From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization

R. Giancarlo, A. Restivo, M. Sciortino

University of Palermo, Dipartimento di Matematica ed Applicazioni, Via Archirafi 34, 90123 Palermo, Italy

Abstract

We introduce a combinatorial optimization framework that naturally induces a class of optimal word permutations with respect to a suitably defined cost function taking into account various measures of relatedness between words. The Burrows and Wheeler transform (bwt) (cf. [M. Burrows, D. Wheeler, A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation, 1994]), and its analog for labelled trees (cf. [P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Structuring labeled trees for optimal succinctness, and beyond, in: Proc. of the 45th Annual IEEE Symposium on Foundations of Computer Science, 2005, pp. 198–207]), are special cases in the class. We also show that the class of optimal word permutations defined here is identical to the one identified by Ferragina et al. for compression boosting [P. Ferragina, R. Giancarlo, G. Manzini, M. Sciortino, Boosting textual compression in optimal linear time, Journal of the ACM 52 (2005) 688–713]. Therefore, they are all highly compressible. We also provide, by using techniques from Combinatorics on Words, a fast method to compute bwt without using any end-of-string symbol. We also investigate more general classes of optimal word permutations, where relatedness of symbols may be measured by functions more complex than context length. For this general problem we provide an instance that is MAX-SNP hard, and therefore unlikely to be solved or approximated efficiently. The results presented here indicate that a key feature of the Burrows and Wheeler transform seems to be, besides compressibility, the existence of efficient algorithms for its computation and inversion.

Keywords: Burrows–Wheeler transform; Optimal word permutation; Suffix tree; Lyndon word

1. Introduction

The Burrows–Wheeler transform [4] (bwt for short) has changed the way in which fundamental tasks for word processing and data retrieval, such as compression and indexing, are designed and engineered (see e.g. [7,15,13]). Given a word $w$, over a finite alphabet $A$, bwt$(w)$ can be formally defined as follows: (A) Create a list of the cyclic shifts of $w$ (see Fig. 1(a)); (B) sort that list (see Fig. 1(b)); (C) apply the permutation resulting from the sorting step

$\star$ Partially supported by the Italian MIUR PRIN project “Automi e Linguaggi Formali: aspetti matematici ed applicativi”, by the Italian MIUR FIRB project “Bioinformatica per la Genomica e la Proteomica” and by MIUR FIRB Italy–Israel project “Pattern Matching and Discovery in Discrete Structures, with applications to Bioinformatics”.

* Corresponding author.

E-mail addresses: raffaele@math.unipa.it (R. Giancarlo), restivo@math.unipa.it (A. Restivo), mari@math.unipa.it (M. Sciortino).

0304-3975/ - see front matter © 2007 Elsevier B.V. All rights reserved.

doi:10.1016/j.tcs.2007.07.019
to \( w \). A more intuitive, and quite common description of the transform algorithm simply states to form a conceptual matrix of the cyclic shifts of \( w \), sort its rows and take the last column \( L \) (see Fig. 1(a) and 1(b) again).

One way or the other, \( \text{bwt} \) can be seen as a word permutation induced by a permutation applied to a list of words, namely its cyclic shifts presented in some suitable order. It is an invertible transform, i.e., we can recover \( w \) from \( \text{bwt}(w) \) and an index \( I \), giving the position of \( w \) in the sorted list of cyclic shifts. Since the decoding step is of limited interest here, we simply mention that we need \( I \) and \( \text{bwt}(w) \). Those latter two items are referred to as the \( L - F \) mapping (see Fig. 1(b)) and can be used for decoding.

In their original paper, Burrows and Wheeler [4] give an intuition why \( \text{bwt}(w) \), a particular permutation of a word \( w \), is more easily compressible than the word itself: symbols that are preceded (actually succeeded) by the same context are grouped together so that a relatively weak locally adaptive scheme, such as Move to Front [2], can achieve a compression ratio comparable to the best compression algorithms. Ferragina et al. [7] have investigated this remarkable property of the transform and formalized it in terms of compression boosting.

Clustering together symbols preceding a “common context” is a key feature in order to make the input highly compressible. That is, the relatedness of two words (“contexts”) is measured by the length of their longest common prefix. It seems natural to ask whether such a grouping is optimal according to some criterion and whether it displays some properties of interest in terms of data compression. Those questions lead us to consider a combinatorial optimization problem that we now state.

Given two words \( x \) and \( y \), let \( lcp(x, y) \) be the longest prefix they have in common. Let \( X = (x_1, \ldots, x_n) \) be a list of words. We are interested in permutations of \( X \), i.e. rearrangements of all words in the list \( X \), such that the longer the \( lcp \) of two words, the closer they should be in an “optimal” rearrangement. In order to avoid confusion, we refer to permutations of list of words as list permutations, while we refer to permutations within a word as word permutations or simply permutations. For instance, the permutation that, given an initial arrangement, sorts the cyclic shifts of a word \( w \) is a list permutation, while \( \text{bwt}(w) \) is a word permutation.

**Problem 1.** Find a permutation \( X_\pi = (x_{i_1}, x_{i_2}, \ldots, x_{i_n}) \) of the list \( X \) so that \( LCP(X_\pi) = \sum_{j=1}^{n-1} |lcp(x_{i_j}, x_{i_{j+1}})| \) is maximum. We refer to such permutations as \( lcp\text{-optimal} \) list permutation.

In Section 2 we show that a lexicographic sort of the list \( X \) is \( lcp\text{-optimal} \). Based on this, we derive \( \text{bwt} \) as a permutation induced by \( lcp\text{-optimal} \) list permutation of suitably chosen lists. For instance, given a word \( w \), \( \text{bwt}(w) \) is a word permutation induced by an \( lcp\text{-optimal} \) list permutation of the cyclic shifts of \( w \). The novelty with respect to what was known in the literature is that \( \text{bwt} \) is optimal according to a well defined measure of relatedness of words.

