represented in one
direction from the root to leaves on a suffix tree, it is difficult to match strings
from right to left. We actually have some on-line string problems, in which we
should match strings in that direction.

* Partially supported by the Kayamori foundation of informational science advance-
ment and by the Okawa foundation for information and telecommunications.

© Springer-Verlag Berlin Heidelberg 1999
Such situations sometimes arise in the context modeling stage in text compression. In most symbolwise (predictive) text compression algorithms, an upcoming symbol is predicted on the basis of its context. Such a "context-based" method gathers previous contexts according to their similarities to the current context. This requires an on-line right-to-left string matching process. Actually, the prefix list presented in this paper was initially suggested as an adaptive implementation of the context table, which was proposed as a common basis for representing text compression algorithms [15]. The most straightforward application of the prefix list is the context-sorting text compression algorithm [14]. It virtually prepares a ranked list of all possible symbols, ordered from most likely to least likely, but actually gives ranks to symbol candidates by searching a sorted list of previous contexts, sorted in reverse lexicographic order. In Matias et al. [9], in which the implementation of similar algorithms was referred to as the HYZ compression problem, the authors proposed to augment suffix trees to solve the problem.

A prefix list represents every prefix in a string as a node in a doubly-linked linear list. It is similar to the suffix tree in that on-line incremental construction is possible, and to the suffix array in that lexicographic linear order is incorporated. It seems that we need $O(n^2)$ time to construct a prefix list from a text of length $n$. However, if the text is an output from a finite-order Markov source, the expected complexity is reduced to $O(n)$. For a pattern generated from the same source, we can match it with the text in time linear in the pattern length.

In the next section, we define the prefix list and give an on-line procedure for its construction. We show that we can build it from a Markovian text in linear time. In Section 3, we slightly augment the prefix list to apply it to estimating the entropy of an actual text. Section 4 briefly reviews the context-sorting text compression algorithm, which motivates the development of prefix list. Section 5 is a survey of other possible applications.

2 Proposed Data Structure and its Construction

Let

$$S[1..n] = s_1 s_2 \cdots s_n \quad (s_i \in \Sigma, \ 1 \leq i \leq n) \quad (1)$$

be an $n$-symbol string over an ordered alphabet $\Sigma$ of size $|\Sigma|$. We represent a substring $s_i \cdots s_j$ as $S[i..j]$ and define $S[i..j] = \varepsilon$, the empty string, for $i > j$. The prefix of a string $S[1..n]$ that ends at position $i$ is $S[1..i]$, and the suffix that begins at position $i$ is $S[i..n]$. The $i$th symbol $s_i$ is also denoted by $S[i]$.

Based on the ordering relation on $\Sigma$, we can define its associated lexicographic order on the set of all strings. Reverse lexicographic ordering is lexicographic ordering of reversed strings. For example, the word ‘dog’ reverse-lexicographically (re-lexically, for short) precedes the word ‘deer’ since ‘god’ lexically precedes ‘reed’. Our new data structure maintains a re-lexically sorted set of all the prefixes of $S[1..n]$. As an example, consider the string:

$$S[1..9] = yabrecabr, \quad (2)$$
\begin{figure}[h]
\centering
\begin{tabular}{ll}
$S[1.0]$ & $\varepsilon$ \\
$S[1.7]$ & yabrec\textsuperscript{a} \\
$S[1.2]$ & ya \\
$S[1.8]$ & yabrecab \\
$S[1.3]$ & yab \\
$S[1.6]$ & yabrec\textsuperscript{c} \\
$S[1.5]$ & yabrec\textsuperscript{a} \\
$S[1.9]$ & yabrecab\textsuperscript{b} \\
$S[1.4]$ & yabr \\
$S[1.1]$ & y \\
\end{tabular}
\caption{Re-lexically sorted list of prefixes of $S[1..9] = 'yabrecab'$.}
\end{figure}

