Computing the Burrows-Wheeler Transform of a String and Its Reverse

Enno Ohlebusch\textsuperscript{1}, Timo Beller\textsuperscript{1}, and Mohamed I. Abouelhoda\textsuperscript{2,3}

\textsuperscript{1} Institute of Theoretical Computer Science, University of Ulm, 89069 Ulm, Germany
\{Enno.Ohlebusch,Timo.Beller\}@uni-ulm.de
\textsuperscript{2} Center for Informatics Sciences, Nile University, Giza, Egypt
mabouelhoda@yahoo.com
\textsuperscript{3} Faculty of Engineering, Cairo University, Giza, Egypt

Abstract. The contribution of this paper is twofold. First, we provide new theoretical insights into the relationship between a string and its reverse: If the Burrows-Wheeler transform (BWT) of a string has been computed by sorting its suffixes, then the BWT and the longest common prefix array of the reverse string can be derived from it without suffix sorting. Furthermore, we show that the longest common prefix arrays of a string and its reverse are permutations of each other. Second, we provide a parallel algorithm that, given the BWT of a string, computes the BWT of its reverse much faster than all known (parallel) suffix sorting algorithms. Some bioinformatics applications will benefit from this.

1 Introduction

The Burrows-Wheeler transform \cite{bzip} is used in many lossless data compression programs, of which the best known is Julian Seward’s bzip2. Moreover, it is the basis of FM-indexes that support backward search \cite{FM}. In some bioinformatics applications, one needs both the Burrows-Wheeler transform \texttt{BWT} of a string \texttt{S} and the Burrows-Wheeler transform \texttt{BWT\texttt{rev}} of the reverse string \texttt{S\texttt{rev}}. For example, in the prediction of RNA-coding genes \cite{RNA} or short read alignment \cite{short_read}, it is advantageous to be able to search in forward and backward direction. This bidirectional search requires \texttt{BWT} as support for backward search and \texttt{BWT\texttt{rev}} as support for forward search. Another example is \textit{de novo} sequence assembly based on pairwise overlaps between sequence reads. Simpson and Durbin \cite{de_novo} showed how an assembly string graph can be efficiently constructed using all pairs of exact suffix-prefix overlaps between reads. (Välimäki et al. \cite{approximate_overlap} provide techniques to find all pairs of approximate suffix-prefix overlaps.) To compute overlaps between reverse complemented reads, they build an FM-index for the set of reads and an FM-index for the set of \textit{reversed} reads.

The Burrows-Wheeler transform \texttt{BWT} of a string \texttt{S} is usually computed by sorting all suffixes of \texttt{S} (hence the suffix array of \texttt{S} is known). Of course, the Burrows-Wheeler transform \texttt{BWT\texttt{rev}} of the reverse string \texttt{S\texttt{rev}} can be obtained in the same fashion. However, because of the strong relationship between a string and its reverse, it is quite natural to ask whether \texttt{BWT\texttt{rev}} can be directly derived
from BWT—without sorting the suffixes of $S^{\text{rev}}$. In this paper, we prove that this is indeed the case. More precisely, we give an algorithm for this task that has $O(n \log \sigma)$ worst-case time complexity. (If needed, the suffix array $SA^{\text{rev}}$ of $S^{\text{rev}}$ can easily be obtained from $\text{BWT}^{\text{rev}}$; see e.g. [13].) Interestingly, essentially the same algorithm can be applied to obtain the Burrows-Wheeler transform of the reverse complement of a DNA sequence.

We further study the relationship between the lcp-array $\text{LCP}$ of $S$ and the lcp-array $\text{LCP}^{\text{rev}}$ of $S^{\text{rev}}$. To be precise, we prove that $\text{LCP}^{\text{rev}}$ is a permutation of $\text{LCP}$. Furthermore, we show that $\text{LCP}^{\text{rev}}$ can also be directly computed: by just one additional statement, it is possible to compute all irreducible lcp-values of $\text{LCP}^{\text{rev}}$ with the same algorithm, and the remaining reducible lcp-values of $\text{LCP}^{\text{rev}}$ can easily be derived from them.

In contrast to suffix sorting, our new algorithm is easy to parallelize. Experiments show that it is faster than all known (parallel) suffix sorting algorithms.

2 Preliminaries

Let $\Sigma$ be an ordered alphabet of size $\sigma$ whose smallest element is the so-called sentinel character $. In the following, $S$ is a string of length $n$ over $\Sigma$ having the sentinel character at the end (and nowhere else). For $1 \leq i \leq n$, $S[i]$ denotes the character at position $i$ in $S$. For $i \leq j$, $S[i..j]$ denotes the substring of $S$ starting with the character at position $i$ and ending with the character at position $j$. Furthermore, $S_i$ denotes the $i$th suffix $S[i..n]$ of $S$.

The suffix array $SA$ of the string $S$ is an array of integers in the range 1 to $n$ specifying the lexicographic ordering of the $n$ suffixes of the string $S$, that is, it satisfies $S_{SA[1]} < S_{SA[2]} < \cdots < S_{SA[n]}$; see Fig. 1 for an example. We refer to the overview article [14] for construction algorithms (some of which have linear run time). In the following, $ISA$ denotes the inverse of the permutation $SA$.

The suffix tree $ST$ for $S$ is a compact trie storing the suffixes of $S$: for any leaf $i$, the concatenation of the edge labels on the path from the root to leaf $i$ exactly spells out the suffix $S_i$. In the following, we denote an internal node $\alpha$ in $ST$ by $\overline{\omega}$, where $\omega$ is the concatenation of the edge labels on the path from the root to $\alpha$. A pointer from an internal node $\overline{\omega}$ to the internal node $\overline{\alpha}$ is called a suffix link; see [8] for details.

The Burrows and Wheeler transform [2] converts a string $S$ into the string $\text{BWT}[1..n]$ defined by $\text{BWT}[i] = S[SA[i] - 1]$ for all $i$ with $SA[i] \neq 1$ and $\text{BWT}[i] = $ otherwise; see Fig. 1. The permutation $LF$, defined by $LF(i) = ISA[SA[i] - 1]$ for all $i$ with $SA[i] \neq 1$ and $LF(i) = 1$ otherwise, is called LF-mapping. The LF-mapping can be implemented by $LF(i) = C[c] + Occ(c, i)$, where $c = \text{BWT}[i]$, $C[c]$ is the overall number of characters in $S$ which are strictly smaller than $c$, and $Occ(c, i)$ is the number of occurrences of the character $c$ in $\text{BWT}[1..i]$.

The lcp-array of $S$ is an array $\text{LCP}$ such that $\text{LCP}[1] = -1$ and $\text{LCP}[i] = |\text{lcp}(S_{SA[i-1]}, S_{SA[i]})|$ for $2 \leq i \leq n$, where $\text{lcp}(u, v)$ denotes the longest common prefix between two strings $u$ and $v$; see Fig. 1. It can be computed in linear time from the suffix array and its inverse; see [11,13,10,6]. A value $\text{LCP}[i]$ is called reducible if $\text{BWT}[i] = \text{BWT}[i - 1]$; otherwise it is irreducible.