In Section 3 we introduce the notion of \( lcp\text{-optimal induced} \) word permutations. Those are word permutations induced by \( lcp\text{-optimal} \) list permutations of the cyclic shifts of the input word, \( \text{bwt} \) being the only studied case. We show that this class of word permutations coincides with the class of word permutations defined by Ferragina et al. [7]. They noticed that such word permutations work for compression boosting as well as \( \text{bwt} \). Basically, **Boosting** is an automatic procedure that allows us to transform a memoryless compressor into one that uses the “best contexts” in a word for compression. The Burrows and Wheeler transform plays a key role in compression boosting. However, Ferragina et al. noticed that \( \text{bwt} \) is a sufficient ingredient, but other word permutations may work as well. In fact, they defined a class of word permutations well suited for compression boosting, i.e., \( \text{bwt} \) is not the only word permutation that is useful for boosting. Starting from a different point of view, we find that those permutations are also optimal with respect to a well defined notion of relatedness of words.
One key result for boosting, implicit in the techniques presented in [7], states that one can partition \( \text{bwt}(w) \) in such a way that the total order-zero empirical entropy of those pieces can be bounded in terms of any higher order empirical entropy of the original word. An analogous result is presented as an observation in [13]. A consequence is that, once identified that partition of \( \text{bwt}(w) \), a good-order zero compression of the single pieces yields, in principle, compression of the original word bounded by higher order entropy of the original word. Here we extend that remarkable result to \( lcp \)-optimal induced word permutations. The Burrows–Wheeler transform provides a method able to compute a very special \( lcp \)-optimal induced word permutation that has two key properties: it is very fast to compute and it is easy to invert. An open problem is to determine whether there exist other \( lcp \)-optimal induced word permutations having the same features as \( \text{bwt} \).

Section 4 is devoted to describe a fast method to compute \( \text{bwt} \) without using any end-of-string symbol. Such a technique uses notions from Combinatorics on Words.

In Section 5 we address a general combinatorial optimization problem. In particular, we rephrase Problem 1, defined in terms of the cost function \( LCP(X_\pi) \), in very general terms by picking a generic function \( r \), not necessarily symmetric, quantifying the relatedness of two words and by defining a total cost function, analogous to \( LCP(X_\pi) \), to be optimized. We show that there exist similarity functions measuring the relatedness of words so that the corresponding instances of the general problem are MAX-SNP hard [16] and therefore unlikely to admit a polynomial time approximation scheme [1]. This result indicates that the efficiency of \( \text{bwt} \) strictly depends on the fact that the relatedness of two “contexts” is measured by their longest common prefix, and this leads to an easy to compute optimization problem.

2. \( lcp \)-Optimal list permutations

In this section we describe a solution to the Problem 1 introduced in Section 1. In particular, the following theorem states that finding an optimal solution is computationally easy.

Let \( A \) be a finite alphabet and let \( \leq \) be a linear order relation defined on the characters of \( A \), from which a lexicographic order relation on \( A^* \) can be induced. In what follows, for our examples, we will assume the standard alphabetic ordering.

**Theorem 2.** Let \( X_{\text{lex}} \) be any permutation of the list of words \( X \) corresponding to a lexicographic order of the words, either non-increasing or non-decreasing. \( X_{\text{lex}} \) is \( lcp \)-optimal.

**Proof.** We give a proof for the non-decreasing case, since the non-increasing one follows along the same lines. In order to prove the result, we need to introduce an elementary sorting step, referred to as basic. It is such that, when recursively applied to a list of words, one obtains a new list that is lexicographically sorted and with at least as good a value of the \( LCP \) function. The basic sorting step is defined as follows. Consider a list of words \( Y = \{y_1, y_2, \ldots, y_m\} \). Let \( \text{clcp}(Y) \) be the longest prefix common to all words in \( Y \). Let \( Y' \) be the list of words \( Y \) lexicographically and stably sorted according to the lexicographic order relation defined on \( A \) and taking into account only the prefixes of length \( |\text{clcp}(Y)| + 1 \) of each word in \( Y \). As a boundary condition, we assume that the empty symbol is the last symbol of each word and that it is smaller than any symbol in \( A \). For future reference, it is convenient to notice that the list \( Y' \) can be partitioned into at most \( |A| + 1 \) sublists, \( Y'_1, \ldots, Y'_t \) such that each \( Y_i \) collects all words in \( Y \) that have the same prefix of length \( |\text{clcp}(Y)| + 1 \).

We now show that \( LCP(Y) \leq LCP(Y') \). To this end, it is convenient to introduce two dummy words \( y_{\text{start}} \) and \( y_{\text{end}} \) such that they have no prefix in common with any of the words in \( Y \). We assume that \( y_{\text{start}} \) and \( y_{\text{end}} \) are added at the beginning and end, respectively, of both \( Y \) and \( Y' \). Moreover, we consider the cost functions \( LCP'(Y) = LCP(Y) + |\text{clcp}(y_{\text{start}}, y_1)| + |\text{clcp}(y_m, y_{\text{end}})| \) and \( LCP'(Y') = LCP(Y') + |\text{clcp}(y_{\text{start}}, y'_1)| + |\text{clcp}(y'_m, y_{\text{end}})| \) and show that the former is bounded by the latter. Since the contributions of \( y_{\text{start}} \) and \( y_{\text{end}} \) are equally zero in both functions, the claimed inequality follows. Let us denote by \( \circ \) the concatenation of two lists. For each word \( y \) in \( Y \circ y_{\text{end}} \), its contribution to the \( LCP'(Y) \) cost function is defined to be the length of the prefix it has in common with the word preceding it in the list. An analogous definition holds for words in \( Y' \circ y_{\text{end}} \).

Let \( y_1, \ldots, y_j \) be a maximal sublist of contiguous words in \( Y \) such that they have a prefix in common of length \( |\text{clcp}(Y)| + 1 \). Because of the stability of the sorting step, those words will be contiguous also in \( Y' \). In order to quantify the change in the cost functions caused by moving that sublist of words from where they occur in \( Y \) to their new position in \( Y' \), we need to consider only \( y_j \) since the contribution of all other words in the sublist is the
same to both cost functions. Its contribution to $LCP(Y)$ is $|clcp(Y)|$, i.e., $y_i$ is preceded by a word $y$ such that $|lcp(y, y)| = |clcp(Y)|$, else the sublist could not be of maximal length. However, its contribution to $LCP(Y')$ is greater than or equal to $|clcp(Y)|$, since it can be placed to the right of a word with which it has $clcp(Y)$ in common.