\begin{figure}[h]
\centering
\begin{tabular}{ll}
\text{sorted} & \text{sorted} \\
$\varepsilon$ & $\varepsilon$ \\
y & yabrec\textsuperscript{b} \\
ya & yab \\
yabrecab & yabrecab\textsuperscript{a} \\
yabrec\textsuperscript{b} & yabrec\textsuperscript{a} \\
yabrec\textsuperscript{c} & yabrec\textsuperscript{a} \\
yabrecab\textsuperscript{b} & yabrecab\textsuperscript{b} \\
yabr & yabr \\
y & y \\
\end{tabular}
\caption{Inserting the next prefix. The upcoming symbol is compared with the following symbols.}
\end{figure}

in which the ten prefixes including the empty string are re-lexically sorted in the order shown in Fig. 1.

For a pair of two adjacent prefixes in a re-lexically sorted list of prefixes, if a prefix $S[1..i]$ immediately follows $S[1..j]$, then the prefix $S[1..j]$ is said to be the immediate successor of $S[1..i]$ and, conversely, $S[1..i]$ is said to be the immediate predecessor of $S[1..j]$. We can insert an upcoming prefix into the re-lexically sorted list of previously occurred prefixes in a simple way. Consider again the example in (2), which may be followed by some symbol $s_{10}$. If the same symbol as $s_{10}$ has not appeared so far, then the reverse lexicographic order of $S[1..10]$ is determined only by its last symbol $s_{10}$. Otherwise, that is, if we have already had $s_{10}$ in $S[1..9]$, then we can find the position of $S[1..10]$ by searching the so-far occurred symbols for the match with $s_{10}$. As shown in Fig. 2, starting from the position corresponding to $s_{10}$, we search bidirectionally the following symbols of the sorted prefixes for the same symbol as $s_{10}$. If we hit a symbol $s_i = s_{10}$ in the $\uparrow$ direction, we should insert $S[1..10]$ as the immediate successor of $S[1..i]$. Conversely, if we find the same symbol as $s_{10}$ in the $\downarrow$ direction then we should insert $S[1..10]$ as the immediate predecessor of $S[1..i]$. These can be validated by the recursive property of reverse lexicographic order. As a specific example, suppose that we have $s_{10} = 'e'$ in the example in Fig. 2. Then, we immediately reach $s_7 = 'e'$ in the $\downarrow$ direction. This implies that we should insert $S[1..10] = 'yabrecab' as the immediate predecessor of $S[1..5]$. Another case may have $s_{10} = 'a'$. In this case, we hit either $s_7 = 'a'$ in the $\uparrow$ direction or $s_9 = 'a'$ in the $\downarrow$ direction. In any of both cases, the position of $S[1..10] = 'yabrecab' in the re-lexically sorted list of these prefixes is known to be between $S[1..7]$ and $S[1..2].$

The prefix list is natural realization of the above idea. It is implemented as a doubly-linked linear list in which each element, or node, contains one integer and three pointers. The three pointers are $\text{pred}$ and $\text{succ}$ list pointers used to organize the doubly-linked list and $\text{next}$ pointer used to designate the next position in the input string. Every prefix in a string is represented by a node of the list. If a node corresponds to the $i$th prefix $S[1..i]$, then its integer field contains the value of position index $i$. Its $\text{pred}$ and $\text{succ}$ pointers point to nodes corresponding to
the immediate predecessor and immediate successor of $S[1..i]$, respectively. The next pointer in the node for $S[1..i]$ points to the node for $S[1..i+1]$. If a node corresponds to the entire string $S[1..n]$, then its next pointer is set to nil. The initial state of a prefix list consists of a single special node $H$, which represents the empty string. We may add an extra node $T$ into the end of a prefix list in order to simplify some list operations. If we schematically represent a node pointed to by a pointer $P$ as is shown in Fig. 3, in which the left ($\leftarrow$) and right ($\rightarrow$) arrows represent the pred and succ pointers, respectively, and the vertical arrow ($\downarrow$) represents the next pointer, then our sample string in (2) is represented by the list shown in Fig. 4.

