A Bijective String Sorting Transform

Joseph (Yossi) Gil* David Allen Scott ‡
Google, Inc.† El Paso, Texas
Haifa, Israel USA

October 29, 2018

Abstract

Given a string of characters, the Burrows-Wheeler Transform rearranges the characters in it so as to produce another string of the same length which is more amenable to compression techniques such as move to front, run-length encoding, and entropy encoders. We present a variant of the transform which gives rise to similar or better compression value, but, unlike the original, the transform we present is *bijective*, in that the inverse transformation exists for all strings. Our experiments indicate that using our variant of the transform gives rise to better compression ratio than the original Burrows-Wheeler transform. We also show that both the transform and its inverse can be computed in linear time and consuming linear storage.

1 Introduction

Informally, the famous *Burrows-Wheeler Transform* (BWT) [**?**] can be described as follows.

> *Given a string α of length n, generate the n cyclic rotations of α, and then sort these. By picking the last character of these sorted strings we obtain a string $\mathsf{BWT}(\alpha)$, the Burrows-Wheeler transform of α.*

*yogi@cs.Technion.ac.IL

†Work done while being on sabbatical leave from the Technion—Israel Institute of Technology.

‡David_a_Scott@email.com

BWT has become very popular for text compression application, because of two important properties. *First*, if α is textual, then $\mathsf{BWT}(\alpha)$ tends to have long runs of identical characters, which makes $\mathsf{BWT}(\alpha)$ more amenable to compression techniques such as run-length-encoding and move-to-front [**?**]. Consider for example an innocent phrase such as "*now is the time for the truly nice people to come to the party*", in which there are no runs of consecutive identical characters. In contrast, the Burrows-Wheeler transform of this phrase becomes

`oewyeeosreeeepi␣mhchlmhp␣tttnt␣puio␣yttcefn␣␣␣ooati␣␣␣␣␣␣␣␣rrolt`

in which 24 characters participate in eight such runs, the longest of which comprises seven repetitions of the space character. To see how these seven spaces are brought together, note that the phrase has seven words beginning with "*t*", and hence seven of its rotations begin with "*t*". When sorted, these seven rotations become neighbors, and since the last character in these seven is space, seven such spaces occur consecutively in the transform.

The *second* important property of the transform is the (rather surprising) fact that it is invertible in the following sense: Given $\mathsf{BWT}(\alpha)$, it is possible to efficiently generate all rotations of α.

The main issue with the inversion that concerns us here is that, by definition, $\mathsf{BWT}(\alpha) = \mathsf{BWT}(\alpha')$ for every α' which is a rotation of α. Therefore, with the absence of additional information, regenerating α given $\mathsf{BWT}(\alpha)$ is impossible.

To make decompression possible, compression algorithms must therefore store not only the compressed representation of $\mathsf{BWT}(\alpha)$, but also the *rotation index*, an integer i, $0 \leq i < n$, of α in the sorted list of its cyclic rotations. A less "pure" alternative is to append a newly introduced end-of-string character $\square$ to α, and then compute $\mathsf{BWT}(\alpha\square)$. String α is then chosen as the rotation in which $\square$ comes last.

This issue, together with a simple counting argument, shows that $\mathsf{BWT}^{-1}(\cdot)$, the inverse Burrows-Wheeler transform, cannot be defined for all strings. In fact, if η is a string of length n selected at random, then $\mathsf{BWT}^{-1}(\eta)$ is defined with probability $1/n$.

A natural question is *whether there exists a similar transform which is truly invertible*, in the sense that the transformed string uniquely identifies the original. In this paper, we answer this question affirmatively by describing a bijective, string sorting transform $\mathcal{S}(\cdot)$, which is similar to the Burrows-Wheeler transform in that it tends to bring identical characters together. In fact, in many cases, the output is quite similar to that of the BWT transform. For example, applying $\mathcal{S}$ to the above phrase yields a string which is different in only six locations:

yoeyeeosreeeepi␣mhchlmhp␣tttnt␣puio␣wttcefn␣␣ooati␣␣␣␣␣␣␣rrotl

Moreover, the $\mathcal{S}$ transform features in this case the same number of runs of identical characters and the same number of identical characters participating in these runs. Our experimental results indicate that compression revolving around $\mathcal{S}$ tends to perform (slightly better) than BWT based compression.

Consider for example, Table 1 which compares the performance of the $\mathcal{S}$ transform with that of the BWT transform when used for compression the famous Calgary corpus, a collection of 18 files which serves as the de-facto standard for benchmarking compression algorithms. In preforming the measurements, we first used Nelson's reference implementation [**?**], which carries out compression in five steps: *(i)* initial run-length-encoding, *(ii)* Burrow-Wheeler transformation *(iii)* move-to-front *(iv)* yet another run-length-encoding *(v)* arithmetical encoding [**?**]. We then repeated the same steps, substituting $\mathcal{S}$-transform for BWT in step *(ii)*. (In this particular experiment, we relied on the bit- rather than byte- representation of the data. The results for byte based compression are similar [**?**]).