Consider the list $X_{opt}$, providing an optimal solution to Problem 1. We show that, by recursively applying the basic sorting step, we obtain a stable lexicographically sorted list $X'_{lex}$ that is optimal.

Apply to $X_{opt}$ the basic sorting step to obtain a new list $X'$ such that: (a) $LCP(X_{opt}) = LCP(X')$ (else $X_{opt}$ could not be optimal); (b) $X'$ is a lexicographically sorted version of $X_{opt}$, where the sorting is limited to prefixes of length $|clcp(X)| + 1$. By the properties of the basic sorting step outlined above, we can divide $X'$ in sublists $X'_1, \ldots, X'_t$ such that each $X'_i$ contains all words in $X$ having the same prefix $p_i$ of length $|clcp(X)| + 1$. We now apply the basic step recursively to each sublist $X'_i$, modified by deleting $p_i$ from all words in $X'_i$. If the resulting modified list is empty, we do nothing. Since at each recursive application of the basic sorting step the value of cost function of the whole list is still optimal, we have that $X'_{lex}$ is optimal.

In order to conclude the proof, we need to remove the constraint that $X'_{lex}$ is a stably sorted version of $X_{opt}$. It suffices to notice that, within runs of identical words in $X'_{lex}$, we can rearrange the occurrences of the words without increasing the value of the cost function. □

Given a word $w = a_1a_2\ldots a_\ell$, let $sh(w, i)$ be the word corresponding to the cyclic shift of $w$ starting at $i$, $1 \leq i \leq \ell$, and let $SH(w)$ be the list of those shifts, given in increasing order of $i$ and starting with $i = 2$.

Moreover, let $f$ be the mapping that associates each position $i$ of $w$, $1 \leq i \leq \ell$, with $sh(w, i + 1)$ where arithmetic is mod $\ell$. Notice that any permutation of the list $SH(w)$ induces, via $f$, a permutation of the word $w$. We refer to those latter as $lcp$-optimal induced word permutations. For example, letting $w = abraca$, $SH(w) = \{braca, racaab, acaabr, caabra, aabrac, abraca\}$.

The following corollary follows by Theorem 2 and the fact that the $\textbf{w}_{\text{rt}}$ algorithm produces a lexicographically sorted list of $SH(w)$.

**Corollary 3.** Consider the list of words $SH(w)$. The permutation $\sigma$ of $SH(w)$ produced by the $\textbf{w}_{\text{rt}}$ algorithm is $lcp$-optimal. Therefore, $\textbf{w}_{\text{rt}}(w)$ is an $lcp$-optimal induced word permutation.

Analogous results can be inferred for the transformation $\textbf{x}_{\text{bw}}$ defined for labelled trees and introduced in [8].

It is of some interest to cast Problem 1 as an optimization problem on graphs. Indeed, it naturally corresponds to finding a Hamiltonian path of maximum weight in an undirected, complete and weighted graph $G_X$, where the vertices are the elements of $X$ and each edge $(x_i, x_j)$ is weighted by $|lcp(x_i, x_j)|$. Fig. 2 gives an example of $G_X$. The formulation of Problem 1 in terms of graphs just given states that all and only the $lcp$-optimal list permutations correspond to Hamiltonian paths of maximum weight in $G_X$. Theorem 2 states that finding an optimal Hamiltonian path in this case is computationally easy. In general, i.e. when relatedness of two words is measured by functions other than the length of their $lcp$, that need not be the case, as shown in Section 5.
Fig. 3. Graph $G'_{SH(w)}$ where $w = abraca$

It is also of interest to consider the special case $X = SH(w)$. Then, we can use the mapping $f$ to obtain a new graph $G'_{SH(w)}$ from $G_{SH(w)}$, where each cyclic shift is substituted by the position of $w$ corresponding to it, via $f$. Fig. 3 gives the graph corresponding to the one given in Fig. 2. Hamiltonian paths of maximum weight in $G'_{SH(w)}$ correspond to lcp-optimal induced word permutation. Using Fig. 3, one can find examples of maximum cost Hamiltonian paths not corresponding to $bwt(w)$. In particular, the Hamiltonian paths of maximum cost $(536412)$, $(536142)$ and $(653412)$ correspond to the permutations $craaab$, $craaab$ and $acraab$, respectively, of $abraca$ while the transform is $caraab$. We also notice that $craaab$ gives a longer run of identical symbols than $caraab$. The former corresponds to the alphabet ordering $a < c < b < r$ while the latter to the standard lexicographic order. Therefore, the alphabet ordering one chooses to derive the transform may play a role in the actual compression of the original word, even though the list permutations from which they are induced are both lcp-optimal.

Note that some experimental papers have already addressed the problem of permuting the alphabet and looked at its impact in compressing the $bwt$ (cf. [5]).

3. lcp-Optimal induced permutations and word permutations realized by a suffix tree

Ferragina et al. [7] introduce the notion of permutations of a word $w$ that are realized by a suffix tree [9], showing that those permutations can be compressed to satisfy high order entropy bounds via boosting techniques. Here we show that all and only the lcp-optimal induced word permutation are realized by a suffix tree. That is, Ferragina et al. characterize compressible permutations in algorithmic terms. Here we show that exactly those permutations are optimal in terms of a very intuitive notion of relatedness of words.

As for $xbw$, preliminary studies [8] indicate that results analogous to the ones reported here must also hold. However, its characterization in terms of entropy of a tree is still under investigation.

Let $T_w$ be the lexicographically sorted suffix tree of a word $w$ [9]. That is, a visit of the leaves of $T_w$ from left to right gives the suffixes of $w$ in lexicographic order. We refer to it as the suffix tree for $w$. We can label each leaf with the symbol preceding the corresponding suffix in $w$. We call such symbols offsprings of the tree.

