Theoretical Computer Science 531 (2014) 77–89

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

A bijective variant of the Burrows–Wheeler Transform using V-order

Jacqueline W. Daykin[a,b,*], W.F. Smyth[c,d]

[a] *Department of Computer Science, Royal Holloway, University of London, UK*
[b] *Department of Informatics, King's College London, UK*
[c] *Algorithms Research Group, Department of Computing & Software, McMaster University, Hamilton ON L8S 4K1, Canada*
[d] *School of Mathematics & Statistics, University of Western Australia, Perth, Australia*

A R T I C L E   I N F O

Article history:
Received 22 September 2012
Received in revised form 19 January 2014
Accepted 4 March 2014
Communicated by A. Apostolico

Keywords:
Algorithm
Bijective
Burrows–Wheeler Transform
Complexity
Lexicographic order
Lex-extension order
Linear
Lyndon word
String
Suffix array
Total order
V-BWT
V-order
V-transform
V-word
Word

A B S T R A C T

In this paper we introduce the *V-transform* (V-BWT), a variant of the classic Burrows–Wheeler Transform. The original BWT uses lexicographic order, whereas we apply a distinct total ordering of strings called V-order. V-order string comparison and Lyndon-like factorization of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ into V-words have recently been shown to be linear in their use of time and space (Daykin et al., 2011) [18]. Here we apply these subcomputations, along with $\Theta(n)$ suffix-sorting (Ko and Aluru, 2003) [26], to implement linear V-sorting of all the rotations of a string. When it is known that the input string $\boldsymbol{x}$ is a V-word, we compute the V-transform in $\Theta(n)$ time and space, and also outline an efficient algorithm for inverting the V-transform and recovering $\boldsymbol{x}$. We further outline a bijective algorithm in the case that $\boldsymbol{x}$ is arbitrary. We propose future research into other variants of transforms using lex-extension orderings (Daykin et al., 2013) [19]. Motivation for this work arises in possible applications to data compression.

1. Introduction

We present here a variant of the Burrows–Wheeler Transform [4,9,33,22] based on a new underlying order. The classic transform applies ***lexorder*** (lexicographic order); we introduce the V-***transform*** (V-*BWT*) which applies V-order [11,14,18,19]. The Burrows–Wheeler Transform (BWT) admits linear-time algorithms for both the transform and inverse [2], and space saving techniques have been achieved by factoring the input string into Lyndon words [8,13,21,28,32,34]. Similarly, we factor the input string into V-words, which are analogues of Lyndon words that use V-order – for V-words, we compute the V-transform in linear time and propose an efficient priority queue approach to computing the inverse. We note that both Lyndon words and V-words have arisen naturally in rhythms in traditional African music [6,7].

[*] Corresponding author.
E-mail addresses: jackie.daykin@rhul.ac.uk, jackie.daykin@kcl.ac.uk (J.W. Daykin), smyth@mcmaster.ca (W.F. Smyth).

http://dx.doi.org/10.1016/j.tcs.2014.03.014
0304-3975/© 2014 Elsevier B.V. All rights reserved.

V-words are defined using a simple extension of lexorder, *lex-extension*, which in this case is a combination of lexorder and V-order, as explained in Section 2. This connection with lexorder allows us to modify and apply linear-time methods for sorting the suffixes of a V-word. The sorted suffixes can then be used to V-order the rotations of a given V-word from which we derive its V-transform. In this paper we describe modifications to the suffix-sorting algorithm of Ko and Aluru [26], but other linear-time suffix-sorting methods such as [24,31] could equally have been used.

The BWT has been heavily researched in the last decade leading to efficient time and space implementations, notably [25,23,10], along with a range of bijective variants such as the sort transform [27,22]; wide ranging applications of the BWT include bioinformatics and multimedia information retrieval [2].

The BWT often clusters occurrences of the same letter in the input string into runs. Hence applications arise in data compression, for instance as preprocessing for the run-length encoding method which is suitable for faxed documents and simple graphic images. To our knowledge this paper is the first to compute the BWT using nonlexicographic ordering of the elements, a demonstration that the concept is viable – see also Question 6.4. Indeed, for some strings the use of V-order yields better clustering, thus better compression. Our ultimate objective is to identify total orders that achieve improved clustering over a wide range of input strings – we provide here one new strategy worthy of both theoretical and experimental further investigation; see motivational Question 6.1.

1.1. Definitions

We use standard terminology from automata theory and stringology, so we just give a few basic stringological definitions (see [34] for further stringology theory and algorithms):

- If for some string $\boldsymbol{x}$ we can write $\boldsymbol{x} = \boldsymbol{uv} = \boldsymbol{wu}$ for some nonempty $\boldsymbol{u}$, then we say that $\boldsymbol{x}$ has ***border u***; if no such $\boldsymbol{u}$ exists, then $\boldsymbol{x}$ is said to be ***border-free***.
- If we can write $\boldsymbol{x} = \boldsymbol{u}^k$ for some nonempty $\boldsymbol{u}$ and some integer $k > 1$, we say that $\boldsymbol{x}$ is a ***repetition***; otherwise, we say that $\boldsymbol{x}$ is ***primitive***.
- If a string $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is said to be a ***rotation*** (cyclic shift) of $\boldsymbol{x}$, specifically the $|\boldsymbol{u}|^{\text{th}}$ rotation $R_{|\boldsymbol{u}|}(\boldsymbol{x})$, where $|\boldsymbol{u}| \in \{0, \dots, |\boldsymbol{x}|\}$. Note that $R_0(\boldsymbol{x}) = R_{|\boldsymbol{x}|}(\boldsymbol{x})$.

If a string $\boldsymbol{x}$ is less than a string $\boldsymbol{y}$ in lexorder, we write $\boldsymbol{x} < \boldsymbol{y}$ and $\boldsymbol{y} > \boldsymbol{x}$. Note that the terms ***string*** and ***word*** mean the same thing, hence we use both here. Also, for the examples included over the non-negative integers, the natural order will be assumed, that is $\{0 < 1 < 2 < \cdots\}$. All strings are written in mathbold: $\boldsymbol{x}$, $\boldsymbol{w}$, and so on; $\boldsymbol{\varepsilon}$ denotes the empty string.

The remainder of this paper is organized as follows. Section 2 gives necessary preliminaries about V-order that will be required in order to understand our algorithms. Section 3 describes the V-order computation of the BWT (and reverse BWT) when the input string is a V-word; this is generalized in Section 4 to an arbitrary input. Section 5 provides a separate description of the method used for V-sorting all the rotations of a given string – this is the main component of the BWT calculation. In Section 6 we suggest future research directions.

2. *V*-order and *V*-words

In this section we define V-order and associated strings called V-words, along with stating related results which will be applied to computing the transform and its inverse. (See [11,12,14] for further details on V-order and V-words; see [18,19] for a linear sequential algorithm for computing the unique V-word factorization of a string – an optimal parallel PRAM variant is given in [16].)

Let Σ be a totally ordered alphabet, and let $\boldsymbol{u} = u_1u_2 \dots u_n$ be a string over Σ. Define $h \in \{1, \dots, n\}$ by $h = 1$ if $u_1 \leqslant u_2 \leqslant \cdots \leqslant u_n$; otherwise, by the unique value h such that $u_{h-1} > u_h \leqslant u_{h+1} \leqslant u_{h+2} \leqslant \cdots \leqslant u_n$; thus $h = n$ if $u_{n-1} > u_n$, since u_{n+1} does not exist. Let $\boldsymbol{u}^* = u_1u_2 \dots u_{h-1}u_{h+1} \dots u_n$, where the star * indicates deletion of the letter u_h. Write $\boldsymbol{u}^{s*}$ for $(\dots(\boldsymbol{u}^*)^* \dots)^*$ with $s \geqslant 0$ stars.[1] Let $g = \max\{u_1, u_2, \dots, u_n\}$, and let k be the number of occurrences of g in $\boldsymbol{u}$. Then the sequence $\boldsymbol{u}, \boldsymbol{u}^*, \boldsymbol{u}^{2*}, \dots$ ends $g^k, \dots, g^2, g^1, g^0 = \boldsymbol{\varepsilon}$. In the *star tree* each string $\boldsymbol{u}$ over Σ labels a vertex, and there is a directed edge from $\boldsymbol{u}$ to $\boldsymbol{u}^*$, with $\boldsymbol{\varepsilon}$ as the root. Note that, in case of equality ($u_h = u_{h+1}$), it is immaterial whether u_h or u_{h+1} is deleted: all equal adjacent letters will be deleted in sequence.

Definition 2.1. We define V-order $\prec$ between distinct strings $\boldsymbol{u}, \boldsymbol{v}$. First $\boldsymbol{v} \prec \boldsymbol{u}$ if $\boldsymbol{v}$ is in the path $\boldsymbol{u}, \boldsymbol{u}^*, \boldsymbol{u}^{2*}, \dots, \boldsymbol{\varepsilon}$. If $\boldsymbol{u}, \boldsymbol{v}$ are not in a path, there exist smallest s, t such that $\boldsymbol{u}^{(s+1)*} = \boldsymbol{v}^{(t+1)*}$. Put $\boldsymbol{c} = \boldsymbol{u}^{s*}$ and $\boldsymbol{d} = \boldsymbol{v}^{t*}$; then $\boldsymbol{c} \neq \boldsymbol{d}$ but $|\boldsymbol{c}| = |\boldsymbol{d}| = m$ say. Let j be the greatest i in $1 \leqslant i \leqslant m$ such that $\boldsymbol{c}[i] \neq \boldsymbol{d}[i]$. If $\boldsymbol{c}[j] < \boldsymbol{d}[j]$ in Σ then $\boldsymbol{u} \prec \boldsymbol{v}$. Clearly $\prec$ is a total order.

The V-ordering of two strings can be computed in linear time using a linked list [18,19]. For the V-BWT, we will be ordering the rotations of a string in V-order $\prec$, illustrated by:

[1] Note that this star operator, as defined in [11,14] etc., is distinct from the Kleene star operator: Kleene star is applied to sets, while this V-star is applied to strings.