As can be seen in the table, using the $\mathcal{S}$-transform, improves compression for all files except for GEO. The gain in compression ratio is about 1%. Note this gain is much greater than what can be attributed to the saving due due bijectivity, that is, the elimination of the end-of-string character, or the rotation index: Even in the PROGL file, in which the relative gain of the $\mathcal{S}$-based compression is the smallest, the size saving is of almost 200 bytes.

Other than better compression, $\mathcal{S}$ offers several other advantages over BWT. First, there is no need to store an end-of-string marker, nor the rotation index, in applying the Burrows-Wheeler transform in loseless compression algorithms. This advantage is prominent especially if the transform is used for very short texts, e.g., in transforming separately each line in a text file, or each field in database. Second, since the algorithm is bijective, it is more adequate for application in which the compressed data is encrypted—a non-bijective transform necessarily reveals information to the attacker. Finally, some may appreciate the elegance in bijectiveness and in the details of the definition of the transform.

For the impatient reader, an informal (and imprecise) description of the $\mathcal{S}$ transform is as follows:

> *Break α into sub-strings by successively selecting and removing its "smallest" suffix. Generate the rotations of each such sub-string, and sort all these rotations together. The transform $\mathcal{S}$ is then obtained by taking the "last" character of this sorted list.*

File	Size	BWT-compression		$\mathcal{S}$-compression		Gain	
		bytes	ratio	bytes	ratio	absolute	relative
BIB	111,261	32,022	28.78%	31,197	28.04%	0.74%	2.58%
BOOK1	768,771	242,857	31.59%	235,913	30.69%	0.90%	2.86%
BOOK2	610,856	170,783	27.96%	166,881	27.32%	0.64%	2.28%
GEO	102,400	66,370	64.81%	66,932	65.36%	-0.55%	-0.85%
NEWS	377,109	135,444	35.92%	131,944	34.99%	0.93%	2.58%
OBJ1	21,504	12,727	59.18%	12,640	58.78%	0.40%	0.68%
OBJ2	246,814	98,395	39.87%	94,565	38.31%	1.55%	3.89%
PAPER1	53,161	19,816	37.28%	18,931	35.61%	1.66%	4.47%
PAPER2	82,199	28,084	34.17%	27,242	33.14%	1.02%	3.00%
PAPER3	46,526	18,124	38.95%	17,511	37.64%	1.32%	3.38%
PAPER4	13,286	6,047	45.51%	5,920	44.56%	0.96%	2.10%
PAPER5	11,954	5,815	48.64%	5,670	47.43%	1.21%	2.49%
PAPER6	38,105	14,786	38.80%	14,282	37.48%	1.32%	3.41%
PIC	513,216	59,131	11.52%	52,406	10.21%	1.31%	11.37%
PROGC	39,611	15,320	38.68%	14,774	37.30%	1.38%	3.56%
PROGL	71,646	18,101	25.26%	17,916	25.01%	0.26%	1.02%
PROGP	49,379	13,336	27.01%	13,010	26.35%	0.66%	2.44%
TRANS	93,695	22,864	24.40%	22,356	23.86%	0.54%	2.22%
Total	3,251,493	980,022	30.14%	950,090	29.22%	0.92%	3.05%
Median	76,923	21,340	36.60%	20,644	35.30%	0.94%	2.58%

Tab. 1: Performance of BWT-based compression $\mathcal{S}$-based compression of the Calgary corpus.

Missing pieces in the above description include: the exact definition of the manner in which suffixes are compared, the specification of the order relation between rotations of different length, elaborating the meaning of the phrase "last" character. More importantly, we will also need to explain why this, seemingly arbitrary process, is reversible.

The transform $\mathcal{S}$ was discovered by the second author but remained unpublished. Unfortunately, his announcements on Internet Usenet groups such as `comp.compression` on December 2007 were received with great skepticism regarding issues including feasibility, correctness, complexity and utility. Here we formalize and describe the algorithm in detail, prove its correctness, provide a linear time and space implementation, compare to related word, and discuss extensions ad generalizations.