A permutation $w' = a_{i_1}a_{i_2} \cdots a_{i_k}$ of $w$ is realized by $T_w$ if $w'$ is the sequence of the left-to-right offsprings of the tree $T'$ obtained by permuting the sons of some internal nodes. In particular $bwt(w)$ is realized by $T_w$ by considering the identity permutation (see Fig. 4(a)).

Given a list of words $Y = (y_1, \ldots, y_m)$ of equal length, let $T$ be a compacted trie [9] storing those words. It has $c \leq m$ leaves, each corresponding to a distinct word in $Y$. We can assume that the words in $Y$ are distinct, so $T$ has exactly $m$ leaves.

The following lemma can be easily proved.
Fig. 4. (a) The suffix tree $T_w$ of the word $w = abraca$. We assume that a character $\$ is added to each suffix of $w$, so that no suffix can be a prefix of any other suffix. (b) The tree $T$ shows that the permutation $craab$ is realized by $T_w$.

**Lemma 4.** Consider a word $w$, the list $SH(w)$ and an lcp-optimal permutation $SH'(w)$. There exists a trie $T$ for $SH(w)$ such that visiting its leaves from left to right corresponds to the order in which the cyclic shifts of $w$ appear in $SH'(w)$.

**Lemma 5.** Consider a word $w$ and let $T$ be a compacted trie storing all of its cyclic shifts. Let $SH'(w)$ be the list of the cyclic shifts obtained by visiting the leaves of $T$ from left to right. $SH'(w)$ is lcp-optimal.

**Proof.** Let $u_1, \ldots, u_\ell$ be the Hamiltonian path in $G_{SH(w)}$ corresponding to $SH'(w)$. Reasoning by induction on the length of the Hamiltonian path, one can show the path is of maximum cost. Indeed, it holds that $u_1, \ldots, u_i$ is a maximum Hamiltonian path for the problem instance restricted to first $i$ words of $SH'(w)$, $1 < i \leq \ell$. □

Using the previous two lemmas and Theorem 2 we can deduce the following corollary.

**Corollary 6.** All and only the lcp-optimal induced permutations of $w$ are realized by $T_w$.

In Theorem 9 we extend to lcp-optimal induced word permutations the results shown in [7] stating a close relation between $\text{bwt}$ and compression boosting. In particular, one can partition $\text{bwt}(w)$ in such a way that the total order-zero entropy of those pieces can be bounded in terms of any higher order entropy of the original word. An analogous result is presented as an observation in [13]. The implication is that, once identified that partition of $\text{bwt}(w)$, a good-order zero compression of the single pieces yields, in principle, compression of the original word bounded by higher order entropy of the original word. In order to give the proof of the theorem, we need to recall some definition. Additional details can be found in [7].

Let $s$ be a word over the alphabet $A = \{a_1, \ldots, a_h\}$ and, for each $a_i \in A$, let $n_i$ be the number of occurrences of $a_i$ in $s$. Throughout this paper we assume that $n_i \geq n_{i+1}$. Moreover, all logarithms are taken to the base 2 and we assume $0 \log 0 = 0$. 
Let \( \{ P_i = n_i/|s| \}_{i=1}^h \) be the empirical probability distribution for the word \( s \). The zero-th order empirical entropy of the word \( s \) is defined as

\[
H_0(s) = - \sum_{i=1}^h P_i \log(P_i).
\]

(1)

For any length-\( k \) word \( w \), we denote by \( \overrightarrow{w} \) the word of single symbols following the occurrences of \( w \) in \( s \), taken from left to right. The \( k \)-th order empirical entropy of \( s \) is defined as:

\[
H_k(s) = \frac{1}{|s|} \sum_{w \in A^k} |\overrightarrow{w}| H_0(\overrightarrow{w}).
\]

(2)

Manzini [13] argued that, for highly compressible words, \( |s|H_k(s) \) fails to provide a reasonable bound to the performance of compression algorithms. For that reason, he introduced the zero-th order modified empirical entropy:

\[
H^*_0(s) = \begin{cases} 
0 & \text{if } |s| = 0 \\
(1 + \log |s|)/|s| & \text{if } |s| \neq 0 \text{ and } H_0(s) = 0 \\
H_0(s) & \text{otherwise}. 
\end{cases}
\]

(3)

Note that for a non-empty word \( s \), \( |s|H^*_0(s) \) is at least equal to the number of bits needed to write down the length of \( s \) in binary. The \( k \)-th order modified empirical entropy \( H^*_k \) is then defined in terms of \( H^*_0 \) as the maximum compression we can achieve by looking at no more than \( k \) symbols preceding the one to be compressed.\(^1\) Formally, let \( S_k \) be a set of factors of \( s \) having length at most \( k \). We say that the set \( S_k \) is a suffix cover of \( A^k \), and write \( S_k \preceq A^k \), if any word in \( A^k \) has a unique suffix in \( S_k \).

For any suffix cover \( S_k \), let

\[
H^*_k(s) = \frac{1}{|s|} \sum_{w \in S_k} |\overrightarrow{w}| H^*_0(\overrightarrow{w}).
\]

(4)

The value \( H^*_S_k(s) \) represents the compression we can achieve using the words in \( S_k \) as contexts for the prediction of the next symbol. The entropy \( H^*_k(s) \) is defined as the compression that we can achieve using a best possible suffix cover:

\[
H^*_k(s) = \min_{S_k \preceq A^k} H^*_S_k(s).
\]

(5)

In the following, we use \( S^* \) to denote a suffix cover for which the minimum of (5) is achieved. Therefore we write

\[
H^*_k(s) = \frac{1}{|s|} \sum_{w \in S^*_k} |\overrightarrow{w}| H^*_0(\overrightarrow{w}).
\]

(6)

Equivalently it is possible to define notions of entropies in which substitute \( \overrightarrow{w} \) with \( \overrightarrow{w} \) representing the sequence of symbols preceding the occurrences of \( w \). Such definitions are equivalent up to exchanging \( w \) with its reverse.

Let \( y \) be a factor of \( w \). The locus of \( y \) is the node \( \tau[y] \) of \( T_w \) that has associated the shortest word prefixed by \( y \). Notice that many words may have the same locus because a suffix tree edge may be labelled by a long factor of \( w \). For example, in Fig. 4(a), the locus of both \( ab \) and \( abraca\$ \) is the node reachable by the path labelled by \( ab\$ \). The definition of locus just given extends verbatim to any trie storing a set of words.