As mentioned above, a prefix list can be constructed incrementally in an online manner. Assume that the list representing all prefixes of an initial segment $S[1..i]$ has been already constructed and that the $i+1$st prefix $S[1..i+1]$ is about to be inserted. Let $P$ be a pointer that points to the just-inserted node for $S[1..i]$. If the upcoming symbol $s_{i+1}$ alphabetically precedes or succeeds any symbol seen so far, then the node for $S[1..i+1]$ should be inserted into the right of the list head ($H$) or the left of the list tail ($T$), respectively. Otherwise, if the symbol $s_{i+1}$ is not included in $S[1..i]$, then the list has a unique position $Q$ where the corresponding node $Q \uparrow$ satisfies

$$S[Q \uparrow \text{idx}] \prec s_{i+1} \prec S[Q \uparrow \text{succ} \uparrow \text{idx}].$$

Here, ‘$\prec$’ denotes the alphabetic order on $\Sigma$. We should insert a new node between the two nodes pointed to by $Q$ and $Q \uparrow \text{succ}$.

If the same symbol as $s_{i+1}$ has already appeared in $S[1..i]$, the inequalities in (3) may hold with equality. In this case, in the re-lexically sorted list of prefixes of $S[1..i+1]$, the immediate predecessor or successor of $S[1..i+1]$ has the same last symbol as $s_{i+1}$. If the immediate predecessor $S[1..j+1]$ of $S[1..i+1]$ has the same last symbol $s_{j+1}$ as $s_{i+1}$ ($0 \leq j < i$), then $S[1..j]$ re-lexically precedes $S[1..i]$. The node corresponding to $S[1..j]$ should be the first node with the same following symbol $s_{j+1}$ as $s_{i+1}$ when traversing the list from the current node to the head. We can see whether the following symbol matches $s_{i+1}$ by traversing the next pointer. Conversely, if the last symbol $s_{j+1}$ of the immediate successor $S[1..j+1]$ of $S[1..i+1]$ is equal to $s_{i+1}$, then the node for $S[1..j]$ should be the first node satisfying $s_{j+1} = s_{i+1}$ when traversing the list from the current node to the
Fig. 5. Re-lexically sorted contexts and their following symbols.

tail. Thus, starting from the current node for $S[1..i]$, we search bidirectionally the list for the node for $S[1..j]$ while comparing the following symbols with $s_{i+1}$. Once we have found the node for $S[1..j]$, we can immediately reach the node for $S[1..j+1]$ via the next pointer. Then, the position where we should insert a new node representing $S[1..i+1]$ is adjacent to that node for $S[1..j+1]$.

Now, we show that a prefix list can be constructed in linear time if the string in question is drawn from a Markov source of finite order. In such a string, the $k$th symbol can be completely characterized by the conditional probabilities \( \Pr(s_k | S[k - m..k - 1]) \) with $s_k \in \Sigma$, $S[k - m..k - 1] \in \Sigma^m$, where $\Pr(s_k | S[k - m..k - 1])$ is the conditional probability of $s_k$ given $S[k - m..k - 1]$. We say that the $k$th symbol $s_k$ occurs in the context $S[k - m..k - 1]$.

Suppose that, for sufficiently large $i$, we are about to insert the node corresponding to $S[1..i+1]$. To do it, we search the list bidirectionally for the match of $s_{i+1}$. We assume that the search is performed in both directions alternately. Figure 5 shows that the symbol-comparisons with $s_{i+1}$ are done in $x_1$, $x_2$, $x_3$, $x_4$, $x_5$, $x_6$, $x_7$ order. We evaluate the number of symbol-comparisons in this search. Let $K_s$ be the total number of symbols compared until we reach the match $s_{i+1} = s$, and $C$ denote the longest common context of those symbols. Then, the expected number of $K_s$ is estimated as

\[
E\{K_s\} = \frac{1}{\Pr(s_{i+1} = s \mid C)}.
\]  

(4)

Conversely, for any context $C$, the expected number of tested symbols over all possible upcoming symbols becomes

\[
E\{K\} = \sum \Pr(s_{i+1} = s \mid C) \cdot E\{K_s\} = |\{s \in \Sigma : \Pr(s \mid C) > 0\}| \triangleq \sigma_C \leq \sigma.
\]  

(5)
Table 1. Number of symbol-comparisons required to insert each prefix.