Outline The remainder of this article is structured as follows. Section 2gives some basic notations and demonstrates these in a precise definition of the algorithm for implementing Burrows-Wheeler transform. Then, the reader is reminded of the linear time algorithm for inverting the transform. We use the description of these two algorithms next, in Section 4 for defining the $\mathcal{S}$ transform, and in Section 5, the algorithm for implementing its inverse. In Section 6 we explain why the algorithm for computing $\mathcal{S}^{-1}$ is indeed correct. Section 7 concludes

2 Preliminaries I: The Burrows-Wheeler Transform

This section serves as a reminder of the details of the Burrows-Wheeler transform. It also sets up some definitions to be used later. Let Σ be an ordered set, the *alphabet* of *characters*, and let Σ^+ be the set of finite non-empty strings of characters chosen from Σ. For a string $\alpha \in \Sigma$, let $|\alpha|$, the *length* of α, be the number of characters in α. The order relation in Σ is extended to a total order of strings of *equal length* in Σ^+, in the usual lexicographical way. At this stage, we leave the comparison of strings of non-equal length unspecified.

We will treat strings as arrays in the C [**?**] programming language, so $\alpha[0]$ shall denote the first character of α, $\alpha[1]$ its second character, etc. Further, let $\alpha[-1]$ denote the last character of α, $\alpha[-2]$ its penultimate character, and more generally, for $i \geq |\alpha|$ or $i < 0$, let $\alpha[i] \equiv \alpha[i \bmod |\alpha|]$.

For strings $\alpha, \beta \in \Sigma$, let $\alpha\beta$ denote the string obtained by their concatenation; we say that α (respectively $\beta >$) is a *prefix* (respectively a *suffix*) of this concatenation. For an integer $m > 0$ let α^m denote the string obtained by concatenating α onto itself m times.

For $\alpha \in \Sigma^+$ and an integer $0 \leq m \leq |\alpha|$ let $\alpha(m)$ denote the m^{th} *rotation* of α, that is, the string obtained by removing the first m characters.of α and adding these at its end. More precisely, if $\alpha = \beta\gamma$ and $|\alpha| = m$, then $\alpha(n) \equiv \gamma\beta$. We extend this definition for all $m \in \mathbb{Z}$ by the equivalence $\alpha(m) \equiv \alpha(m \bmod |\alpha|)$. For example, if $\alpha =$ `tartar`, then $\alpha(1) = \alpha(4) = \alpha(-2)) =$ `artarta` and $\alpha(2) = \alpha(5) = \alpha(-1) =$ `rtarta`. We have

$$\alpha(i)[j] = \alpha[i + j] \tag{1}$$

for all $\alpha \in \Sigma^+$ and $i, j \in \mathcal{Z}$.

Algorithm 2.1 describes, using these notations, the first step of the Burrows-Wheeler transform, that is, the generation of the list of rotations of a given string.

Algorithm 2.1 CyclicRotations(α) *// Return the set of all cyclic rotations of $\alpha \in \Sigma^+$.*

1: **let** $n \equiv |\alpha|$
2: **For** $i = 0, \ldots n-1$ **do**
3: $\quad R \leftarrow R \cup \{\alpha(i)\}$
4: **Return** R

Algorithm 2.1 requires $\mathcal{O}(n)$ time and storage: Assuming that α is allocated in immutable storage, then each of $\alpha(i)$ can be represented by a triple of scalars: the address of the first character of α, the index i, and the length $n = |\alpha|$. Henceforth, we tacitly assume this triple based representation.

The second step of the BWT transformation can now be described concisely as depicted by Algorithm 2.2.

Algorithm 2.2 Last(R) *// Given a set $R \subset \Sigma^+$, return the string composed of the last character // of each of the members of R, enumerated in lexicographical order.*

1: **let** $n \equiv |R|$
2: **let** η be an uninitialized string of length n
3: **For** $i = 0, \ldots, n-1$ **do**
4: $\quad$ **let** $\alpha \equiv \min R$
5: $\quad \eta[i] \leftarrow \alpha[-1]$
6: $\quad R \leftarrow R \setminus \{\alpha\}$
7: **Return** η

Note that the algorithm is tantamount to sorting the input set R. If R has n elements, then this sorting can be done in $\mathcal{O}(n \log n)$ string comparisons. Each such comparison may require $\mathcal{O}(n)$ character comparisons, leading to an $\mathcal{O}(n^2 \log n)$ implementation. Yet, as Giancarlo, Restivo and Sciortino [**?**] observe, in the case that R is indeed a set of rotations, then the sorting can be done in $\mathcal{O}(n)$ time, by reduction to the problem of sorting the suffixes of a given string, which is known to be linear time.

Functions CyclicRotations (Algorithm 2.1) and Last (Algorithm 2.2) are combined in Algorithm 2.3, which realizes the Burrows-Wheeler transform. Evidently, the algorithm requires linear time and space.