Example 2.2. Consider the rotations of $\boldsymbol{x} = 53531$. We have $(53531)^{2*} = (53153)^{2*} = 535$, namely their common ancestor in the star tree. Then $(53531)^* = 5353$ and $(53153)^* = 5315$, with a rightmost mismatch in position 4 ($3 < 5$), from which we conclude that $53531 \prec 53153$. For 53153 and 15353 the common ancestor is 55; adding back the last deleted letter yields 535 and 155, respectively, with a rightmost mismatch in position 2 ($3 < 5$), so that $53153 \prec 15353$. Similarly $15353 \prec 35315 \prec 31535$, completing all the rotations.

In contrast to lexorder, if $\boldsymbol{u}$ is any proper subsequence of $\boldsymbol{x}$, then $\boldsymbol{u} \prec \boldsymbol{x}$:

Lemma 2.3. *(See [18].) Given a string $\boldsymbol{x}$ of length n, let $\boldsymbol{u} = \boldsymbol{x}[i_1]\boldsymbol{x}[i_2]\ldots\boldsymbol{x}[i_r]$ for $1 \leqslant i_1 < i_2 < \cdots < i_r$, $0 \leqslant r < n$. Then $\boldsymbol{u} \prec \boldsymbol{x}$.*

For the calculation of the inverse V-BWT, the following will be useful:

Remark 2.4. Given strings $\boldsymbol{u}$ and $\boldsymbol{v} \neq \boldsymbol{u}$, suppose that in the deletion sequence of $\boldsymbol{u}$ and $\boldsymbol{v}$, s, t are the least integers such that $\boldsymbol{u}^{(s+1)*} = \boldsymbol{v}^{(t+1)*}$. If $\boldsymbol{u} \prec \boldsymbol{v}$ (respectively, $\boldsymbol{u} \succ \boldsymbol{v}$), then for every $s_1 \in 0..s$, $t_1 \in 0..t$, $\boldsymbol{u}^{s_1*} \prec \boldsymbol{v}^{t_1*}$ (respectively, $\boldsymbol{u}^{s_1*} \succ \boldsymbol{v}^{t_1*}$).

Using this remark, we can now establish an important new property of V-order that, as we shall see, facilitates the ordering of the rotations of a given string, often without the need to access the text.

Lemma 2.5. *Given strings $\boldsymbol{u}$ and $\boldsymbol{v} \neq \boldsymbol{u}$, where $|\boldsymbol{u}| = |\boldsymbol{v}|$,*

$$\boldsymbol{u} \prec \boldsymbol{v} \quad \Longleftrightarrow \quad \lambda\boldsymbol{u} \prec \lambda\boldsymbol{v},$$

where λ is any letter.

Proof. The proof is by induction on $|\boldsymbol{u}|$. For $|\boldsymbol{u}| = 1$ and $\boldsymbol{u} \prec \boldsymbol{v}$ (thus $\boldsymbol{u} < \boldsymbol{v}$), there are five initial cases (the underscore denotes the first letter deleted):

(a) $\lambda < \boldsymbol{u}$ (for example, $\lambda = 1 : 1\underline{2} \prec 1\underline{4}$);
(b) $\lambda = \boldsymbol{u}$ (for example, $\lambda = 2 : 2\underline{2} \prec 2\underline{4}$);
(c) $\boldsymbol{u} < \lambda < \boldsymbol{v}$ (for example, $\lambda = 3 : 3\underline{2} \prec \underline{3}4$);
(d) $\lambda = \boldsymbol{v}$ (for example, $\lambda = 4 : 4\underline{2} \prec \underline{4}4$);
(e) $\lambda > \boldsymbol{v}$ (for example, $\lambda = 5 : 5\underline{2} \prec 5\underline{4}$).

In each case, $\boldsymbol{u} \prec \boldsymbol{v}$ ($2 < 4$) ensures that $\lambda\boldsymbol{u} \prec \lambda\boldsymbol{v}$ and *vice versa.*

Suppose then that the lemma holds for every $|\boldsymbol{u}| = |\boldsymbol{v}| = k \geqslant 1$; that is, for $\boldsymbol{u} \neq \boldsymbol{v}$, $\boldsymbol{u} \prec \boldsymbol{v}$ if and only if $\lambda\boldsymbol{u} \prec \lambda\boldsymbol{v}$. Consider therefore strings $\boldsymbol{u}$ and $\boldsymbol{v} \neq \boldsymbol{u}$, where $|\boldsymbol{u}| = |\boldsymbol{v}| = k + 1$. Two cases arise: Case 1, where $\boldsymbol{u}^* = \boldsymbol{v}^*$ ($s = t = 0$ in the deletion sequence of $\boldsymbol{u}$ and $\boldsymbol{v}$) or else Case 2, where for $\boldsymbol{u} \prec \boldsymbol{v}$,

$$\boldsymbol{u}^* \prec \boldsymbol{v}^* \quad \text{(by Remark 2.4)} \tag{1}$$

so that, for arbitrary λ,

$$\lambda\boldsymbol{u}^* \prec \lambda\boldsymbol{v}^* \quad \text{(by the inductive assumption)}, \tag{2}$$

with $\prec$ replaced by $\succ$ for $\boldsymbol{u} \succ \boldsymbol{v}$. We consider these cases in terms of the four possibilities that arise in the deletion sequence:

(P1) $(\lambda\boldsymbol{u})^* = \lambda\boldsymbol{u}^*$, $(\lambda\boldsymbol{v})^* = \lambda\boldsymbol{v}^*$.
Here the same letter is deleted from $\lambda\boldsymbol{u}$ as from $\boldsymbol{u}$, from $\lambda\boldsymbol{v}$ as from $\boldsymbol{v}$. Thus in Case 1, adding back the deleted letters yields the same $\prec$ relationship for $\boldsymbol{u}, \boldsymbol{v}$ as for $\lambda\boldsymbol{u}, \lambda\boldsymbol{v}$, as required. In Case 2, since $|\boldsymbol{u}^*| = |\boldsymbol{v}^*| = k$, the result follows immediately from (1) and (2).

(P2) $(\lambda\boldsymbol{u})^* = \boldsymbol{u}$, $(\lambda\boldsymbol{v})^* = \boldsymbol{v}$.
Now $\boldsymbol{u}$ and $\boldsymbol{v}$ are on the same deletion path as $\lambda\boldsymbol{u}$ and $\lambda\boldsymbol{v}$, respectively (that is, $(\lambda\boldsymbol{u})^{2*} = \boldsymbol{u}^*$ and $(\lambda\boldsymbol{v})^{2*} = \boldsymbol{v}^*$), so that by Remark 2.4 the result for both pairs must be the same.

(P3) $(\lambda\boldsymbol{u})^* = \lambda\boldsymbol{u}^*$, $(\lambda\boldsymbol{v})^* = \boldsymbol{v}$.
Each of $\lambda\boldsymbol{v}$, $\boldsymbol{v}$ and $\boldsymbol{v}^{r*}$, for every $r \in 1..|\boldsymbol{v}|$, must be a monotone nondecreasing (MND) sequence of letters, with the leftmost letter therefore always deleted; on the other hand, $\lambda\boldsymbol{u}$ is not MND because its leftmost position is not deleted. Let $s \in 0..|\boldsymbol{u}| - 1$ be the least integer such that $\boldsymbol{u}^{s*}$ is MND. Then

$$(\lambda\boldsymbol{u})^{s*} = \lambda\boldsymbol{u}^{s*}, \qquad (\lambda\boldsymbol{v})^{s*} = \boldsymbol{v}^{(s-1)*}, \qquad (\lambda\boldsymbol{v})^{(s+1)*} = \boldsymbol{v}^{s*}.$$

Note that, even though $\boldsymbol{u}^{s*}$ is MND, it may still be true that $\lambda > \boldsymbol{u}^{s*}[1]$; thus we know only that $(\lambda\boldsymbol{u})^{(s+1)*}$ equals one of $\boldsymbol{u}^{s*}$ or $\lambda\boldsymbol{u}^{(s+1)*}$. Accordingly, let $t \in s..|\boldsymbol{u}|$ be the least integer such that $\boldsymbol{u}^{t*} = \boldsymbol{v}^{t*}$, and let t' be the least integer such that $(\lambda\boldsymbol{u})^{t'*} = (\lambda\boldsymbol{v})^{t'*}$. Then either $t' = t$ or $t' = t+1$. Next let $s' \in s..|\boldsymbol{u}|$ be the least integer such that $(\lambda\boldsymbol{u})^{(s'+1)*} = \boldsymbol{u}^{s'*}$. Since certainly $(\lambda\boldsymbol{u})^{t*} = \boldsymbol{u}^{t*}$, we see that $t \geqslant s'$. We therefore consider three possibilities, which it may be helpful to relate to three examples:

$$t > s': \quad \boldsymbol{u} = 322, \quad \boldsymbol{v} = 222, \quad \lambda = 1;$$

$$t' = t = s': \quad \boldsymbol{u} = 311, \quad \boldsymbol{v} = 223, \quad \lambda = 2;$$

$$t' = t+1 = s'+1: \quad \boldsymbol{u} = 311, \quad \boldsymbol{v} = 233, \quad \lambda = 2.$$

If $t > s'$, then the addback letters for $\lambda\boldsymbol{u}, \lambda\boldsymbol{v}$ cannot include λ; thus these letters are identical to the addback letters of $\boldsymbol{u}, \boldsymbol{v}$, and so the $\prec$ relation between $\boldsymbol{u}, \boldsymbol{v}$ must be the same as that between $\lambda\boldsymbol{u}, \lambda\boldsymbol{v}$, as required. Suppose then that $t = s'$. If $t' = t$, then t is the least integer such that $\boldsymbol{u}^{t*} = \boldsymbol{v}^{t*}$ and independently such that $(\lambda\boldsymbol{u})^{t*} = (\lambda\boldsymbol{v})^{t*}$. It follows that the addback letter for both $\boldsymbol{u}^{t*}$ and $(\lambda\boldsymbol{u})^{t*}$ must be the same: $\boldsymbol{u}^{t*}[j]$ for some $j \geqslant 1$. On the other hand, the addback letters for $\boldsymbol{v}^{t*}$ and $(\lambda\boldsymbol{v})^{t*}$ will be the adjacent entries $\boldsymbol{v}[t]$ and $\boldsymbol{v}[t-1]$, respectively, with

$$\boldsymbol{u}^{t*}[j] < \lambda \leqslant \boldsymbol{v}[t-1] \leqslant \boldsymbol{v}[t],$$

so that $\boldsymbol{u} \prec \boldsymbol{v}$ and $\lambda\boldsymbol{u} \prec \lambda\boldsymbol{v}$, as required. Finally, suppose that $t' = t+1 = s'+1$. In this case the addback letters for $\boldsymbol{u}, \boldsymbol{v}$ will be $\boldsymbol{u}[j] < \lambda$ and $\boldsymbol{v}[t'-1] \geqslant \lambda$, while for $\lambda\boldsymbol{u}$ and $\lambda\boldsymbol{v}$, they will be λ and $\boldsymbol{v}[t] \geqslant \boldsymbol{v}[t'-1]$. Thus here also $\boldsymbol{u} \prec \boldsymbol{v}$ and $\lambda\boldsymbol{u} \prec \lambda\boldsymbol{v}$.