<table>
<thead>
<tr>
<th>FILE</th>
<th>size (bytes)</th>
<th>number of distinct symbols</th>
<th>number of symbol-comparisons maximum</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand</td>
<td>300000</td>
<td>53</td>
<td>36481</td>
<td>42.24</td>
</tr>
<tr>
<td>paper1</td>
<td>53161</td>
<td>95</td>
<td>34520</td>
<td>24.53</td>
</tr>
<tr>
<td>bib</td>
<td>111261</td>
<td>81</td>
<td>47442</td>
<td>29.69</td>
</tr>
<tr>
<td>alice29.txt</td>
<td>152089</td>
<td>74</td>
<td>34520</td>
<td>24.53</td>
</tr>
<tr>
<td>news</td>
<td>377109</td>
<td>98</td>
<td>200587</td>
<td>78.28</td>
</tr>
<tr>
<td>plrabin12.txt</td>
<td>481861</td>
<td>81</td>
<td>164435</td>
<td>23.11</td>
</tr>
<tr>
<td>book2</td>
<td>610856</td>
<td>96</td>
<td>459095</td>
<td>70.11</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>82</td>
<td>419926</td>
<td>33.10</td>
</tr>
<tr>
<td>obj2</td>
<td>246814</td>
<td>256</td>
<td>119013</td>
<td>124.55</td>
</tr>
<tr>
<td>pic</td>
<td>513216</td>
<td>159</td>
<td>287924</td>
<td>46.95</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>1029744</td>
<td>256</td>
<td>671660</td>
<td>124.36</td>
</tr>
</tbody>
</table>

Here,

$$\sigma \triangleq |\{a \in \Sigma | P(a) > 0\}|$$

(6)

denotes the number of symbols with non-zero probability. Therefore, the expected time complexity of the construction of a prefix list is linear in the string length with the coefficient of

$$\sum \Pr(C) \sigma_c \leq \sigma.$$  (7)

Note that the equality in (5) holds when a data string is drawn from a memoryless source.

The above estimation is valid for Markovian data of finite order. In order to validate it on actual data, we performed simple measurement on the number of symbol-comparisons on artificial and natural data. In actual situations, \(\sigma\), the number of symbols with non-zero probability, can be regarded as the number of distinct symbols occurred in the string. Thus, we make comparisons between the number of distinct symbols actually occurred and the number of symbol-comparisons required in the insertion of the prefixes. Table 1 shows some results of the measurement.

The first file, "rand," consists of lower- and upper-case letters and spaces, totally 53 distinct symbols. We assigned random but fixed probabilities to these symbols, and generated a sample sequence of 300000 symbols. Thus, the file can be thought of as a realization of a memoryless source. The other files come from the Calgary [2] and Canterbury [1] corpora, both of which are collected as standard data for the evaluation of text compression algorithms. The "size" column in Table 1 includes the length of each file. The number of distinct symbols in each file, which corresponds to \(\sigma\), is shown in the third column. The fourth column represents the maximum number of symbol-comparisons required.
in searching the list for each symbol when we insert prefixes into the list. The last column is the average number of symbol-comparisons. Obviously, it never exceeds the corresponding number of distinct symbols in any file. On the file "rand," both numbers are almost the same, which is naturally expected on data from a memoryless source.

3 On-line Computation of the Shortest Unique Substrings with an Application to Entropy Estimation

Figure 6 shows an extension of our data structure, where we add an auxiliary quantity to each node which represents the length of the longest common suffix of the strings corresponding to the node and to its immediate successor. This quantity serves as a measure for context similarity between two contexts which are adjacent to each other in a re-lexically sorted list of contexts. In this section, we describe on-line computation of the quantities and its application to the estimation of the entropy of a data source.