For any suffix tree node \( u \) in \( T_w \), let \( b\$\tau(w)_u \) denote the factor of \( b\$\tau(w) \) obtained by concatenating, from left to right, the symbols associated to the leaves descending from node \( u \).

\(^1\) Note that \( H_k \) is defined in terms of \( H_0 \) as the maximum compression we can achieve by looking at exactly \( k \) symbols preceding the one to be compressed. For \( H^*_k \) we instead consider \( H^*_0 \) and a context of at most \( k \) symbols. This is to ensure that \( H^*_k(s) \leq H^*_k(s) \) for any string \( s \). See [13] for details.
Given $T_w$, we say that a subset $L$ of its nodes is a leaf cover if every leaf of $T_w$ has a unique ancestor in $L$. Any leaf cover $L = \{u_1, \ldots, u_p\}$ naturally induces a partition of the leaves of $T_w$. Because of the relationship between those leaves and $\text{bwt}(w)$, it induces also a partition of $\text{bwt}(w)$.

Let $C$ denote the function which associates to every word $y \in \Sigma^*$, the positive real value

$$C(y) = \lambda |y| H^*_\mu(y) + \mu \quad (7)$$

where $\lambda$ and $\mu$ are positive constants. For any leaf cover $L$, we define its cost as:

$$C(L) = \sum_{y \in L} C(\text{bwt}(w)_y). \quad (8)$$

Notice that the definition of $C(L)$ is additive and, loosely speaking, accounts for the cost of individually compressing the factors of the partition of $\text{bwt}(w)$ induced by $L$.

Let $L_{\text{min}}$ be the leaf cover of $T_w$, which minimizes $C(L)$. We need to recall Lemmas 4.4 and 4.6 from [7] that, for conciseness, we restate as follows:

**Lemma 7 ([7]).** Let $L_{\text{min}}$ be an optimal leaf cover for the cost function $C$ defined by (7). For any $k \geq 0$ there exists a constant $g_k$ (depending also on the alphabet) such that, for any word $w$

$$C(L_{\text{min}}) \leq |w| H^*_k(w) + g_k.$$  

An optimal leaf cover can be computed in $O(|w|)$ time and $O(|w| \log |w|)$ bits of space.

We notice that all the definitions just given can be extended verbatim to any other suffix tree obtained by $T_w$ by permuting some subtree. We need the following lemma:

**Lemma 8.** Let $w$ be a word, $T_w$ be its suffix tree and $L$ be a leaf cover of $T_w$. Let $T$ be any other suffix obtained by permuting some subtree of $T_w$ and let $L'$ be the leaf cover of $T$ corresponding to $L$ under such a permutation. Then, $C(L) = C(L')$.

**Proof.** Let $u$ be a node in $L$ and let $u'$ be the corresponding node in $L'$. Notice that the subtrees rooted at $u$ and $u'$ can be transformed one into the other by rearranging the nodes. Therefore, letting $w'_u$ be the symbols of $w'$ assigned to the leaves of $T$ descending from $u'$, $w'_u$ must be a permutation of $\text{bwt}(w)_u$. But $H^*_k(w'_u) = H^*_k(\text{bwt}(w)_u)$, since the order-zero entropy is the same for a word and all of its permutations. Now the result follows, by the assumption and by the definition of cost of a leaf cover (8). □

It is convenient to extend the definition of the cost of a leaf cover (8) to arbitrary partitions of a word, as follows. Let $y$ be a word and let $P(y)$ be a partition of $y$ into factors, i.e., $P(y) = y_1 \cdots y_m$. Its cost is:

$$C(P(y)) = \sum_{y_i \in P(y)} C(y_i). \quad (9)$$

We have:

**Theorem 9.** Let $w$ be a word and $w'$ be an lcp-optimal induced word permutation of $w$. There exists a partition $P(w')$ of $w'$ such that, for any $k \geq 0$ there exists a constant $g'_k$ for which $C(P(w')) \leq \lambda |w| H^*_k(w) + \log_2 |w| + g'_k$. Moreover, an optimal partition can be computed in $O(|w|)$ time and $O(|w| \log |w|)$ bits of storage, given the suffix tree “realizing” $w'$.

**Proof.** Consider $L_{\text{min}}$, the optimal leaf cover for $T_w$ and let $T$ be the suffix tree “realizes” $w'$. Then, there must be a leaf cover $L'$ of this latter tree such that by Lemma 8, $C(L_{\text{min}}) = C(L')$. The result now would follow from Lemma 7 and the fact that a leaf cover induces a partition of $w'$, except that we have to identify $L'$ in the claimed time and space bounds, given $T$. But that is simple. Indeed, we explicitly build the transformation between $T_w$ and $T$. We can then find $L_{\text{min}}$ in the claimed time and space bounds (again by Lemma 7) and use the transformation on the trees to obtain $L'$. □
Notice that Theorem 9 says nothing about the actual compression of $w$. Indeed, it simply states that a good partition of any $\text{lcp-optimal induced}$ permutation $w'$ of $w$ can be found efficiently and with provably good bounds in terms of higher order entropy of $w$. Now, one could apply the boosting techniques in [7] in order to perform the actual compression of $w$, by compressing separately each piece of the “optimal” partition of $w'$. However, one could have a problem at decompression time. Indeed, even if one recovers $w'$ from the compressed pieces, $w'$ may not be easily invertible, i.e., it may not be easy to recover $w$ from $w'$. Actually, $\text{bwt}(w)$ is an easily invertible transform, i.e., given $\text{bwt}(w)$ and the position of $w$ within the sorted cyclic shifts of $w$, one can efficiently recover $w$ [4]. An open problem is determining whether there exist other $\text{lcp-optimal induced}$ word permutations having the same features as $\text{bwt}$.