(P4) $(\lambda\boldsymbol{u})^* = \boldsymbol{u},\ (\lambda\boldsymbol{v})^* = \lambda\boldsymbol{v}^*$.
This case is complementary to (P3) with complementary results. □

This fundamental property of V-order allows us to assume that for distinct strings $\boldsymbol{u}, \boldsymbol{v}$ of equal length, $\boldsymbol{wu} \prec \boldsymbol{wv}$ if and only if $\boldsymbol{u} \prec \boldsymbol{v}$, where $\boldsymbol{w}$ is any string. The condition $|\boldsymbol{u}| = |\boldsymbol{v}|$ is in a sense superfluous, since for $|\boldsymbol{v}'| > |\boldsymbol{u}|$, the lemma applies after the deletion sequence reduces $\boldsymbol{v}'$ to $\boldsymbol{v}$.

The ***V-form*** of a string $\boldsymbol{x}$ is defined in [11,18] as

$$V_k(\boldsymbol{x}) = \boldsymbol{x} = \boldsymbol{x}_0 g \boldsymbol{x}_1 g \ldots \boldsymbol{x}_{k-1} g \boldsymbol{x}_k, \tag{3}$$

for possibly empty $\boldsymbol{x}_i, i = 0, 1, \ldots, k$. (Thus we suppose that the largest letter g in $\boldsymbol{x}$ occurs exactly k times.)

The following lemmas are applied in Section 3 to form and invert the V-transform.

Lemma 2.6. *(See [14].) Suppose distinct strings $\boldsymbol{u}$ and $\boldsymbol{v}$ have V-forms*

$$\boldsymbol{u} = \boldsymbol{u}_0 g_{\boldsymbol{u}} \boldsymbol{u}_1 g_{\boldsymbol{u}} \ldots \boldsymbol{u}_{j-1} g_{\boldsymbol{u}} \boldsymbol{u}_j, \qquad \boldsymbol{v} = \boldsymbol{v}_0 g_{\boldsymbol{v}} \boldsymbol{v}_1 g_{\boldsymbol{v}} \ldots \boldsymbol{v}_{k-1} g_{\boldsymbol{v}} \boldsymbol{v}_k,$$

where $g_{\boldsymbol{u}}$ and $g_{\boldsymbol{v}}$ are the largest letters in $\boldsymbol{u}$ and $\boldsymbol{v}$, respectively. Then $\boldsymbol{u} \prec \boldsymbol{v}$ if and only if one of the following conditions holds:

(1) *$g_{\boldsymbol{u}} < g_{\boldsymbol{v}}$;*
(2) *$g_{\boldsymbol{u}} = g_{\boldsymbol{v}}$ and $j < k$;*
(3) *$g_{\boldsymbol{u}} = g_{\boldsymbol{v}}$ and $j = k$ and $\boldsymbol{u}_h \prec \boldsymbol{v}_h$, where $h \in 0..j$ is the least integer such that $\boldsymbol{u}_h \neq \boldsymbol{v}_h$.*

Similar to variants of the original Burrows–Wheeler Transform which utilize Lyndon words [27,22], we will assume that the input is factored into V-words, a natural analogue of Lyndon words (see Example 2.2):

Definition 2.7. (See [14].) A string $\boldsymbol{x}$ over Σ is a V-***word*** if it is the unique minimum in V-order $\prec$ among the set of rotations of $\boldsymbol{x}$.

Note that, like Lyndon words, V-words are both primitive and border-free [15], properties which we will use for inverting the V-transform. Despite these similarities, Lyndon words and V-words are generally distinct [19] except for trivial cases (individual letters, for example).

Example 2.8. (1) The V-ordering of all rotations of the string 12345 gives $51234 \prec 45123 \prec 34512 \prec 23451 \prec 12345$. Hence 12345 is not a V-word (though it is a Lyndon word), while 51234 is a V-word but not Lyndon.

(2) 321312 and 4440414243 are both V-words, while 123213 and 0414243444 are Lyndon.

In order to describe the structure of V-words, we introduce the ***lexicographic extension (lex-extension)*** $\prec_{LEX}$ of $\prec$ order [14] as follows:

Definition 2.9. Suppose that according to some factorization F, two strings $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^+$ are expressed in terms of nonempty factors:

$$\boldsymbol{u} = \boldsymbol{u}_1\boldsymbol{u}_2 \ldots \boldsymbol{u}_m, \qquad \boldsymbol{v} = \boldsymbol{v}_1\boldsymbol{v}_2 \ldots \boldsymbol{v}_n.$$

Then $\boldsymbol{u} \prec_{LEX(F)} \boldsymbol{v}$ if and only if one of the following holds:

(1) $\boldsymbol{u}$ is a proper prefix of $\boldsymbol{v}$ (that is, $\boldsymbol{u}_i = \boldsymbol{v}_i$ for $1 \leqslant i \leqslant m < n$); or
(2) for some $i \in 1..\min(m,n)$, $\boldsymbol{u}_j = \boldsymbol{v}_j$ for $j = 1, 2, \ldots, i-1$, and $\boldsymbol{u}_i \prec \boldsymbol{v}_i$.

In other words, $\boldsymbol{u}_1\boldsymbol{u}_2 \ldots \boldsymbol{u}_m \prec \boldsymbol{v}_1\boldsymbol{v}_2 \ldots \boldsymbol{v}_n$ in the lexicographic order of strings, using not $<$ but $\prec$. We will generally write $\prec_{LEX}$ rather than $\prec_{LEX(F)}$ when the factorization F is clear from the context. For further discussion of lex-extension see [19].

Clearly $\boldsymbol{x}_i \preccurlyeq \boldsymbol{x}_j$ implies $g\boldsymbol{x}_i \preccurlyeq g\boldsymbol{x}_j$, which leads to the following insight, where we just use the g's to distinguish the substrings $\boldsymbol{x}_i$ and apply the Lyndon property.

Theorem 2.10. *(See [14].) Let $\boldsymbol{x} \in \Sigma^+$. Then $\boldsymbol{x}$ is a V-word if and only if its V-form (3) has $\boldsymbol{x}_0 = \varepsilon$ with $\boldsymbol{x}_1\boldsymbol{x}_2 \ldots \boldsymbol{x}_k$ a Lyndon word under lex-extension.*

From now on we will denote the set of V-words by $\mathcal{V}$ (like Lyndon words, $\mathcal{V}$ is an instance of a *circ-UMFF*, see [14,15]).

Notice that the V-form (3) of a V-word $\boldsymbol{x}$ must, by the properties of V-order, begin with the largest letter g in $\boldsymbol{x}$. Thus in (3) $\boldsymbol{x}_0 = \boldsymbol{\varepsilon}$, and a V-word must take the form

$$V_k(\boldsymbol{x}) = \boldsymbol{x} = g\boldsymbol{x}_1 \ldots g\boldsymbol{x}_{k-1}g\boldsymbol{x}_k. \tag{4}$$

For the sorting of the suffixes of a string into V-order, it is important that V-words are of type ***Flight Deck*** [17]; that is, that for every V-word $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $\boldsymbol{x}[1] \leqslant_{\mathcal{O}} \boldsymbol{x}[i]$, for every $1 \leqslant i \leqslant n$, where $\mathcal{O}$ is some global ordering. For example, the set of Lyndon words is type Flight Deck under lexorder, while the V-words are also Flight Deck according to the ordering $\mathcal{V}$ – that is, using V-order and lex-extension but with an inverted alphabet (see [19], Lemma 3.16). The Flight Deck condition includes trivial cases known as *V-letters* [19], where the first letter in a string is strictly greater (over Σ) than the subsequent letters; for example, the V-word 51234.

When sorting sets of rotations into their V-order in Section 5, we will use the natural cyclic order of V-letters (see Example 2.8):

Lemma 2.11. *(See [19].) Let $\boldsymbol{x}$ be a V-letter of length n. If $\boldsymbol{r}_i = R_i(\boldsymbol{x})$, $i = 1, 2, \ldots, n$, then $\boldsymbol{r}_1 \succ \boldsymbol{r}_2 \succ \cdots \succ \boldsymbol{r}_n = \boldsymbol{x}$.*

The following lemma, slightly more precise than in its original form, and also relevant to sorting rotations, shows that similar to a Lyndon word, a V-word precedes any of its proper suffixes.