Let $S[1..j]$ denote the immediate successor of the $i$th prefix $S[1..i]$. Letting $l_i$ be the maximum $l$ such that $S[1..i+1-l] = S[1..i+j-l]$, we add it to the node corresponding to $S[1..i]$. Assume that the node has both immediate successor $S[1..j]$ and immediate predecessor $S[1..k]$ just after inserting that node ($1 \leq j < i$, $1 \leq k < i$). This implies that, immediately before the insertion of the node, the two nodes $S[1..k]$ and $S[1..j]$ are directly adjacent to each other. Suppose that these two nodes have had $l_k$ and $l_j$, respectively, as shown in Fig. 7. The state in Fig. 7 may be changed into a new one by the insertion of the node for $S[1..i]$. As shown in Fig. 8, the value of $l_j$ remains unchanged while the value of $l_k$ may increase to $l_k'$. These satisfy both $l_i \geq l_k$ and $l_k' \geq l_k$. More specifically,

- If $l_k' > l_k$ then $l_i = l_k$ otherwise $l_i \geq l_k';$
- If $l_k > l_k'$ then $l_k' = l_k$ otherwise $l_k' \geq l_k$.

We can use the above relations to minimize the number of actual comparisons required to compute $l_k'$ and $l_i$.

![Fig. 6. Context similarities with the immediate successors (lengths of the common suffixes).](image)

![Fig. 7. A pair of adjacent nodes.](image)

![Fig. 8. After inserting the $i$th prefix.](image)
Our first application of the prefix list is the estimation of entropy of actual data. As is well known in information theory, the data compression limit of a string is given by the entropy of its source. Of the methods for estimating entropy from sample data, the ones most related to our method are the SWE (Sliding-window Entropy) estimator and Grassberger’s estimator [13]. Our estimate [15] from a string $S[1..n]$ is defined by

$$\hat{H}_n = n \log n \left( \frac{1}{n} \sum_{i=1}^{n} L_i \right)^{-1},$$

where $L_i$ is the minimum $l$ such that a copy of the substring $S[i-l+1..i]$ does not appear anywhere else in the string. Thus, $L_i$ represents the length of the shortest unique substring ending at position $i$. For the immediate predecessor $S[1..k]$ of $S[1..n]$, the value of $L_i$ can be calculated as

$$L_i = \max\{l_i, l_k\} + 1.$$ 

Combining $L_i$ with the equation (8), we can perform entropy estimation in an on-line manner.

Figure 9 shows an example of entropy estimation, where the estimate $\hat{H}_n$ is plotted as a function of the input length $n$. The sample text is a concatenation of four Jane Austen’s novels [7], which is the same as that used by Kontoyiannis et al. Compared with their results, we know that our method is not only efficient but also provides very good estimates. However, since the problem of estimating the entropy of English text itself is not the main focus of this paper, we will discuss our estimates elsewhere.
4 Implementing the Context-Sorting Text Compression Algorithm

The context-sorting text compression algorithm [14] is an on-line data compression method, which can be regarded as a kind of symbol-ranking compressors [5]. It is important in that it connects the block-sorting compression method mentioned in the next section with Lempel–Ziv-type dictionary-based methods [14], [9]. Although the context-sorting compression algorithm is asymptotically optimal for data from a finite-order Markov source, its existing implementation is naive and quite slow. In our previous implementation [14], we maintained relexically sorted contexts using a binary search tree. We had to limit the length of context and to consume time proportional to that bounded length. These have prevented us from introducing more sophisticated codes into the coding stage.