Algorithm 2.3 $\mathsf{BWT}(\alpha)$ // *Given a string* $\alpha \in \Sigma^+$*, return its Burrows-Wheeler transform.*

1: $R \leftarrow \mathsf{CyclicRotations}(\alpha)$
2: **Return** $\mathsf{Last}(R)$

3 Preliminaries II: Inverting the Burrows-Wheeler Transform

Let $\alpha \in \Sigma^+$ be a string of length n, and let $\eta = \mathsf{BWT}(\alpha)$, then, examining Algorithm 2.3, we see that it effectively defines a permutation π, such that $\eta[i] = \alpha[\pi(i)]$ for $i = 0, 1, \ldots, n-1$. Given η, we would like to generate the inverse permutation π^{-1}. Unfortunately, as explained above, this is impossible.

Instead, the algorithm conceived by Burrows and Wheeler produces from η a permutation ϑ from which a *rotation* of α can be generated. The defining property of permutation ϑ is

$$\forall k \bullet (0 \leq k < n) \wedge (k = \pi(i)) \implies \vartheta(k) = \pi(i - 1 \bmod n). \tag{2}$$

That is, having matched a position i in α with a position k in η, we can match, $i - 1 \bmod n$, the cyclically preceding location in α with the position $\vartheta(k)$ in η. Let us therefore define a permutation ϱ by applying ϑ upon itself successively, *i.e.*,

$$\varrho(i) = \begin{cases} \vartheta(0) & i = 0 \\ \vartheta(\varrho(i-1)) & i > 0. \end{cases}.$$

Applying ϱ to reorder the characters in η generates, last to first, the characters of some rotation of α. More precisely, Burrows and Wheeler's inversion procedure generates the string β, defined by $\beta[j] = \eta[\varrho(n-j)]$, which is not only *a* cyclic rotation of α, it is the *lexicographically smallest* such rotation.

The full process of generating β from ϑ is described in Algorithm 3.1.

The algorithm is rather straightforward, except for Line 3, which initiates the threading process from the first character for η. By doing that, we ensure that the smallest rotation of α is returned. To see that, observe that the last character of this smallest rotation is the one mapped by the transform to $\eta[0]$. In mapping back this character to the last character of the output, as done in the first time Line 5 is executed, we ensure that the we generate precisely this rotation.

Algorithm 3.1 $\mathsf{Thread}(\eta, \vartheta)$ *// Given the transform* $\eta = \mathsf{BWT}(\alpha)$*, and the cyclic permutation* ϑ*,*
// return the string β*, the lexicographically smallest rotation of* α*.*

1: **let** $n \equiv |\eta|$
2: **let** β be an uninitialized string of length n
3: $k \leftarrow 0$
4: **For** $i = n-1, n-2, \ldots, 0$ **do** *// fill in* β*, last to first*
5: $\quad \beta[i] \leftarrow \eta[k]$
6: $\quad k \leftarrow \vartheta(k)$
7: **Return** β

Concentrate on the sorted list of the rotations $\alpha(i)$, which we will denote by L. The following two lemmas establish the means for generating the permutation ϑ from η.

Lemma 1. *If for some* i*,* $0 \leq i < n$*, rotation* $\alpha(i+1)$ *occurs at position* k *in* L*, while* $\alpha(i)$ *occurs at position* j *in it, then* $j = \vartheta(k)$*.*

Proof. We have that $\eta[k] = \alpha(i+1)[-1] = \alpha[i]$, and $\eta[j] = \alpha(i)[-1] = \alpha[i-1]$. Thus, if we knew that $\eta[k]$ is mapped to a certain position in β, we will be able to conclude that $\eta[j]$ is mapped to the cyclically previous position in β. □

Observe that Lemma 1 does not require the knowledge of i. All we need to know is that the rotation at position k is obtained by omitting the first character of the rotation at position j, what is called by Burrows and Wheeler a *match* between j and k.

Lemma 2. *For an arbitrary character* $c \in \Sigma$*, consider the sorted list* $\alpha(i_0), \ldots, \alpha(i_{\ell-1})$ *of those rotations* $\alpha(i)$*,* $0 \leq i < n$*, for which* $\alpha(i)[-1] = c$ *(that is, the rotations which correspond to occurrences of* c *in* η*). Then, the list* $\alpha(i_0 - 1), \ldots, \alpha(i_{\ell-1} - 1)$ *is also sorted. Moreover, this list occurs consecutively in* L*.*

Proof. Since the first character of each of the rotations $\alpha(i_0-1), \ldots, \alpha(i_{\ell-1}-1)$ is c, we can rewrite these as $c\alpha(i_0), \ldots, c\alpha(i_{\ell-1})$. This list is sorted since we assumed that $\alpha(i_0), \ldots, \alpha(i_{\ell-1})$ are sorted. Further, the elements of this list occur consecutively in L since they all begin with c and no other rotation begins with c.. □