Lemma 2.12. *(See [19].) Let $\boldsymbol{x} \in \mathcal{V}$, where $\boldsymbol{x} = g\boldsymbol{x}_1 g\boldsymbol{x}_2 \ldots g\boldsymbol{x}_k$ is in V-form (4). If $\boldsymbol{p} = g\boldsymbol{x}_1 g\boldsymbol{x}_2 \ldots g\boldsymbol{x}_j \in \mathcal{V}$, where $1 \leqslant j < k$, and if $\boldsymbol{s}$ is any proper suffix of $\boldsymbol{x}$, then $\boldsymbol{px}, \boldsymbol{pxs}, \boldsymbol{xs} \in \mathcal{V}$.*

3. BWT using V-order (V-word input)

In this section we show how to compute the BWT (and its inverse) using V-order on a given string known to be a V-word; the next section deals with the corresponding construction on an arbitrary string.

3.1. The V-BWT

The method is analogous to the standard Burrows–Wheeler Transform, but exchanges lexorder for V-order. Also, similarly to variants of the original transform which assume the input to be a Lyndon word [27], we assume that the input string has been factored into V-words: this saves $O(\log n)$ space for a pointer to the least rotation of a V-word. The method for computing the bijective V-transform of a given V-word $\boldsymbol{x}$ is then:

1. Form the unordered set (as an $n \times n$ matrix) R of all cyclic rotations of an input string $\boldsymbol{x}$ of length n.
2. Sort all the rows (all the suffixes) of R into increasing V-order, forming $R^{\prec}$.
3. The last column of $R^{\prec}$ is the V-transform, denoted $\mathcal{T}$.

Example 3.1. Let the V-word $\boldsymbol{x} = 5312543$. Then, using Lemma 2.6, the sorted set of rotations $R^{\prec}$ is:

5	3	1	2	5	4	3
5	4	3	5	3	1	2
2	5	4	3	5	3	1
1	2	5	4	3	5	3
3	5	3	1	2	5	4
3	1	2	5	4	3	5
4	3	5	3	1	2	5

The V-transform, $\mathcal{T}$, is the last column: 3213455. Note that the 5's have been clustered into a run, but not the 3's. However, the classic lexorder BWT of $\boldsymbol{x}$ yields the transform 3154532, with no clustering of letters at all.

We defer till Section 5 a description of the details of the modifications to a standard suffix-sorting algorithm [26] required to convert it from lexorder to V-order.

3.2. The inverse V-BWT

We show here how to recover the input data from the V-transform $\mathcal{T}$, where the input (first row of $R^{\prec}$) is assumed to be a V-word $\boldsymbol{x}$, hence the minimum rotation (Definition 2.7). Starting with $\mathcal{T}$, the last column in $R^{\prec}$, we deduce the first column $\mathcal{C}$, then use information about the path traced in this process to compute $\boldsymbol{x}$ in reverse order. Let $\mathcal{T} = t_1t_2\dots t_n$, and suppose that g is the greatest letter in $\mathcal{T}$. The column $\mathcal{C}$ consists of the single-letter prefixes of the strictly increasing (in V-order) rotations of $\boldsymbol{x}$ that form the rows of $R^{\prec}$. By the structure (4) of V-words, the input string $\boldsymbol{x}$ starts with $\boldsymbol{x}[1] = g$. Letting k denote the frequency of g in $\mathcal{T}$, we know moreover from Lemma 2.6 that the first k rows in $R^{\prec}$ must start with g (that is, in the V-form (3), $\boldsymbol{x}_0 = \varepsilon$). Hence these k rows have the form $g \dots t_i$ for $1 \leqslant i \leqslant k$, which starts our boot-strapping procedure for computing the inverse.

From the first k rows we know that there are other rotations of $\boldsymbol{x}$ in $R^{\prec}$ starting with t_ig which we then sort into their increasing V-order, and repeat the process for longer prefixes. The invariant is that, during the reconstruction of $R^{\prec}$, the current list of prefixes in $\mathcal{C}$ is correctly sorted in V-order. The method for prefix sorting applies Lemmas 2.3, 2.5 and 2.6; note that the first k rows of $R^{\prec}$ are in the V-form (4), while all subsequent rows are in V-form (3) with $\boldsymbol{x}_0 \neq \varepsilon$.

In order to determine the next entry in $R^{\prec}$, we maintain a priority queue (say a heap $\mathcal{H}$) of the subsequent rotations of each row already entered. Thus, for the first k rows, the rotations beginning $t_1g, t_2g, \dots, t_kg$ need to be entered in some form into $\mathcal{H}$. (However, to avoid duplicates, we enter into $\mathcal{H}$ only those items i for which $t_i < g$; this means that the maximum number of entries in $\mathcal{H}$ will be, not k, but k', the number of *sequences* of largest letters g.) To assist in determining the minimum of each pair of rotations, each entry in $\mathcal{H}$ contains the following fields:

r: the *rank* of the preceding item in $R^{\prec}$ (for example, i whenever $i \in 1..k$);
ℓ: the number of letters so far rotated into the first position (for $i \in 1..k$, $\ell = 1$);
λ: the letter currently rotated from $\mathcal{T}$ (and prefix of the current rotation);
$\hat{\lambda}$: the maximum over all letters λ so far rotated into the first position;
$\#$: the number of occurrences of $\hat{\lambda}$;
j: the leftmost position at which $\hat{\lambda}$ occurs ($1 \leqslant j \leqslant \ell$).

Using Lemma 2.5, the variables (r, ℓ, λ) may determine the order of $\boldsymbol{u}$ and $\boldsymbol{v}$ as follows:

(R1) whenever $\ell_{\boldsymbol{u}} = \ell_{\boldsymbol{v}}$ **and** $\lambda_{\boldsymbol{u}} = \lambda_{\boldsymbol{v}}$, then $\boldsymbol{u} \prec \boldsymbol{v} \Longleftrightarrow r_{\boldsymbol{u}} < r_{\boldsymbol{v}}$.

Similarly, the variables $(\hat{\lambda}, \#, j)$ enable the order of two entries $\boldsymbol{u}$ and $\boldsymbol{v}$ to be computed in constant time in many cases, making use of the following rules (derived from Lemma 2.6):

(R2) if $\hat{\lambda}_{\boldsymbol{u}} < \hat{\lambda}_{\boldsymbol{v}}$, choose $\boldsymbol{u}$;
(R3) if $\hat{\lambda}_{\boldsymbol{u}} = \hat{\lambda}_{\boldsymbol{v}}$ **and** $\#_{\boldsymbol{u}} < \#_{\boldsymbol{v}}$, choose $\boldsymbol{u}$;
(R4) if $\hat{\lambda}_{\boldsymbol{u}} = \hat{\lambda}_{\boldsymbol{v}}$ **and** $\#_{\boldsymbol{u}} = \#_{\boldsymbol{v}}$ **and** $j_{\boldsymbol{u}} < j_{\boldsymbol{v}}$, choose $\boldsymbol{u}$.

However, when none of these rules apply, for instance with $\boldsymbol{u} = 13$ and $\boldsymbol{v} = 23$, then it will be necessary to directly compare the letters in $\boldsymbol{u}$ and $\boldsymbol{v}$, using for example one of the algorithms described in [19,3].

Fig. 1 shows the computation of $\mathcal{C}$ and $\boldsymbol{x}$ from $\mathcal{T}$. Each insertion (denoted by $\overset{+}{\leftarrow}$) into the heap $\mathcal{H}$ occurs at the lowest level, resulting in a sequence of pairwise comparisons of tuples that restores the heap invariant; that is, that the root entry is minimum. The "`top`" operator removes the current root, after which the heap invariant is similarly restored. Since the occurrence of consecutive g's in $\boldsymbol{x}$ ($\mathcal{C}[i] = \mathcal{T}[i] = g$) leads to no insertion in $\mathcal{H}$, it follows that the maximum number of

```
procedure TC($\mathcal{T}, \mathcal{C}, \boldsymbol{x}, n, k, g$)
  – Compute the reverse V-BWT $\mathcal{C}$ and pointers $\mathcal{P}$.
$k_0 \leftarrow 0$
for $i \leftarrow 1$ to $k$ do
    $\mathcal{C}[i] \leftarrow g$
    if $\mathcal{T}[i] = g$ then
        $k_0 \leftarrow k_0 + 1$;  $\mathcal{P}[i] \leftarrow k_0$
    else
        $\mathcal{H} \overset{+}{\leftarrow} (i, 1, \mathcal{T}[i], \mathcal{T}[i], 1, 1)$
for $i \leftarrow k+1$ to $n$ do
    $(r, \ell, \lambda, \hat{\lambda}, \#, j) \leftarrow \mathtt{top}(\mathcal{H})$
    $\mathcal{C}[i] \leftarrow \lambda$;  $\mathcal{P}[r] \leftarrow i$
    if $\mathcal{T}[i] = g$ then
        $k_0 \leftarrow k_0 + 1$;  $\mathcal{P}[i] \leftarrow k_0$
    else
        UPDATE$(r, \ell, \lambda, \hat{\lambda}, \#, j)$
        $\mathcal{H} \overset{+}{\leftarrow} (r, \ell, \lambda, \hat{\lambda}, \#, j)$
  – Compute $\boldsymbol{x}$ from the pointers $\mathcal{P}$.
$\boldsymbol{x}[n] \leftarrow \mathcal{T}[1]$;  $h \leftarrow 1$
for $i \leftarrow n-1$ downto 1 do
    $h \leftarrow \mathcal{P}[h]$;  $\boldsymbol{x}[i] \leftarrow \mathcal{T}[h]$

procedure UPDATE($r, \ell, \lambda, \hat{\lambda}, \#, j$)
$r \leftarrow i$;  $\ell \leftarrow \ell + 1$;  $\lambda \leftarrow \mathcal{T}[i]$
if $\lambda > \hat{\lambda}$ then
    $\hat{\lambda} \leftarrow \lambda$;  $\# \leftarrow 1$;  $j \leftarrow 1$
elsif $\lambda = \hat{\lambda}$ then
    $\# \leftarrow \# + 1$;  $j \leftarrow 1$
else
    $j \leftarrow j + 1$
```

Fig. 1. Compute $\mathcal{C}$ and the original V-word $\boldsymbol{x}$ from $\mathcal{T}$: heap $\mathcal{H}$ contains the number k' of sequences of g's in $\boldsymbol{x}$, and on exit is empty.

entries is k' and that therefore its maximum height is $\lceil \log_2 k' \rceil$. Thus $\Omega(\log k')$ processing time is required for each of n accesses to $\mathcal{H}$, but it may be as much as $O((n-k)^2/k)$. In the final stage, $\boldsymbol{x}$ is computed by a straightforward **for** loop that requires $\Theta(n)$ time.