In the context-sorting method, we enumerate previous contexts in the order of their similarities to the current context. Then, we give ranks to distinct symbols in accordance with the orders of their contexts. The next symbol is encoded as its actual rank. Figure 10 shows an example of giving ranks to symbol candidates. In the figure, we assume that we have already encoded an initial segment ending with ‘... to define’ and are going to encode its following symbol. In this example, if the next symbol is ‘m’ then it is encoded as rank 0. The space ‘ ‘ is encoded as rank 1, ‘d’ as rank 2, and so on. These ranks may be encoded by a fixed static code or an adaptive code. However, since the original implementation took much time in the ranking phase, it was difficult to use adaptive codes, which are generally slower than static ones. In our new implementation, we can combine an adaptive arithmetic code [2] with the prefix list. We no longer need to restrict the context length. The new implementation incorporating the prefix list runs more than ten times faster than the previous one with the bounded context length of 8 symbols.

Obviously, the context-sorting compression method stimulates the development of prefix list. We may compare the prefix list and the context-sorting compression method to two sides of the same coin. The essential component of

---

re-lexically sorted contexts

... new method
... is readable
... combine
... to define ?
... refine
... affine
... is linear
... compose

Candidates for the current symbol ‘?’ with their ranks:

0 1 2 3 4 5
m -> a -> d -> a -> b -> t -> ...

Fig. 10. Ranked candidates in the context-sorting compression algorithm.
the latter method is the calculation of symbol’s rank. Therefore, when we design an improvement of the prefix list, we should consider not only its construction speed but also the possibility of efficient ranking of symbol candidates.

Although the context-sorting compression method is basically a symbolwise algorithm, it can be extended to Ziv-Lempel-type phrase-based compression methods [14]. The HYZ compression method [9] mentioned in the introductory section includes such an extension. Another example is the ACB algorithm of Buyanovsky [11], whose primal version is essentially the same as LZ77 [16]. What most distinguishes ACB from other LZ77 variants is its method of specifying the position of the longest match. If the longest match in the previous text begins with $x_k$ in Fig. 5, where the current phrase begins with $s_{i+1}$, then the value of $k$ is encoded as the match position. Therefore, we can apply the prefix list to the ACB algorithm to calculate the value of $k$.

5 Other Applications

String matching (Finding the longest common suffix): The (exact) string matching problem is to find a string called the pattern in a longer string called the text. We interpret it as a problem of finding a position to insert the pattern into the re-lexicographically sorted list of prefixes of the text. If we wish to find an occurrence of ‘cabr’ in $S[1..9]$ given in (2), it is enough to find the reverse lexicographic relationship:

$$S[1..5] = 'yabre' \prec 'cabr' \prec 'yabrecab' = S[1..9],$$

where ‘$\prec$’ denotes reverse lexicographic order. In this case, we can immediately know that the pattern in question appears as $S[6..9]$. If we have another pattern ‘rabr’, then a similar relation

$$S[1..9] = 'yabrecab' \prec 'rabr' \prec 'yabr' = S[1..4]$$

holds. This time, there is no exact matching; instead the longest common suffix ‘abr’ can be found.

Thus, the problem of finding an occurrence of pat of length $m$ in text of length $n$ is conceptually the same as building a prefix list for

$$S[1..n + m + 1] = text \& pat,$$

where the symbol $\& \not\in \Sigma$ is a special delimiter that alphabetically precedes any symbol in $\Sigma$. Of course, there is no need for the actual insertion of prefixes ending in ‘&pat’. Actually, we first build a prefix list for the text. In order to expedite the matching process, we use an auxiliary array of pointers, which map symbols to nodes in the prefix list. The array element corresponding to a symbol $s$ includes a pointer $Q$ that points to the node such that

$$S[Q \uparrow pred \uparrow idx] \prec s = S[Q \uparrow idx].$$
Namely, the pointer $Q$ points to the leftmost node among nodes representing prefixes with the same last symbol $s$. This array of pointers can also be used in the course of constructing a prefix list in order to check whether the next symbol has appeared in the initial segment seen so far. If the array element corresponding to the first symbol of the pattern is a nil pointer, then we know that the pattern is not contained in the text. Otherwise, we proceed to a procedure similar to the insertion of the rest of the pattern. If the pattern is generated from the same Markov source as for the text, this procedure runs linearly in the pattern length.