4. Computation of pure BWT via Lyndon words

The most important factor in the speed of computing the Burrows–Wheeler transform of a word $w$ is the time taken to sort the cyclic shifts of $w$. In order to speed up the computation, an alternative definition of the Burrows and Wheeler transform of a word $w$ is given (cf. [4]). An end-marker symbol $\$ is added at the end of the word and it is also assumed that $\$ does not occur in the input alphabet and it is smaller than any symbol of the alphabet (with respect to lexicographic order relation). Then, one proceeds as in Section 1, except that the $\$ symbol is ignored in the final result. Fig. 5(a) and (b) provide an example. In order to avoid confusion, we refer to the transform defined in Section 1 as pure-$\text{bwt}$ and to the one just defined as $\text{bwt}$. In practical terms, there seems to be no difference between the transforms, except for the algorithm computing them.

Indeed, pure-$\text{bwt}$ is computed by sorting the cyclic shifts of $w$, as required in the original paper by Burrows and Wheeler [4]. As for $\text{bwt}$, it is computed by sorting only the suffixes of $w\$$, since the assumptions on end-marker symbol reduces the sorting of the cyclic shifts of $w\$ to the sorting of its suffixes. Notice that the computation of both pure-$\text{bwt}$ and $\text{bwt}$ can be done in linear time and space (cf. [17]); however the computation of $\text{bwt}$ is much faster in practice [4].

Although $\text{bwt}$ of a word $w$ is an approximation of pure-$\text{bwt}(w)$, Burrows and Wheeler recommend this latter algorithm since it avoids the sorting of long words, i.e. the cyclic shifts of $w$. Such a pragmatic recommendation is so important that $\text{bwt}$ is de facto the definition provided for the transform (see for instance [13]).

In this section we show that the computation of the pure transform can be reduced to the sorting of only the suffixes of $w$. The reduction takes linear time, uses $|w|$ read only memory cells and $|w| + c$, $c$ constant, work memory cells and does not use sorting at all. This is accomplished by using the self-evident (and most of the times overlooked) correspondence between Lyndon words [10], conjugacy classes and the pure transform. As a special case, we show that the introduction of the extra symbol $\$ in $w$ for the computation of $\text{bwt}$ is a clever way to obtain the reduction, even if the output produced is slightly different from the pure version of the transform.

Given a word $x$, let $\text{suff}(x)$ be the set of all of its suffixes. A set of words $X$ is prefix-free if no word in $X$ is a proper prefix of any other word in $X$. A word $v \in A^*$ is a factor of $x \in A^*$ if $x = uvw$, $u, w \in A^*$. Given two words $x$ and $y$, let $\text{lcp}(x, y)$ be their longest common prefix.

A word $v \in A^*$ is primitive if $v = w^k$ implies $v = w$ and $k = 1$. Notice that we are using a somewhat more constrained definition of primitivity with respect to the standard one in the literature [14]. It is well known that every
word \( v \in A^* \) can be written in a unique way as a power of a primitive word, i.e., there exists a unique primitive word \( w \) and a unique integer \( k \) such that \( v = w^k \). Let \( \text{period}(v) \) be an algorithm that, given in input \( v \), returns that integer \( k \geq 1 \). Several of such algorithms can be obtained via standard word matching tools [9] and they take linear time and use only \(|v| + c \) work memory cells, \( c \) constant.

Two words \( x, y \in A^* \) are conjugate if \( x \) is cyclic shift of \( y \), i.e., \( x = uv \) and \( y = vu \) for some \( u, v \in A^* \). It is easy to see that conjugacy is an equivalence relation. Moreover, for later use, we point out that, for primitive words, all words in the same conjugacy class are distinct [10]. A Lyndon word is a primitive word which is also the minimum in its conjugacy class, with respect to the lexicographic order relation. Let \( \min(v) \) be an algorithm that, given in input a primitive word, returns an integer \( j, 1 \leq j \leq |v| \), indicating at which symbol the minimum of its cyclic shifts starts. That is, it computes the Lyndon word corresponding to \( v \). An algorithm for this task that takes linear time and uses \(|v| + c \) work memory cells, \( c \) constant, and no sorting, can be found in [11].

We show how to compute \( \text{bwt}(v) \) for a word \( v \) by sorting only the suffixes of a related word \( y \). This latter word can be identified in linear time and no sorting. The following result states that when \( v \) is periodic, we can limit ourselves to compute \( \text{bwt}(v) \) of its primitive root. That allows us to restrict attention to primitive words.

**Proposition 10** ([12]). Let \( v = w^k \), for some integer \( k > 1 \), and \( \text{bwt}(w) = a_0a_1\ldots a_{\ell-1}. \) Then \( \text{bwt}(v) = a_0^k \ldots a_{\ell-1}^k. \)

Let \( A^*_p \) be the set of primitive words in \( A^* \). It is natural to define a set \( P \subseteq A^*_p \) such that, for any \( v \in P \), the permutation of \( 1, 2, \ldots, |v| \) induced by the lexicographic sort of its cyclic shifts is the same as the one induced by the lexicographic sort of its suffixes. Notice that for a word \( v \in P \) one has that \( \text{bwt}(v) = S - \text{bwt}(v) \). Therefore, the \( \text{bwt} \) for the words in \( P \) can be computed via a lexicographic sort of their suffixes. The characterization of \( P \) is of some combinatorial interest. To this end, we need to introduce some notation.

Given two words \( u \) and \( v \) in \( A^* \), if \( u \) is a prefix of \( v \), we denote by \( s(u, v) \) the corresponding suffix, i.e., the word \( z \) such that \( uz = v \). Similarly, if \( v \) is a suffix of \( u \), we denote by \( p(u, v) \) the corresponding prefix, i.e., the word \( z \) such that \( u = zv \).

**Lemma 11.** A primitive word \( w \) belongs to \( P \) if and only if, for any \( x, y \in \text{suff}(w) \), if \( x \) is prefix of \( y \) then \( p(w, x) < s(x, y) \cdot p(w, y) \).

**Proof.** We need two preliminary remarks: (a) if \( \text{suff}(x) \) is prefix-free then the condition of the lemma is trivially satisfied; (b) if \( x, y \in \text{suff}(w) \) and \( x \) is a prefix of \( y \), then the three words \( p(w, x), p(w, y) \) and \( s(x, y) \) are defined and the condition in the statement of the lemma has a meaning.