To see that the algorithm is correct, observe that at step $i \geqslant k+1$ the i^{th} minimum rotation must be the least rotation of one of the at most k' entries most recently determined – that is, one of the entries beginning with $\mathcal{C}[j]$ and ending with $\mathcal{T}[j]$, $1 \leqslant j \leqslant i-1$. The heap $\mathcal{H}$ keeps track of this minimum by applying the rules (R1)–(R4) to the $\log k'$ pairwise comparisons required for update, and, when necessary, using the linear-time V-order comparison algorithm of [19].

Next consider the entries in the "previous" array $\mathcal{P}$ – the pointer from the previously selected item r to i, the currently selected one. Each row $i \geqslant k+1$ of $R^{\prec}$ is reached from the current minimum entry in $\mathcal{H}$, whose rank is r; thus $\mathcal{P}[r] \leftarrow i$. If in addition $\mathcal{T}[i] = g$, we must link row i to the preceding row $k_0 \leqslant k$ for which $\mathcal{C}[k_0] = g$ by setting $\mathcal{P}[i] \leftarrow k_0$. Exactly k of these latter settings will be made, so that each (perhaps void) sequence of non-g's will relate to each previous such sequence in $\boldsymbol{x}$. Thus following the links of $\mathcal{P}$ yields $\boldsymbol{x}$ in reverse order; in other words $\mathcal{P}$ is an implementation of the usual BWT Last First (LF) mapping for the V-BWT. We have

Theorem 3.2. *Given a string $\boldsymbol{x}[1..n]$ on an ordered alphabet, Algorithm TC correctly computes the inverse $\mathcal{C}$ and the original input V-word $\boldsymbol{x}$ corresponding to the V-BWT $\mathcal{T}$ in time $O(n^2 \log k')$, using $O(n+k')$ words of additional storage, where k' is the number of sequences of largest letters g in $\boldsymbol{x}$.*

Example 3.3. Suppose we are given $\mathcal{T} = 2515355$. Then $k = 4$ and $\mathcal{C}[1..4] = 5^4$. The first two entries in $\mathcal{H}$ will therefore be $\boldsymbol{u} = (1,1,2,2,1,1)$, $\boldsymbol{v} = (3,1,1,1,1,1)$ with $\boldsymbol{v} < \boldsymbol{u}$ by $(R2)$. Since $\mathcal{T}[2] = 5$ and $\mathcal{T}[4] = 5$, therefore $\mathcal{P}[2] \leftarrow 1$ and $\mathcal{P}[4] \leftarrow 2$, respectively.

$i = 5$ $(3,1,1,1,1,1)$ is removed from $\mathcal{H}$ and $\mathcal{C}[5] \leftarrow 1$, $\mathcal{P}[3] \leftarrow 5$. Since $\mathcal{T}[5] = 3$, the entry $(5,2,3,3,1,1)$ will be inserted in $\mathcal{H}$ with $(1,1,2,2,1,1)$ as root.
$i = 6$ $(1,1,2,2,1,1)$ is removed, $\mathcal{C}[6] \leftarrow 2$, $\mathcal{P}[1] \leftarrow 6$. Since $\mathcal{T}[6] = 5$, $\mathcal{P}[6] \leftarrow 3$.
$i = 7$ $(5,2,3,3,1,1)$ is removed, $\mathcal{C}[7] \leftarrow 3$, $\mathcal{P}[5] \leftarrow 7$; since $\mathcal{T}[7] = 5$, $\mathcal{P}[7] \leftarrow 4$.

Thus we find $\mathcal{C} = 5555123$ and $\mathcal{P} = 6152734$. Accordingly we set $\boldsymbol{x}[7] \leftarrow \mathcal{T}[1] = 2$, then $\boldsymbol{x}[6] \leftarrow \mathcal{T}[\mathcal{P}[1]] = \mathcal{T}[6] = 5$, $\boldsymbol{x}[5] \leftarrow \mathcal{T}[\mathcal{P}[6]] = \mathcal{T}[3] = 1$, and so on, yielding $\boldsymbol{x} = 5553152$, the unique V-word implied by $\mathcal{T} = 2515355$.

4. BWT using V-order (arbitrary input)

We now outline the computation of the V-transform and inverse for an arbitrary input which is not necessarily a V-word. The method is derived from Scott's bijective variant of the lexorder Burrows–Wheeler Transform as described by Kufleitner [27].

Since we are dealing with rotations of strings, we also adopt the extension of lexorder $\leqslant$ to an infinite order $\leqslant^{\omega}$ given (with more detail) in [27]. For $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^+$, let $\boldsymbol{u}^{\omega} = \boldsymbol{uuu}\ldots$ be the infinite sequence obtained as the infinite power of $\boldsymbol{u}$; then the ω-lexorder $\boldsymbol{u} \leqslant^{\omega} \boldsymbol{v}$ means that the infinite sequences $\boldsymbol{u}^{\omega}$ and $\boldsymbol{v}^{\omega}$ satisfy $\boldsymbol{u}^{\omega} \leqslant \boldsymbol{v}^{\omega}$.

The bijective lexorder BWT of an arbitrary word $\boldsymbol{x}$ of length n is defined as follows. Let $\boldsymbol{x} = \boldsymbol{v}_s \ldots \boldsymbol{v}_1$ with $\boldsymbol{v}_s \geqslant \cdots \geqslant \boldsymbol{v}_1$ be the Lyndon factorization of $\boldsymbol{x}$. Denote by $\mathcal{R}^{\omega}(\boldsymbol{v}_i)$ the set of rotations (conjugacy class) of $\boldsymbol{v}_i$ in ω-lexorder; $\mathcal{R}^{\omega}(\boldsymbol{x})$ is the set of rotations of the union of the $\mathcal{R}^{\omega}(\boldsymbol{v}_i)$ all in ω-lexorder (ω-lexorder is used to homogenize the dimensions of the $\boldsymbol{v}_i$). Then, the bijective BWT of $\boldsymbol{x}$ is $\mathrm{BWT}(\boldsymbol{x}) = \mathcal{R}^{\omega}(\boldsymbol{x})[i, n]$, for $1 \leqslant i \leqslant n$, that is, the transform is the usual last column of the BWT matrix. The inverse, that recovers the individual factors $\boldsymbol{v}_i$ and hence the input $\boldsymbol{x}$, can be achieved by detecting Lyndon seeds of periodicities in the rows of the reconstructed $\mathcal{R}^{\omega}(\boldsymbol{x})$ – however, we give an alternative method for the V-BWT below. Note that as well as being bijective, no indexes for the least rotations of each of the conjugacy classes, or 'End-of-File' symbols are required.

This concept of constructing a bijective multi-word lexorder BWT was also given earlier by Mantaci et al. [29,30], wherein the input was divided into blocks of equal length. Moreover, Mantaci et al. proposed an extension of lexorder to infinite words, and showed that the order relation $\leqslant^{\omega}$ between two primitive strings $\boldsymbol{u}$ and $\boldsymbol{v}$ can be determined with at most $|\boldsymbol{u}| + |\boldsymbol{v}| - gcd(|\boldsymbol{u}|, |\boldsymbol{v}|)$ letter comparisons. (As shown in [18], to compute the regular V-order $\prec$ of $\boldsymbol{u}$ and $\boldsymbol{v}$, $|\boldsymbol{u}| \leqslant |\boldsymbol{v}|$, requires at most $4|\boldsymbol{u}| + 2|\boldsymbol{v}|$ letter comparisons.)

We now overview Scott's method [22] along with some obvious modifications from lexorder to V-order (see [27] for fuller details). Also, to ease notation, from now on we will let $\mathcal{R}^{\omega}(\boldsymbol{x})$ denote the associated modification to an ω-lex-extension order, and note that in practice the $\mathcal{R}^{\omega}(\boldsymbol{x})$ matrix is not necessarily $n \times n$ here.

(i) Compute the linear-time Lyndon factorization of the input [21,13] $\rightarrow$ linear-time V-word factorization [18,19].

(ii) Compute the multi-word BWT of the Lyndon factors $\rightarrow$ multi-word V-BWT. We process the V-words $\boldsymbol{v}_i$ in $\boldsymbol{x}$, determined by the factorization in (i), from right to left in *groups* G_g of $\boldsymbol{v}_i$ which all have the same maximum letter g – let the V-word $\boldsymbol{v}^*$ have the largest number of V-letters of the $\boldsymbol{v}_i$ in a group. For the multi-word form: construct the conjugacy class $\mathcal{R}^{\omega}(\boldsymbol{v}_i)$ for each V-word $\boldsymbol{v}_i$, where the dimensions of the $\boldsymbol{v}_i$ in a group are first all homogenized to have the same number of V-letters as $\boldsymbol{v}^*$. Therefore the new length of a group is $O(n^2)$. The conjugacy classes from each of the h, say, V-words in a group are then sorted in lex-extension order in linear-time using suffix-sorting techniques from Section 5; hence $O(n^2)$ overall. These h sorted conjugacy classes are then merge sorted into $\mathcal{R}^{\omega}(G_g)$ using linear V-comparison of strings, that is $O(n^2 \log h)O(n) = O(n^3 \log h)$ time for the V-transform of a group.

(iii) Since the conditions of Lemma 2.6(3) are satisfied for the V-ordered conjugates in $\mathcal{R}^{\omega}(G_g)$, we can apply procedure TC for the inversion of each group of $\boldsymbol{x}$ independently; in particular, for each group procedure TC correctly starts by putting all largest letters g in the column C.