**Suffix array construction:** The suffix array for a string $S[1..n]$ is an array of the indexes from 1 to $n$, specifying the lexicographic orders of the suffixes of $S[1..n]$ [8], [6].

A prefix list maintains all prefixes in reverse lexicographic order. Thus, the construction of a suffix array is straightforward if we apply our prefix list construction procedure to a reversed string. It is sufficient to sequentially copy the indexes of a prefix list into a suffix array by traversing the list via $suc$ links. Obviously, the conversion of a prefix list to a suffix array can be done in linear time.

In some applications of suffix arrays, information about the longest common prefixes (lcp) plays an important role [8]. The quantity $l_i$ of the $i$th node can be used as the lcp of consecutive elements of a resulting suffix array.

**Block-sorting data compression:** The block-sorting data compression algorithm of Burrows and Wheeler [3] has received considerable attention in anticipation that it may outperform the Lempel–Ziv codes. Its operation begins with a special sort procedure, called the $BW$ transform, which is followed by a sequential application of move-to-front heuristics and statistical encoding.

We can apply the prefix list to performing the BW transform. In our terms, the BW transform can be described in the following. Here, $H$ and $T$ denote the pointers to the list head and the list tail, respectively.

$$P \leftarrow H;$$

while ($P \neq T$) {
  if ($P\uparrow.next = \text{nil}$) output(‘$’) else output($S[P\uparrow.next.indx]$);
  $P \leftarrow P\uparrow.succ$;
}

This is not identical with the original definition of the transform but is essentially the same. The prefix list in Fig. 4 converts our sample string into

$$S'[1..10] = ybbrrac&ea,$$

which in turn is encoded by a move-to-front coder. We are omitting the second half of the algorithm; see [3] for more details.

The BW transform is reversible: we can reconstruct $S[1..n]$ from its transformed string $S'[1..n+1]$. In order to explain the reverse transformation, we add subscripts to indicate the position of each symbol in $S'[1..n+1]$. In the above example, the input into the reverse transformation is $S'[1..10] = y_1 b_2 b_3 a_4 a_5 a_6 c_7 & s_8 a_9 a_{10}$. First, we sort alphabetically the symbols in this string in a stable manner.
Then, we have
\[ S''[1..10] = a_8 \cdot a_6 \cdot \bar{a}_6 \cdot b_2 \cdot b_3 \cdot c_7 \cdot \bar{a}_4 \cdot \bar{r}_5 \cdot y_1. \] (14)

Second, we write
\[ \pi(i) = j, \quad 1 \leq i, j \leq n + 1 \] (15)
for the symbol \( S'[i] \) if it appears as the \( j \)th symbol \( S''[j] \) in \( S''[1..n + 1] \). In the present example, we have \( \pi(8) = 1, \pi(6) = 2, \pi(10) = 3 \), and so on. Then, beginning with \( S \leftarrow \varepsilon \) and \( i \leftarrow 1 \), we repeat
\[
S \leftarrow S \cdot S'[i]; \quad \text{or equivalently,} \quad i \leftarrow \pi(i); \quad S \leftarrow S \cdot S''[i]
\]
n times to recover the original string in \( S \).

The (forward) BW transform is more demanding than the corresponding inverse transform. Our prefix list performs the forward transform in linear time at least on a string that is drawn from a finite-order Markov source.

6 Conclusion

We have presented a conceptually simple data structure, called the prefix list. It is a linked-list representation of prefixes of a string sorted in reverse lexicographic order. It is quite similar to the suffix array in that lexicographic linear order is incorporated. While the suffix array has an off-line nature, a prefix list can be built in an on-line manner. This yields its characteristic applications. The prefix list provides a powerful tool to a class of context-based symbol-ranking data compression algorithms. We have also shown that the prefix list is applicable to other interesting problems.

Acknowledgments

I am grateful to Taro Yamagishi who implemented the proposed data structure and provided experimental results. Detailed comments by an anonymous referee should be acknowledged although they are not yet reflected on the paper.

References