In order to prove the lemma, let \( v \) and \( v' \) be two different cyclic shifts of \( w \). This means that \( w = uv = u'v' \). We need to show that \( v < v' \) if and only if \( uu < v'v' \). In the non-trivial case that \( v \) is a prefix of \( v' \), by the condition in the statement of the lemma, we have \( uu = vp(w, v) < vs(v, v')p(w, v') = v'u' \).

**Lemma 12.** Let \( w \) be a Lyndon word. Then \( w \in P \). Moreover, there exist words in \( P \) that are not Lyndon.

**Proof.** Let \( w = uv = u'v' \) be a Lyndon word, where \( |v| \neq |v'| \). We show that \( v' < v \) if and only if \( v'u' < uu \), which implies \( w \in P \). This is enough since, as pointed out earlier, no two cyclic shifts of a primitive word can be equal.

Assume that \( v'u' < uu \). Let \( i \) the first position in which \( v'u' \) and \( uu \) have a different character. There are two different cases to consider:

1. If \( i < |v'| \), then it trivially follows that \( v' < v \). Otherwise, \( v' \) is a prefix of \( v \), so \( v' < v \).
2. If \( v' > |v'| \). Assume that \( i > |v'| \). So, \( uu = ydxu \) and \( v'u' = yczx \) where \( y, x, z \) are words in \( A^* \) and \( c, d \in A \) with \( c < d \). It follows that the cyclic shift \( yczv \) should be smaller that \( ydxv = uv = w \), contradicting the fact that \( w \) is a Lyndon word. When \( i \geq |v'| \), it trivially follows that \( v' < v \).

Assume now that \( v' < v \). We have again two cases:

1. \( v' \) is a prefix of \( v \). Then, \( uu = v'yv \). If \( v'u' \) is greater then \( uu \) then \( v'u' = v'xz \), where \( |x| = |y| \) and \( xz > yu \).

   So, the cyclic shift \( yuv' < xzv = u'v' = w \), contradicting the assumption that \( w \) is a Lyndon word.
2. \( v' \) is not a prefix of \( v \). It immediately follows that \( v'u' < uu \).

To conclude the proof, we show there exist words in \( P \) that are not Lyndon words. For instance, the word \( braac{ad} \) is not a Lyndon word. Yet, that word is in \( P \).
Theorem 13. Given a word \( v \), there exists a linear time algorithm that reduces the computation of \( \text{bwt}(v) \) to sorting only the suffixes of \( w \), where \( w \) is the primitive root of \( v \). It uses \( |v| + c \) work memory cells, \( c \) constant, and no sorting.

Proof. The reduction consists of the following steps. First compute the integer \( k \) such that \( v = w^k \), via algorithm period. Use algorithm \( \text{min}(w) \) to compute the Lyndon word \( y \) corresponding to \( w \) and then \( \text{bwt}(y) \) via a sort of its suffixes. The definition of \( \mathcal{P} \), Lemma 12 and the identity \( \text{bwt}(y) = \text{bwt}(w) \) guarantee correctness for the case \( k = 1 \). When \( k > 1 \) we also need to expand \( \text{bwt}(y) \) according to Proposition 10. □

So far, we have not discussed the decoding process; i.e., how to go from \( \text{bwt}(v) \) to \( v \). In order to invert the transform, we need to know the rank \( i \) of the word \( v \) in the lexicographic sort of its cyclic shifts (see [4] for details). We limit ourselves to discuss the case in which \( v \) is primitive, leaving its extension to the periodic case to the reader. The following well-known fact is useful (see [10]). Here we provide an alternative short proof:

Proposition 14. A word \( v \in A_p^* \) is Lyndon if and only if it is lexicographically smaller than all of its suffixes.

Proof. The if part is the only non-trivial one. Since \( v \) is Lyndon, it is the minimum among all of its cyclic shifts. By Lemma 12, it is in \( \mathcal{P} \) and the definition of that set concludes the proof. □

Now, in the reduction of Theorem 13, we know that the word \( y \) has rank one in the lexicographic sort of its suffixes, by Proposition 14, and therefore of its cyclic shifts, by definition of \( \mathcal{P} \) and Lemma 12. However, based on those facts and the output of algorithm \( \text{min} \), it is a simple exercise (left to the reader) to infer the rank of \( v \) needed for proper decoding.

Finally, we point out that the use of \$ as an end-marker symbol to compute \$-\( \text{bwt}(v) \) via a lexicographic sort of its suffixes is a very elegant way to avoid the reduction in Theorem 13. Indeed, \( \$, \) \( v \) is already a Lyndon word and therefore in \( \mathcal{P} \). As for decoding, \$ \( v \) has rank one in the sorted list of its suffixes (Proposition 14) and, from that, we can recover \( v \).

5. A general combinatorial optimization problem

In this section we consider a generalization of the Problem 1 defined in Section 1. Actually, Problem 1, defined in terms of the cost function \( \text{LCP}(X_{\pi}) \), can be rephrased in very general terms by picking a generic similarity function \( r \), not necessarily symmetric, quantifying the relatedness of two words and by defining a total cost function, analogous to \( \text{LCP}(X_{\pi}) \), to be optimized. We provide a formal definition:

Problem 15. Given a similarity function \( r \) and a list of words \( X \), find a permutation \( X_{\pi} = (x_{i_1}, x_{i_2}, \ldots, x_{i_n}) \) of the list \( X \) so that \( \text{COST}(X_{\pi}) = \sum_{j=1}^{n-1} r(x_{i_j}, x_{i_{j+1}}) \) is maximized.

We show that such a general problem is MAX-SNP hard [16], that is, no polynomial time approximation scheme exists for it, unless \( P = \text{NP} \) [1]. The hardness result is proved by considering the following similarity function that measures the overlap of two words. Given two words \( x \) and \( y \), let \( \text{ov}(x, y) \) be the longest suffix of \( x \) equal to a prefix of \( y \). Remark that \( \text{ov}(x, y) \) is defined if \( x \) and \( y \) are not internal factors one of the other. So, in the following Problem 16 we assume that the list \( X \) of words is such that no element of \( X \) is an internal factor of another element of \( X \).