To establish that we can invert the groups independently we need Lemma 2.6(1) and the following:

Lemma 4.1. *Let $\boldsymbol{v} \in \mathcal{V}$ with* $\max\{\boldsymbol{v}\} = g$ *and* $|\boldsymbol{v}| = n$. *Suppose that V-BWT$(\boldsymbol{v}) = t_1 t_2 \ldots t_n$. Then $t_n = g$.*

Proof. Suppose not; then the last row r in the BWT matrix $\mathcal{R}^{\prec}$ ends with some $f < g$, and starts with a prefix $\boldsymbol{p}$ (which is non-empty) prior to the first letter g. Consider a rotation of $\boldsymbol{v}$ in $\mathcal{R}^{\prec}$ starting with $f\boldsymbol{p}g$; applying Lemma 2.3, $\boldsymbol{p}g \prec f\boldsymbol{p}g$, and hence the last row r in $\mathcal{R}^{\prec}$, starting $\boldsymbol{p}$ and ending f, is not the largest in $\mathcal{R}^{\prec}$. $\square$

So we apply the $O(n^2 \log k')$ inversion method of Section 3.2 to each group G_g, recovering the V-words in the factorization from smallest to largest in $\mathcal{V}$, that is, from right to left. Note that given the V-word factorization $\boldsymbol{v}_s \geqslant_{\mathcal{V}} \boldsymbol{v}_{s-1} \geqslant_{\mathcal{V}} \cdots \geqslant_{\mathcal{V}} \boldsymbol{v}_1$ of a string, the notation $<_{\mathcal{V}}$ indicates concatenation UMFF-order over $\mathcal{V}$ (hence $\geqslant_{\mathcal{V}}$ is factorization); although for Lyndon words the concatenation order is the same as that for selecting a conjugate, namely lexorder, these orders are not the same in the case of V-words – see [14,19] for more details on the concatenation UMFF-order of $\mathcal{V}$; see [15,17] for more on UMFF-order in general. The inversion procedure induces cycles of letters via the LF mapping, one for each V-word. In the case of multiple identical rows in $\mathcal{R}^{\omega}(\boldsymbol{x})$, when forming an LF cycle, from the rows currently available the one with the smallest index is chosen.

The following lemma establishes the disjoint nature of the LF cycles, where by an *element* E_i^j we mean the j-th consecutive pair of letters in the LF mapping of a V-word $\boldsymbol{v}_i$.

Lemma 4.2. *Let $\boldsymbol{v}_s \geqslant_{\mathcal{V}} \boldsymbol{v}_{s-1} \geqslant_{\mathcal{V}} \geqslant \cdots \geqslant_{\mathcal{V}} \boldsymbol{v}_1$ be the V-word factorization of a string $\boldsymbol{x}$ of length n, and let $LF(\boldsymbol{v}_i)$ be the LF mapping of $\boldsymbol{v}_i$, with elements E_i^j, $1 \leqslant j \leqslant n$. For $\boldsymbol{v}_{i+1} \geqslant_{\mathcal{V}} \boldsymbol{v}_i$, if E_i^j is the prefix of row r_s in $\mathcal{R}^{\omega}(\boldsymbol{x})$, E_{i+1}^j the prefix of row r_t in $\mathcal{R}^{\omega}(\boldsymbol{x})$, and if $E_i^j = E_{i+1}^j$, then $s < t$.*

Proof. In the case of $\boldsymbol{v}_{i+1} = \boldsymbol{v}_i$, this is immediate from repeatedly choosing, for equivalent rows, the row with the smallest index to construct the LF mapping. Otherwise, for $\boldsymbol{v}_{i+1} >_{\mathcal{V}} \boldsymbol{v}_i$, the index of the row with prefix $\boldsymbol{v}_i$ in $\mathcal{R}^{\omega}(\boldsymbol{x})$ is smaller than the index of the row with prefix $\boldsymbol{v}_{i+1}$. Let $\boldsymbol{u}, \boldsymbol{w}$ be two arbitrary rows in $\mathcal{R}^{\omega}(\boldsymbol{x})$ ending λ, μ, with indexes p, q $(p < q)$ respectively, hence $\boldsymbol{u} \prec \boldsymbol{w}$. Write $\boldsymbol{u}^p, \boldsymbol{w}^p$ for the prefix of $\boldsymbol{u}$, $\boldsymbol{w}$ without the last letter λ, μ respectively. If λ and μ are distinct then the LF function maps these suffix letters to equivalent prefix letters, that is distinct rotations in $\mathcal{R}^{\omega}(\boldsymbol{x})$.

So suppose that $\lambda = \mu$, and assume that $\boldsymbol{u}, \boldsymbol{w}$ are in V-form (3). With reference to (3), if the determining point for deciding $\boldsymbol{u} \prec \boldsymbol{w}$ occurred between $\boldsymbol{u}_t$ and $\boldsymbol{w}_t$, for $0 \leqslant t \leqslant k-1$, then Lemma 2.5 applies and $\lambda \boldsymbol{u}^p \prec \mu \boldsymbol{w}^p$. Otherwise suppose that the decision point is at $t = k$. If $\lambda = \mu < g$, then if either both λ, μ or neither were deleted then they are not involved in the final V-order decision, and again $\lambda \boldsymbol{u}^p \prec \mu \boldsymbol{w}^p$; otherwise, consider the suffixes $g \ldots \gamma\lambda$ and $g \ldots \delta\mu$, then one deletion of λ or μ implies that $\gamma > \lambda$ and $\delta \leqslant \lambda$, say, and hence the strings cannot become equal under a sequence of $\star$ deletions, so this case is impossible. Finally, if $\lambda = \mu = g$, then $\boldsymbol{u}_k = \boldsymbol{w}_k = \varepsilon$ in (3), and if the decision point is between $\boldsymbol{u}_{k-1}$ and $\boldsymbol{w}_{k-1}$, then $\lambda = \mu$ will be the last deletions (see Section 2) and hence not involved with the V-ordering.

Therefore, the LF function maps the first occurrence of λ as a suffix to the first occurrence of λ as a prefix in $\mathcal{R}^{\omega}(\boldsymbol{x})$ – in general the i^{th} occurrences. □

Hence choosing the first currently available occurrence in $\mathcal{R}^{\omega}(\boldsymbol{x})$ for the LF mapping of a V-word guarantees remaining in the cycle for that word; once completed, that cycle is effectively deleted from $\mathcal{R}^{\omega}(\boldsymbol{x})$, and the process iterates. Since we can deal with inverting factors in groups, within which LF cycles are disjoint, the inversion of Section 3.2 applies with overall complexity $O(n^2 \log k')$ for a group.

Theorem 4.3. *The multi-word V-BWT is bijective.*

Proof. The proof derives from several results. Firstly, the V-word factorization of a given string $\boldsymbol{x}$ is unique [14,15], and hence the multi-words formed from $\boldsymbol{x}$, each with an associated conjugacy class, are likewise unique. Furthermore, the factors in a V-factorization are non-increasing ($\geqslant_{\mathcal{V}}$) and partially determined by their first letter [14,19] – hence the groups G_g can be inverted independently. Within a group, Lemma 4.2 shows that the LF cycles are disjoint – hence the inversion procedure TC and Theorem 3.2 apply. □

We illustrate the inversion of the multi-word V-BWT with the following example.

Example 4.4. Consider the input string $\boldsymbol{x} = 323132412$ and its unique V-word factorization $32 \geqslant_{\mathcal{V}} 3132 \geqslant_{\mathcal{V}} 412$. After forming powers of V-letters where necessary to homogenize the dimensions for a group, the conjugacy class for each factor is then:

$\mathcal{R}^{\omega}(412)$:	$\mathcal{R}^{\omega}(3132)$:	$\mathcal{R}^{\omega}(32)$:
412	3132	32(32)
124	1323	23(23)
241	3231	
	2313	

Ordering all the conjugates for each group into lex-extension order gives $\mathcal{R}^{\omega}(\boldsymbol{x})$:

$\mathcal{R}^{\omega}(\boldsymbol{x})$:
412
241
124
——————— (This separates the groups: G_3 and G_4.)
3132
3231
32(32)
1323
2313
23(23)

Then $\mathcal{T}$, that is, V-BWT(323132412) = 214212333.

For the inverse, $\mathcal{T}$ is scanned from right to left detecting maximal letters: 3 and 4. We first find the inverse of group G_4, where the existence of one maximal letter 4 indicates that this factor is a V-letter, and so by applying Lemma 2.11, the inverse is given simply by reading off letters in the transform $\mathcal{T}$: 412.

For the inverse of the lower group G_3, using the method in Section 3.2:

After the first **for** loop:	After the second **for** loop:
3 - - 2	3 - - 2
3 - - 1	3 - - 1
3 - - 2	3 - - 2
- - - 3	1 - - 3
- - - 3	2 - - 3
- - - 3	2 - - 3

Once the left-hand column C is complete, two cycles of LF elements can be detected starting from the top row in G_3: (23 - 31 - 13 - 32) and (23 - 32); using the ordering of factors in a factorization, we recover $32 \geqslant_{\mathcal{V}} 3132$.

5. Sorting the rotations of a *V*-word

Following the algorithm of Ko and Aluru [26], we now describe how to sort the set of rotations R of a V-word (or any string) into the set $R^{\prec}$ of the rotations in increasing V-order in $O(n)$ time. We partition the set R into two subsets: rotations starting with g called G, and those not starting with g called $\overline{G}$; applying Lemma 2.6, we sort G first as these are the lesser rows in V-order, followed by sorting $\overline{G}$ to complete the construction of the sorted set $R^{\prec}$.

5.1. Sorting the set G

The first row in $R^{\prec}$ is the given input $\boldsymbol{x}$ which is assumed to be a V-word (Definition 2.7) in $\mathcal{V}$. Therefore $\boldsymbol{x}$ is in V-form (4), and by Theorem 2.10, $\boldsymbol{x}$ has the form of a Lyndon word under lexicographic extension $\prec_{LEX}$. Note that if $\boldsymbol{x}$ is a V-letter, then applying Lemma 2.11, the V-transform is obtained simply by reversing $\boldsymbol{x}$, and likewise to recover $\boldsymbol{x}$ from the V-transform. More generally, Lemma 2.12 shows that $\boldsymbol{x}$ precedes any of its suffixes, hence we can apply a method for sorting suffixes composed of V-letters so as to V-order R into $R^{\prec}$.