Lemma 2 provides the means for matching the location of a rotation $\alpha(i)$ with the location of the rotation $\alpha(i-1)$. To understand the process, consider first the case that c is the smallest character occurring in η, and that it is

found at locations $k_0, \ldots, k_{\ell-1}$ in it. We know that there are some $i_0, \ldots, i_{\ell-1}$, such that $\alpha(i_0)[-1] = \alpha(i_1)[-1] = \cdots = \alpha(i_{\ell-1})[-1] = c$. Also, the rotations $\alpha(i_0), \ldots, \alpha(i_{\ell-1})$ are sorted into into locations $k_0, \ldots, k_{\ell-1}$. Although we do know the values $i_0, \ldots, i_{\ell-1}$, we can use Lemma 2 to infer the locations of the "preceding" rotations $\alpha(i_0 - 1) \ldots, \alpha(i_\ell - 1)$: By this lemma, these must occur in L together and at the same order. Since c is the smallest character in η, we can infer that these preceding rotations occur precisely at locations $0, 1, \ldots, \ell - 1$. Matching the location in L of each $\alpha(i_j)$ with that of $\alpha(i_j - 1)$, and applying Lemma 1 ℓ times we conclude that

$$\vartheta(k_0) = 0, \vartheta(k_1) = 1, \ldots, \vartheta(k_{\ell-1}) = \ell - 1. \tag{3}$$

Having done that, we can continue to the second smallest character occurring in η, and repeat the process, except that this time, the preceding rotations must occur at location ℓ in L. So, if this character is found in locations $k'_0, \ldots, k'_{\ell'-1}$, we have

$$\vartheta(k'_0) = \ell, \vartheta(k'_1) = \ell + 1, \ldots, \vartheta(k'_{\ell-1}) = \ell + \ell' - 1. \tag{4}$$

Algorithm 3.2 applies this process to create ϑ. The algorithm uses characters of the alphabet Σ as array indices, tacitly assuming that $\Sigma = \{0, \ldots, |\Sigma| - 1\}$.

Lines 1 through 8 in the algorithm are mundane; their main purpose is to compute the contents of array before, which, at its c^{th} position contains the number of times a character strictly smaller than c occurs in the in the input.

The heart of the algorithm is in lines 10 through 13. This loop effectively implements the process described above for each of the characters that occur in η. The tricky part is that this is done simultaneously for all characters. Thus, instead of iterating over the different characters in η, and then, examining for each character all its locations, the loop in line 10 scans the positions in η in order. Array seen records at position c, the number of times that c was seen in course of this scan.

Line 12 is the essence of the loop; this line generalizes (3) and (4). The value of before[c] provides the baseline, that is, the locations which are reserved for smaller characters (these locations were matched in previous iterations, or will be matched by subsequent iterations of this loop), while seen[c] is the number of matches of c-locations which were recorded in previous iterations of this loop into ϑ.

Finally, Algorithm 3.3, combines functions Thread (Algorithm 3.1) and Match (Algorithm 3.2) for inverting the Burrows-Wheeler transform.

Algorithm 3.2 $\mathsf{Match}(\eta)$ // *Given a string $\eta \in \Sigma^+$, return the permutation ϑ.*

1: **let** $n \equiv |\eta|$ // *determine the input's length*
2: **let** counts be a zero initialized array of size $|\Sigma|$
3: **For** $i = 0, \ldots, n-1$ **do** // *set* $\mathsf{counts}[c] = |\{i \mid \eta[i] = c\}|$ *for all* $c \in \Sigma$
4: **let** $c \equiv \eta[i]$ // *the character in the input we currently inspect*
5: $\mathsf{counts}[c] \leftarrow \mathsf{counts}[c] + 1$ // *count this occurrence of* c
6: **let** before be a zero initialized array of size $|\Sigma|$
7: **For** $c = 2, \ldots, |\Sigma|$ **do** // *set* $\mathsf{before}[c] = |\{i \mid \eta[i] < c\}|$ *for all* $c \in \Sigma$
8: $\mathsf{before}[c] = \mathsf{before}[c-1] + \mathsf{before}[c]$ // *standard prefix sum*
9: **let** seen be a zero initialized array of size $|\Sigma|$
10: **For** $i = 0, \ldots, n-1$ **do** // *set* $\vartheta(i)$ *to the next available match*
11: **let** $c \equiv \eta[i]$ // *the character in the input we currently inspect*
12: $\vartheta(i) \leftarrow \mathsf{before}[c] + \mathsf{seen}[c]$ // *locations* $0, \ldots, \mathsf{before}[c]-1$ *are reserved for* $c' < c$,
// *while locations* $\mathsf{before}[c], \ldots, \mathsf{before}[c] + \mathsf{seen}[c] - 1$
// *were used for earlier occurrences of* c
13: $\mathsf{seen}[c] \leftarrow \mathsf{seen}[c] + 1$ // *mark this occurrence of* c *as seen*
14: **Return** ϑ