Problem 16. Find a permutation \( X_{\pi} = (x_{i_1}, x_{i_2}, \ldots, x_{i_n}) \) of the list \( X \) so that \( \text{OV}(X_{\pi}) = \sum_{j=1}^{n-1} |\text{ov}(x_{i_j}, x_{i_{j+1}})| \) is maximum. We refer to \( X_{\pi} \) as an \( \text{OV-optimal} \) list permutation.

We now show the connection of Problem 16 with Shortest Common Superstring [9]. Given a set \( X \) of words, a common superstring of \( X \) is a word \( w \) such that each word \( x \) in \( X \) is a factor of \( w \). The Shortest Common Superstring problem is to find a common superstring for \( X \) of minimal length.

Given two words \( x \) and \( y \), denote by \( \text{pref}(x, y) \) the prefix of \( x \) such that \( x = \text{pref}(x, y)\text{ov}(x, y) \), and denote by \( \text{suff}(x, y) \) the suffix of \( y \) such that \( y = \text{ov}(x, y)\text{suff}(x, y) \).

Given now a list \( X \) of words and a permutation \( X_{\pi} = (x_{i_1}, x_{i_2}, \ldots, x_{i_n}) \) of the list \( X \), we define the superstring \( x_{\pi} = (x_{i_1}, x_{i_2}, \ldots, x_{i_n}) \) to be the word \( \text{pref}(x_{i_1}, x_{i_2}) \cdots \text{pref}(x_{i_{n-1}}, x_{i_n})\text{suff}(x_{i_{n-1}}, x_{i_n}) \). That is, \( x_{\pi} \) is the shortest word such that \( x_{i_1}, x_{i_2}, \ldots, x_{i_n} \) appear in order in that word. One has that:

\[ |x_{\pi}| = \sum_{j=1}^{n} |x_{i_j}| - \sum_{j=1}^{n-1} |\text{ov}(x_{i_j}, x_{i_{j+1}})|. \]
So the problem of finding a permutation \( \pi \) for which the word \( x_\pi \) has minimal length is equivalent to the problem of finding a permutation that maximizes \( OV(X_\pi) \). Hardness result for Shortest Common Superstring is a classic in the literature (see [3]) and we have the following

**Theorem 17** ([3]). **Problem 16 is MAX-SNP hard.**

The solution of the general problem addressed in this section depends on the function \( r \). In particular, we have shown that, by choosing \( r(x, y) = lcp(x, y) \), the problem can be solved in polynomial time, whereas, by choosing \( r(x, y) = ov(x, y) \) the problem is MAX-SNP hard. However, there are similarity functions \( r \), closely related to the ones defined earlier, and for which the computational complexity of the corresponding optimization problem is open.

Consider for instance the function \( lcf(x, y) \) that denotes the longest factor that appears both in the word \( x \) and in the word \( y \). By using such a function one can state the following problem.

**Problem 18.** Given a list of words \( X \) over the alphabet \( A \), find a permutation \( X_\pi = (x_{i_1}, x_{i_2}, \ldots, x_{i_n}) \) of the list \( X \) so that \( LCF(X_\pi) = \sum_{j=1}^{n-1} lcf(x_{i_j}, x_{i_{j+1}}) \) is maximized.

It is an open question whether there exists an efficient algorithm solving Problem 18.

The notion of relatedness of words can also be naturally formulated in terms of distances. We now briefly discuss word permutations in regard to some simple distance functions.

Following Choffrut [6], we define two distance functions on words: \( d_f(x, y) = |x| + |y| - 2|lcf(x, y)| \) and \( d_p(x, y) = |x| + |y| - 2|lcp(x, y)| \). Problem 15 becomes the following:

**Problem 19.** Find a permutation \( X_\pi = (x_{i_1}, x_{i_2}, \ldots, x_{i_n}) \) of the list \( X \) so that \( D_f(X_\pi) = \sum_{j=1}^{n-1} |d_f(x_{i_j}, x_{i_{j+1}})| \) is minimum. We refer to \( X_\pi \) as a \( d_f \)-optimal list permutation.

We can define an analogous problem for the \( d_p \) distance. The computational complexity of Problem 19 for both distance \( d_f \) and \( d_p \) is open. In particular, it is very easy to show that an analogous of Theorem 2 cannot hold. Indeed, let \( X = \{a, c, bb\} \) and consider the graph \( G_X \), defined in Section 2, where its weights are changed from \( lcp \) to distance \( d_p \). Note that in this case the Hamiltonian paths corresponding to a lexicographic sort of the list \( X \) are not necessarily of minimum cost. However, for the important special case of a list composed of words of equal length, an analogue of Theorem 2 holds.

**Theorem 20.** Let \( X \) be a list of words having equal length and let \( X_{\text{lex}} \) be any permutation of the list \( X \) corresponding to a lexicographic order of the words, either non-increasing or non-decreasing. \( X_{\text{lex}} \) is \( d_p \)-optimal.

**Proof.** It is sufficient to notice that, since all words in \( X \) are of equal length, we work with a distance \( d'(u, v) = -|lcp(u, v)| \). So, the result comes from Theorem 2 by substituting maximum with minimum. \( \square \)

It is not known whether, in the special case of words having the same length, the Problem 18 become easier to solve. So, the Problem 19 is open even in the case of equal word length.

We point out that, based on Theorem 20, Corollary 3 can be rephrased in terms of distance.

6. Conclusions and open problems

We have introduced the notion of \( lcp \)-optimal list permutations. We have also shown that those permutations allow us to derive \( bwt \) as permutation induced by them. We have also studied \( lcp \)-optimal induced word permutations. The \( bwt \) is the only studied case in this class. We have also shown that those permutations correspond to the ones identified by Ferragina et al. for compression boosting. Moreover, we have also discussed some generalizations of \( lcp \)-optimal list permutations, by stating a general combinatorial optimization problem. The major open problem stemming from this research is to establish whether \( bwt \) is the unique highly compressible, easily computable and invertible optimal word permutations, among the ones characterized in Section 3.
References