However, the following example shows that, unlike Lyndon words, the V-order of two suffixes of a V-word is not the same as the V-order of the rotations that they belong to.

Example 5.1. Let the V-word $\boldsymbol{v} = 9191919293$, and consider two suffixes, 919293 and 91919293. From Lemma 2.3, $919293 \prec 91919293$. Consider the rotations of $\boldsymbol{v}$ for which these suffixes are the prefix, namely 9192939191 and 9191929391 respectively. With reference to their V-form (4), the number of maximal letters in both strings is $k = 5$. Hence we can apply Lemma 2.6(3) with lex-extension ordering, giving $9192939191 \succ 9191929391$.

Due to Lemma 2.6(3), and since we are really interested in ordering rotations of a string, where each rotation has the same number of maximal letters, we assume here that each suffix is concatenated with (an invisible) suffix, which is its prefix in the input $\boldsymbol{x}$. Hence in this case of strings derived from rotations, we can assume lex-extension ordering, and so we would decide in the above example that $91919293 \prec_{LEX} 919293$ ($91919293[91] \prec 919293[9191]$).

Furthermore, since, as for Lyndon words, the Flight Deck property holds for V-words, the linear suffix array construction of Ko and Aluru [26] (which hinges on identifying locally minimal/maximal suffixes) can be modified from lexorder to V-order basically by interchanging single letters for V-letters (essentially just a change of alphabet). We now outline the key steps of this modification, using the notation of Ko and Aluru, thus also providing a summary of their clever technique.

We assume that the input string $\boldsymbol{x}$ is a text $T = t_1t_2 \dots t_n$ over the alphabet $\Sigma = \{1 < 2 < \cdots < n\}$ which is augmented by a least 'End-of-File' character $\$ = 0$ (we assume $\$ \prec \varepsilon$ for the case $\boldsymbol{x}_i = \varepsilon$ in (4)) which occurs uniquely in $\boldsymbol{x}$; we assume the V-letter $g\$$ has been appended to $\boldsymbol{x}$, specifically $\boldsymbol{x} = x_1 \dots x_{n-2}g\$$. Expressing $\boldsymbol{x}$ in its V-form (4), we have $\boldsymbol{x} = g\boldsymbol{x}_1 \dots g\boldsymbol{x}_{m-1}g\boldsymbol{x}_m$, where $m < n$. Then let $T_i = g\boldsymbol{x}_i \dots g\boldsymbol{x}_m$ denote the i-th suffix (of V-letters) of $\boldsymbol{x} = T$, represented by the starting position of $g\boldsymbol{x}_i$ in T, say i', where $i' \geqslant i$.

We divide all suffixes of T into two types, those S smaller than their righthand neighbour and those L larger than their righthand neighbour:

- Type S suffixes $= \{T_i\colon\ T_i \prec_{LEX} T_{i+1}\}$;
- Type L suffixes $= \{T_j\colon\ T_j \succ_{LEX} T_{j+1}\}$.

The last suffix, $T_m = g\$$, is both type S and L. The type of each suffix in T can be determined in one scan of the string. This follows from the corresponding result, Lemma 1 in [26], along with lex-extension Definition 2.9, together with linear V-order comparison of V-letter substrings [18]. When scanning for types S and L we identify two further features in T (w.l.o.g. assume the S suffixes are not more in number):

- Type S positions – position i' of T is a type S position if the associated suffix T_i is of type S; from the Flight Deck property, these include the starting positions of each $g\boldsymbol{x}_1$ in T.

- Type S substrings – the substring $t_{i'} \dots t_{j'}$ is called a type S substring if both i' and j' are type S positions, and every position between i' and j' is not a type S position (that is, it is either a type L position or not the starting position of a V-letter).

Given a text T, the Ko–Aluru technique consists of three main phases:

1. Bucket suffixes into an array A according to their first letter – we modify this to bucket suffixes according to their first V-letter in $O(n)$.
2. Sort all suffixes of one of the types S or L in $O(n)$ (assumed to be the one with least, which w.l.o.g. is S (see [26] for details of when L is least)) – modified here from lexorder to lex-extension order. A linear scan of the sorted list of S suffixes is performed whereby the S suffixes are moved to the current end of their buckets, and hence into their correct positions in A.
3. Using a linear scan of A, the lexorder of all suffixes in T is obtained from the sorted S suffixes in step (2), modified here to lex-extension; within V-letter buckets, suffixes are ordered according to their type, S or L (Lemma 5.4).

In order to sort all the type S suffixes in T for step (2), the type S substrings (which are prefixes of S suffixes) are first sorted according to lex-extension by defining buckets of identical substrings of V-letters. These sorted buckets are numbered using consecutive integers starting from 1, which generates a new string T' of bucket numbers (in the order in which the type S substrings appear in T). Note that each type S suffix in T naturally corresponds to a suffix in the new string T', which is of length at most $\lceil n/2 \rceil$; T' is sorted recursively with an additional cost of $O(n)$ for sorting type S substrings using Bucket Sort (implemented with an initial pre-sorting on a finite alphabet of the V-letters in T).

We now relate suffix-sorting in the text string T to suffix-sorting in the string T' of bucket numbers.

Lemma 5.2. *Let T_i and T_j be two suffixes of T and let $T'_{i'}$ and $T'_{j'}$ be the corresponding suffixes of T'. Then, $T_i \prec_{LEX} T_j \Leftrightarrow T'_{i'} < T'_{j'}$.*

Proof. The proof follows from the corresponding result of Ko and Aluru: Lemma 4 [26]. In particular, we replace character with V-letter, substring with substring of V-letters, and lexorder of strings with lex-extension in their proof. □

We further relate the sorting of all type S suffixes in T using lex-extension $\prec_{LEX}$, to sorting all suffixes of bucket numbers in T' using lexorder.

Corollary 5.3. *The sorted lexorder of the suffixes of T' determines the sorted lex-extension order of the type S suffixes of V-letters of T.*

Proof. Since every type S suffix in T starts with a type S substring, there is a one-to-one correspondence between type S suffixes of T and all suffixes of T'. The lexicographical nature of Definition 2.9 then applies, and as shown in Corollary 2 [26], the proof follows from Lemma 5.2. □

The sorted S suffixes are bucketed into the array A, hence at this stage all type S suffixes are correctly sorted in A (the other buckets are not necessarily sorted). We proceed to overview the modifications for step (3). The following lemma, required for this computation, is then immediate from Lemma 2 in [26]; we detail the adaptation of their short proof from lexorder to V-order.

Lemma 5.4. *A type S suffix is greater in lex-extension $\prec_{LEX}$ than a type L suffix that begins with the same first V-letter.*

Proof. We prove this by contradiction. Suppose a type S suffix T_i and a type L suffix T_j are two suffixes that start with the same V-letter $\boldsymbol{v}$, such that $T_i \prec_{LEX} T_j$. We can write $T_i = \boldsymbol{v}\boldsymbol{\alpha}\boldsymbol{v}_1\boldsymbol{\beta}$ and $T_j = \boldsymbol{v}\boldsymbol{\alpha}\boldsymbol{v}_2\boldsymbol{\gamma}$, where $\boldsymbol{v}, \boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are all V-letters with $\boldsymbol{v}_1 \neq \boldsymbol{v}_2$, and $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ are (possibly empty) strings of V-letters.

Case 1: $\boldsymbol{\alpha}$ contains a V-letter other than $\boldsymbol{v}$. Let $\boldsymbol{v}_3$ be the leftmost V-letter in $\boldsymbol{\alpha}$ that is different from $\boldsymbol{v}$. Since T_i is a type S suffix, it follows that $\boldsymbol{v}_3 \succ \boldsymbol{v}$. Similarly, for T_j to be a type L suffix, $\boldsymbol{v}_3 \prec \boldsymbol{v}$, a contradiction.
Case 2: $\boldsymbol{\alpha}$ does not contain any V-letter other than $\boldsymbol{v}$, or is empty. In this case, we have the following: T_i of type $S \Rightarrow \boldsymbol{v}_1 \succeq \boldsymbol{v}$ while T_j of type $L \Rightarrow \boldsymbol{v}_2 \preccurlyeq \boldsymbol{v}$, hence $\boldsymbol{v}_2 \preccurlyeq \boldsymbol{v} \preccurlyeq \boldsymbol{v}_1$. However, $T_i \prec_{LEX} T_j$ implies $\boldsymbol{v}_1 \prec \boldsymbol{v}_2$, a contradiction. □

Similarly, the following corollary is derived from Corollary 1 in [26].

Corollary 5.5. *In the suffix array of T, among all suffixes that start with the same V-letter, the type S suffixes appear after the type L suffixes.*

The array A is scanned from left to right. For each entry $A[i]$, if $T_{A[i]-1}$ is a type L suffix, it is moved to the front of its bucket in A, and the current front is advanced. This step takes $O(n)$ time and is supported by the following adaptation of Lemma 3 in [26], established by similar modifications to the proof as in the above results.

Lemma 5.6. *When the linear scan of A reaches $A[i]$, then suffix $T_{A[i]}$ of V-letters is already in its sorted position in A.*

At the end of this step, the array A contains all the suffixes of T which start with g in sorted V-order, and hence we have sorted all rows in $R^{\prec}$ starting with g in linear time.

5.2. Sorting the set $\overline{G}$

In this case we append a virtual letter g to the beginning of each of the rows in $\overline{G}$, and then apply the method of Section 5.1 (Example 3.1 shows the necessity of sorting $\overline{G}$ independently). In this case, the finite alphabet for pre-sorting comprises all proper suffixes of each of the V-letters in the text T; for each V-letter, the V-order of these suffixes is given by Lemma 2.11. The sorted set G precedes the sorted set $\overline{G}$, and hence overall, V-ordering all the rotations of a string can be achieved in linear time.