Algorithm 3.3 $\mathsf{BWT}^{-1}(\eta)$ // *For a string $\eta \in \Sigma^+$, return the smallest string β,such that* $\mathsf{BWT}(\beta) =$

1: **let** $\vartheta \equiv \mathsf{Match}(\eta)$
2: **Return** $\mathsf{Thread}(\eta, \vartheta)$

4 The Bijective String Sorting Transform

In this section, we present the $\mathcal{S}$-transform, our bijective- Burrows-Wheeler-like string sorting transform and its inverse $\mathcal{S}^{-1}$. The outline of the algorithm for computing $\mathcal{S}$ is similar to that of the Burrows-Wheeler including computing rotations, sorting these, and then selection of the last character.

The main difference is that the $\mathcal{S}$-transform does not work on the entire input as a whole. Instead, given a string α, the transform decomposes it into words,

$$\alpha = \omega_0\omega_1 \cdots \omega_{m-1} \tag{5}$$

and then proceeds to computing the rotations of each of these words, sorting all of the rotations together, and then selecting the last character of the rotations in their sorted order. The details are supplied Algorithm 4.1.

Algorithm 4.1 $\mathcal{S}(\alpha)$ *// Given a string $\alpha \in \Sigma^+$, return the bijective string sorting transform $\mathcal{S}(\alpha)$.*

1: $W \leftarrow \mathsf{Factor}(\alpha)$ *// compute the Lyndon factorization of α*
2: $R \leftarrow \emptyset$ *// R will be the set of rotations of these fragments*
3: **For all** $\omega \in W$ **do** *// retrieve all rotations of ω*
4: $R \leftarrow R \cup \mathsf{CyclicRotations}(\omega)$ *// and collect these into R*
5: **Return** $\mathsf{Last}(R)$

The algorithm uses as subroutines function CyclicRotations (presented above in Algorithm 2.1) to produce all the cyclic rotations of the fragments, and function Last (presented above in Algorithm 2.2) for sorting these, and selecting their last element.

The factorization (5) is such that each ω_i is a Lyndon [**?**], i.e., a word which is smaller than all of its rotations, $\omega_i < \omega_i(j)$ for $j = 1, 2, \ldots, |\omega_i| - 1$. It is also required that $\omega_0 < \omega_1 < \cdots < \omega_{m-1}$. The presentation of α in the form (5) ssatisfying these two properties is known as the *Lyndon factorization*. It is well known that the Lyndon factorization is unique. Function Factor called in 1 in the Algorithm 4.1 uses Duval's [**?**].algorithm for computing the Lyndon factorization in linear time and space.

Recall that we have left open the issue of extending the order relation in Σ to strings of unequal length in Σ^+. Our transform works with two possible such extensions, the usual lexicographical comparison in which if α is a prefix of β then $\alpha < \beta$. The other extension, which can be viewed as slightly more elegant, is that in comparing two string of unequal length, we compare the *infinite* periodic repetitions of each of these, or, phrased differently, comparing $\alpha^{|\beta|}$ with $\beta^{|\alpha|}$. Consider for example the strings "***the***"' and "***there***".

Then, $\texttt{the} < \texttt{there}$ according to the first definition, while $\texttt{the} > \texttt{there}$ according to the *infinite-periodic* order since $\texttt{the}\cdots > \texttt{therethere}\cdots$.

Interestingly, the Lyndon factorization algorithm works for both variations of the "lexicographical" orders. The point where these differ is in sorting together the the rotations of these Lyndon words. For the first, simple and standard (but somewhat less elegant), definition we can sort the rotations together using the linear time suffix array construction algorithm of Kärkkäinen, Sanders and Burkhardt [**?**].

Recall that Kärkkäinen et. al's algorithm sorts the suffixes recursively, where in each recursive step, the algorithm partitions the input into character triples, where each character triple is considered a new character. A linear time, radix sort, algorithm is applied to the new set of characters, and the recursion continues only if these new characters are not all distinct. In applying this algorithm to sorting the rotations of a single, non-periodic string (as we have in the Lyndon decompositions) ω, it is sufficient to sort the suffixes of $\omega\omega$. Thus, we can sort set of rotations of a single Lyndon factor separately.

Consider now the problem of comparing rotations of of the factors $\omega_0, \omega_1, \ldots, \omega_{m-1}$, where we leave aside the issue of comparing rotations of the same Lyndon factor. We deal with this specific problem using the same paradigm of Kärkkäinen et. al's, except that the grouping together of the triples is a cyclic fashion is done in a cyclic fashion. Of course, if the size of some ω_i is not divisible by three, the recursive step does *not* reduce the number of characters in it. In the j^{th} recursive step, the algorithm thus manipulates "characters" which belong to sequences of length 3^j in the input. To ensure that only linear work is done, we prune at the j^{th} step those characters which belong to factors whose length is no greater than 3^j. This pruning is carried out *even* if these characters are not unique!

Merging the result of the inter-factor and intra-factor sorting steps can be easily done in linear time. Unfortunately, this technique only works in linear time for the standard lexicographical order. Sorting according to the infinite periodic order shall require $O(n \lg n)$ time.

Note. *The Lyndon factorization probably accounts for the better compression results achieved by the $\mathcal{S}$-transform. Recall that the standard application of BWT breaks the input into blocks, and then applies the "block-sorting" procedure to each block. In using $\mathcal{S}$-transform, we break each block into smaller blocks, the Lyndon words, and apply a similar (but not identical) process to each such word. Now, the fact that this refining breakdown into blocks is not arbitrary, but rather depends on underlying properties of the*

input, may very well be the reason for the better performance we witness.

5 Inverting the String-Sorting Bijective Transform

Algorithm 5.1 gives the procedure for inverting the transform, $\mathcal{S}^{-1}$.

Algorithm 5.1 $\mathcal{S}^{-1}(\eta)$ *// For a string $\eta \in \Sigma^+$, return the string α, such that $\mathcal{S}(\alpha) = \eta$.*

1: **let** $\vartheta \equiv \mathsf{Match}(\eta)$
2: **Return** $\mathsf{MultiThread}(\eta, \vartheta)$

Evidently, just like the inversion of the Burrows-Wheeler transform, the inversion $\mathcal{S}^{-1}$ relies on an auxiliary permutation ϑ, which plays a similar role in both inversions. Rather surprisingly, the same function Match (recall Algorithm 3.2 above) can be used for generating the permutation ϑ. The difference is that this time ϑ is not cyclic. Instead, applying Match returns a permutation ϑ which has m cycles, each corresponding to a word ω_i.

More specifically, traversing the cycle $0, \vartheta(0), \vartheta(\vartheta(0)), \ldots, \vartheta^{-1}(0)$ produces the word ω_{m-1}, last to first character. That is to say, if $\eta = \mathcal{S}(\alpha)$, then the last character of w_{m-1} (and hence of α) is $\eta[0]$, the once preceding it is $\eta[\vartheta(0)]$, etc.

Let k now be the smallest integer which is not included in this first cycle. Then, traversing the cycle $k, \vartheta(k), \vartheta(\vartheta(k)), \ldots, \vartheta^{-1}(k)$ produces ω_{m-2}, again, from last character to first. The remaining words are produced by carrying out this process iteratively.

Algorithm 5.1 thus cannot use function Thread (Algorithm 3.1 above) to reconstruct the original string α. Instead, it uses a more general function, MultiThread, for traversing the permutation ϑ. Curiously, function MultiThread is a true generalization of Thread, in the sense that the call to Thread in the inverse Burrows-Wheeler transform, Algorithm 3.3, can be transparently be replaced by a call to MultiThread.

The details of MultiThread are depicted in Algorithm 5.2, which given a string $\eta \in \sigma^+$, $\eta = \mathsf{Acronym}(\alpha)$, and a permutation ϑ, with the properties as described above, traces the cycles in ϑ, to produce the inverse transform α, last character to first, out of η.

Lines 1 through 4 in this algorithm are mundane. They produce a temporary array T, which initially reflects the permutation ϑ. As we traverse

Algorithm 5.2 MultiThread(η, ϑ) *// Given a string $\eta = \mathcal{S}(\alpha)$, and the permutation ϑ, return α.*

```
 1: let n ≡ |η|                                   // determine the input's length
 2: let T be an uninitialized integers array of length n   // used for tracking
    cycles in ϑ
 3: For i = 0, ..., n − 1 do                      // initialize T with the permutation ϑ
 4:    T[i] ← ϑ(i)                                // initialize the i^th position
 5: let α be an uninitialized string of length n  // to be filled with the result,
    last to first
 6: i ← n − 1                                     // i is the next position in α to be filled
 7: For j = 0, ..., n do                          // follow all cycles defined by ϑ
 8:    If T[i] ≠ ⊥ then                           // a new cycle, starting at i, was discovered
 9:       k ← j                                   // k is used for traversing the cycle beginning at i
10:       Repeat                                  // traverse each element in the cycle which begins at i
11:          α[i] ← n[k]                          // produce the next output character
12:          i ← i − 1                            // and step back to the next output character to fill
13:          t ← k                                // t stores the previous value of k
14:          k ← T[k]                             // proceed to the next element in the cycle
15:          T[t] ← ⊥                             // mark previous element as visited
16:       until T[k] = ⊥                          // the current cycle was exhausted
17: Return α
```

the cycles of ϑ, we mark each traversed element by setting the corresponding value of array T to $\perp$.