6. Conclusion

In this paper we have introduced the V-transform, a bijective variant of the lexicographic Burrows–Wheeler Transform. For this we applied V-words derived from V-order, which are analogous structures to Lyndon words based on lexorder. We have also shown that the Ko–Aluru linear suffix-sorting technique can be modified from lexorder to V-order, which we then applied to V-order the rotations of a string.

In some cases the V-BWT produces better clustering of letters than the original BWT. The maximal letters g play a significant role in V-order, so consider the case of clustering the k maximal letters g of a string $\boldsymbol{x}$ into g^k in a transform. For the classic lexorder BWT to achieve g^k, it is required that the substrings $\boldsymbol{v}_i$ of $\boldsymbol{x}$ between g's $(g\boldsymbol{v}_i g)$ occur *compactly* in lexorder, meaning that no other rotations of $\boldsymbol{x}$ lie within this lexordering which would split up the g's in the transform. However, to get g^k with the V-BWT, only a weaker condition overall is required, namely that the conditions of Lemma 2.6 are satisfied for compact lex-extension ordering of the $\boldsymbol{v}_i$ – these features are illustrated in Example 3.1. Similarly the weaker conditions underlie that, for the V-word 5215125432, the lexorder transform is 5253145122, whereas the V-transform 2122315545 gives slightly better clustering.

Since, due to fast suffix-sorting, computing both the BWT and V-BWT for a V-word is linear, the type of transform which gives better clustering for an input string can be determined in $O(n)$ time, for instance by applying run-length encoding to the text transformations. For lossless compression and other applications requiring inversion, with the V-BWT we are thus offering a trade-off between: instances of better clustering in the transform versus complexity (the inverse V-BWT computation is supralinear).

We conclude by proposing some future lines of enquiry:

Question 6.1. Which transform gives the best degree of clustering for which types of input data? For instance, the lexorder transform of the V-letter string 54324321 is 23^24^2251; applying Lemma 2.11, the V-transform is the reversed string 12342345 – hence no new clustering although any existing clustering is not reduced, for instance the (reversed) given substring $(234)^2$. For the V-word string 414141414243, the lexorder transform is 444444311112, while the V-transform is 311112444444; both of these transforms, while distinct, have produced perfect clustering of letters. The V-transform of the V-word 521512521522 is $2^21^22^315^4$, better clustered than the lexorder transform $52^25^321^22^21$.

Question 6.2. If we define the transform to be the last column of V-letters in $R^{\prec}$, rather than single letters, does this lead to efficiencies?

Question 6.3. Can the inverse V-BWT calculation be performed in close to linear time in the worst case?

Question 6.4. With reference to the effective strategy of [5,1], where it is shown that re-ordering the letters of the alphabet achieves superior compression for the lexorder BWT compared to using the original ordering of the letters, in particular grouping the vowels at the start of the alphabet: which alphabet re-ordering algorithm can similarly achieve improved compression for the V-BWT?

V-words and Lyndon words are instances of Hybrid Lyndons [19]. We suggest further research into Burrows–Wheeler type transforms for other Hybrid Lyndons, for example [20]. Overall, the goal is a taxonomy of BWT type transforms. This leads naturally to considering further adaptations of suffix array techniques for other forms of order. Finally, we propose that parallelism is worth investigating for computing the V-BWT (see [16] for an optimal PRAM V-word factorization algorithm) and related novel transformations.

Acknowledgements

We thank Travis Gagie, Juha Kärkkäinen and Jouni Sirén for fruitful discussions. We also thank the anonymous referees for their very thorough reviews and valued comments and suggestions which stimulated us to achieve these results. The work of the second author was supported in part by Grant No. 8180-2012 from the Natural Sciences and Engineering Research Council of Canada.

References

[1] J. Abel, W. Teahan, Universal text preprocessing for data compression, IEEE Trans. Comput. 54 (5) (2005) 497–507.
[2] D. Adjeroh, T. Bell, A. Mukherjee, The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching, Springer, 2008, 352 pp.
[3] A. Alatabbi, J.W. Daykin, M.S. Rahman, W.F. Smyth, Simple linear comparison of strings in V-order, in: Proc. 8th International Workshop on Algorithms and Computation (WALCOM), in: Lecture Notes in Comput. Sci., vol. 8344, 2014, pp. 80–89.
[4] M. Burrows, D. Wheeler, A block sorting lossless data compression algorithm, Technical report 124, Digital Equipment Corporation, 1994.
[5] B. Chapin, S.R. Tate, Higher compression from the Burrows–Wheeler transform by modified sorting, in: Proc. 1998 Data Compression Conf. (DCC), 1998, p. 532.
[6] M. Chemillier, Periodic musical sequences and Lyndon words, Soft Comput. 8 (9) (2004) 611–616, http://dx.doi.org/10.1007/s00500-004-0387-2.
[7] M. Chemillier, C. Truchet, Computation of words satisfying the "rhythmic oddity property" (after Simha Arom's works), Inform. Process. Lett. 86 (2003) 255–261.
[8] K.T. Chen, R.H. Fox, R.C. Lyndon, Free differential calculus IV – The quotient groups of the lower central series, Ann. Math. 68 (1958) 81–95.
[9] M. Crochemore, J. Désarménien, D. Perrin, A note on the Burrows–Wheeler transformation, Theoret. Comput. Sci. 332 (1–3) (2005) 567–572.
[10] M. Crochemore, R. Grossi, J. Kärkkäinen, G.M. Landau, A constant-space comparison-based algorithm for computing the Burrows–Wheeler transform, in: Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM), 2013, pp. 74–82.
[11] T.-N. Danh, D.E. Daykin, The structure of V-order for integer vectors, in: A.J.W. Hilton (Ed.), Congressus Numerantium, vol. 113, Utilitas Mat. Pub. Inc., Winnipeg, Canada, 1996, pp. 43–53.
[12] T.-N. Danh, D.E. Daykin, Ordering integer vectors for coordinate deletions, J. Lond. Math. Soc. (2) 55 (1997) 417–426.
[13] D.E. Daykin, Algorithms for the Lyndon unique maximal factorization, J. Combin. Math. Combin. Comput. 77 (2011) 65–74.
[14] D.E. Daykin, J.W. Daykin, Lyndon-like and V-order factorizations of strings, J. Discrete Algorithms 1 (2003) 357–365.
[15] D.E. Daykin, J.W. Daykin, Properties and construction of unique maximal factorization families for strings, Internat. J. Found. Comput. Sci. 19 (4) (2008) 1073–1084.
[16] D.E. Daykin, J.W. Daykin, C.S. Iliopoulos, W.F. Smyth, Generic algorithms for factoring strings, in: H. Aydinian, F. Cicalese, C. Deppe (Eds.), Information Theory, Combinatorics, and Search Theory (In Memory of Rudolf Ahlswede), in: Lecture Notes in Comput. Sci., vol. 7777, 2013, pp. 402–418.
[17] D.E. Daykin, J.W. Daykin, W.F. Smyth, Combinatorics of unique maximal factorization families (UMFFs), in: R. Janicki, S.J. Puglisi, M.S. Rahman (Eds.), Special Issue on Stringology, Fund. Inform. 97 (3) (2009) 295–309.
[18] D.E. Daykin, J.W. Daykin, W.F. Smyth, String comparison and Lyndon-like factorization using V-order in linear time, in: R. Giancarlo, G. Manzini (Eds.), Proc. 22nd Annual Symposium on Combinatorial Pattern Matching (CPM), in: Lecture Notes in Comput. Sci., vol. 6661, 2011, pp. 65–76.
[19] D.E. Daykin, J.W. Daykin, W.F. Smyth, A linear partitioning algorithm for Hybrid Lyndons using V-order, Theoret. Comput. Sci. 483 (2013) 149–161.
[20] J.W. Daykin, et al., Computing the Burrows–Wheeler transform using binary orders, in preparation.
[21] J.P. Duval, Factorizing words over an ordered alphabet, J. Algorithms 4 (1983) 363–381.
[22] J.Y. Gil, D.A. Scott, A bijective string sorting transform, arXiv:1201.3077, 2012.
[23] J. Kärkkäinen, Fast BWT in small space by blockwise suffix sorting, Theoret. Comput. Sci. 387 (3) (2007) 249–257.
[24] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: Proc. 30th Internat. Conf. Automata, Languages & Programming, 2003, pp. 943–955.
[25] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (6) (2006) 918–936.
[26] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, in: Proc. 14th Annual Symposium on Combinatorial Pattern Matching, 2003, pp. 200–210.
[27] M. Kufleitner, On bijective variants of the Burrows–Wheeler transform, Proc. Stringology (2009) 65–79.
[28] M. Lothaire, Combinatorics on Words, 2nd edition, Addison–Wesley, Reading, MA, 1983; Cambridge University Press, Cambridge, 1997.
[29] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows Wheeler transform and applications to sequence comparison and data compression, in: Alberto Apostolico, Maxime Crochemore, Kunsoo Park (Eds.), Combinatorial Pattern Matching, Proceedings of the 16th Annual Symposium, CPM 2005, Jeju Island, Korea, 19–22 June 2005, in: Lecture Notes in Comput. Sci., vol. 3537, Springer, 2005, pp. 178–189.
[30] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows–Wheeler transform, Theoret. Comput. Sci. 387 (3) (2007) 298–312.
[31] G. Nong, S. Zhang, W.H. Chan, Linear suffix array construction by almost pure induced-sorting, in: Proc. 2009 Data Compression Conf. (DCC), 2009, pp. 193–202.
[32] C. Reutenauer, Free Lie Algebras, Lond. Math. Soc. Monogr. New Ser., vol. 7, Oxford University Press, 1993, 288 pp.
[33] M. Salson, T. Lecroq, M. Léonard, L. Mouchard, A four-stage algorithm for updating a Burrows–Wheeler transform, Theoret. Comput. Sci. 410 (43) (2009) 4350–4359, http://dx.doi.org/10.1016/j.tcs.2009.07.016.
[34] B. Smyth, Computing Patterns in Strings, Pearson, 2003, 423 pp.