Next, the algorithm proceeds to producing the returned string α, starting at its last character, working its way to its first. The outer loop (lines 7–16) examines each position in array T in turn. If the value stored in this position is not $\perp$, then the inner loop (lines 10–16) follows up the cycle in ϑ that starts at this position, producing an output character in each iteration, and marking each visited position by setting the corresponding location of T to $\perp$.

6 Correctness of the Inversion Algorithm

It is easy to check that algorithms Algorithm 5.1 and Algorithm 5.2 require linear time and space. We now turn to the issue of their correctness.

Examining Algorithm 4.1, we see that it effectively computes a permutation π of the input. A position i in the input string α is first associated with a certain word $\omega \in W$. Exactly one of the rotations of ω is such that this

position comes to be the last character. The sorting together of all rotations of the words in W assigns an ordinal number to this rotation; this ordinal number is nothing else than $\pi(i)$.

Given $\eta = \mathcal{S}(\alpha)$, finding the inverse of π is done again by computing the auxiliary permutation ϑ, but this time, ϑ is defined in a piecemeal fashion. Let $\pi_\omega : \{0, \ldots, |\omega| - 1\} \to \{0, \ldots, n-1\}$ be the function describing the mapping from the positions of a word $\omega \in W$ into positions of η as carried out by $\mathcal{S}$. The defining property of ϑ is

$$\forall k, (0 \leq k < n) \wedge (k = \pi_\omega(i)) \implies \vartheta(k) = \pi(i - 1 \bmod |\omega|). \tag{6}$$

% That is, having matched a position k in η not only with some word ω but also with a position i in that word, we can match position $i - 1 \bmod |\omega|$, the cyclically preceding position, in ω, with $\vartheta(k)$.

To understand why Algorithm 3.2 computes ϑ also for the $\mathcal{S}$ transform, let us consider the general setting in which we sort together rotations of multiple words. Henceforth, let $W \subset \Sigma^+$ be a fixed finite set of words, and let $n = \sum_{\omega \in W} |\omega|$, be the total length of all the words in W. Also, let $L = L_0 L_1 \cdots L_{n-1}$ be the sorted list of all rotations of the words in W, so each L_i is a rotation of some word in W, and let η be the string defined by $\eta[k] = L_k[-1]$.

The following generalizes Lemma 1.

Lemma 3. *If $L_j = L_k(-1)$ then $j = \vartheta(k)$.*

Proof. The proof is essentially the same as that of the proof of Lemma 1. From the assumptions it follows that there is a word $\omega \in W$ and an index i such that $L_k = \omega(i + 1)$ and $L_j = \omega(i)$. The last character in L_k is therefore $\omega[i]$ while the last character in L_j is $\omega[i - 1]$. Thus, if we knew that $\eta[k]$ is mapped to a certain position in ω, we will be able to conclude that $\eta[j]$ is mapped to the cyclically previous position in ω. □

Lemma 4. *For an arbitrary $c \in \Sigma$, let $L_{i_0}, \ldots, L_{i_{\ell-1}}$ be the list of all rotations L in which c occurs as the last character, that is $\eta[i_0] = \eta[i_1] = \cdots = \eta[i_{\ell-1}] = c$. Then, the list $L_{i_0}(-1), \ldots, L_{i_{\ell-1}}(-1)$ occurs consecutively and in that order in L.*

Proof. Since the first character of each of the rotations $L_{i_0}(-1), \ldots, L_{i_{\ell-1}}(-1)$ is c, we can rewrite these as $cL_{i_0}, \ldots, cL_{i_{\ell-1}}$. This list is sorted since we assumed that $\alpha()i_0, \ldots, \tau^{i_{\ell-1}}$ are sorted. Further, the elements of this list occur consecutively in L since they all begin with c and no other rotation begins with c.. □

7 Future Work

Clearly, the work ahead of us is in evaluating the efficacy of the transform described here in state of the art compression programs such as *bzip2* and *7-zip*.

We are intrigued by the question of sorting the rotations of the Lyndon decomposition in linear time with the *infinite periodic* order.