Computation of the suffix array, burrows-wheeler transform and FM-index in V-order

Daykin, Jacqueline; Mhaskar, Neerja; Smyth, W. F.

Published in:
Theoretical Computer Science

DOI:
10.1016/j.tcs.2021.06.004

Publication date:
2021

Citation for published version (APA):
Daykin, J., Mhaskar, N., & Smyth, W. F. (2021). Computation of the suffix array, burrows-wheeler transform and FM-index in V-order. *Theoretical Computer Science*, *880*, 82-96. https://doi.org/10.1016/j.tcs.2021.06.004

Computation of the Suffix Array, Burrows-Wheeler Transform and FM-Index in V-Order*

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

[a] *Department of Computer Science, Aberystwyth University, Wales*
[b] *Department of Information Science, Stellenbosch University, South Africa*
[c] *Department of Computing and Software, McMaster University, Canada*
[d] *School of Engineering & Information Technology, Murdoch University, Australia*
[e] *Department of Informatics, King's College London, UK*

Abstract

V-order is a total order on strings that determines an instance of Unique Maximal Factorization Families (UMFFs), a generalization of Lyndon words. The fundamental V-comparison of strings can be done in linear time and constant space. V-order has been proposed as an alternative to lexicographic order (lex-order) in the computation of suffix arrays and in the suffix-sorting induced by the Burrows-Wheeler transform (BWT). In line with the recent interest in the connection between suffix arrays and Lyndon factorization, in this paper we obtain similar results for the V-order factorization. Indeed, we show that the results describing the connection between suffix arrays and Lyndon factorization are matched by analogous V-order processing. We also describe a methodology for efficiently computing the FM-Index in V-order, as well as V-order substring pattern matching using backward search.

Keywords: Burrows-Wheeler transform, Combinatorics, FM-index, Lexorder, String comparison, Substring pattern matching, Suffix sorting, V-BWT, V-order.

*The paper is an outgrowth of a contribution to WALCOM 2019, published in [1].

*Corresponding author.
Email addresses: jackie.daykin@gmail.com (Jacqueline W. Daykin), pophlin@mcmaster.ca (Neerja Mhaskar), smyth@mcmaster.ca (W. F. Smyth)

1. Introduction

This paper extends current knowledge on applications of a non-lexicographic global order known as V-order [2] (see Definition 2). It is an intriguing question, now investigated for more than 20 years, whether such a counter-intuitive ordering might nevertheless yield algorithmic efficiencies, or other theoretical/computational benefits, compared to lexicographic order (*lexorder*). We begin by mentioning some applications already considered.

Analogous to lexicographic comparison, the central problem of efficient V-ordering of strings was considered in [3, 4, 5, 6, 7], culminating in a remarkably simple, linear time, constant space comparison algorithm [8], further improved in [9]. In work closely related to ideas in this paper, [3, 4] also described efficient Lyndon-like factorization of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ into V-words (Definition 11).

Other V-order applications include a variant (V-BWT or V-transform) of the standard lexicographic Burrows-Wheeler transform, introduced in [10], where instances of enhanced data clustering were demonstrated. Also described in [10] was a linear-time algorithm for V-sorting all the conjugates of a string, based on the work in [3, 4] and the linear-time suffix-sorting algorithm of Ko-Aluru [11]. In particular, [10] showed how to compute the V-transform of a V-word $\boldsymbol{x}$ in $\Theta(n)$ time and space; in addition, inverting the V-transform to recover the original $\boldsymbol{x}$ was achieved in time $O(n^2 \log k)$, using $O(n + k)$ words of additional storage, where k is the number of sequences of the largest letter in $\boldsymbol{x}$.

In this paper, we modify ideas given in [12] that relate Lyndon factorization to suffix arrays and the Burrows-Wheeler transform in order to obtain similar results for the V-order factorization (Section 4). We go on to introduce FM-index type functions in V-order (Section 5), thus raising the possibility of directly applying them to pattern matching for letters in V-order, as done for lexorder in [12]. We conclude by applying the backward search technique to substring pattern matching in V-order (Section 6). The differences between lexorder and V-order are intriguing: while the latter generally appears trickier to work with,

on the other hand computing suffix arrays in V-order is trivial.

More generally, several recent papers have revealed deep connections between suffix arrays and Lyndon decomposition that are still not well understood. For example, in the first step of his linear-time non-recursive suffix array algorithm, Baier[13] actually computes a sorted version of the *Lyndon array* $\mathrm{LA}_{\boldsymbol{x}}$ of string $\boldsymbol{x}$, where $\mathrm{LA}_{\boldsymbol{x}}[i]$ identifies the longest Lyndon word occurring at $\boldsymbol{x}[i]$. The same structure arises in the recent proof [14] that the number of maximal periodicities ("runs") in a string is less than its length. For further commentary see [15, 16, 17].

In this paper we begin the process of exploring analogous connections in the context of V-order.

2. Preliminaries

We are given a finite totally ordered set Σ of cardinality $\sigma = |\Sigma|$, called the *alphabet*, whose elements are *characters* (equivalently *letters*). A *string* (equivalently *word*) is a sequence of zero or more characters over Σ. (In our examples, we shall often suppose that $\Sigma = \{a, b, c, \dots\}$, the Roman alphabet in its natural order, or $\Sigma = \{1, 2, 3, \dots\}$, the natural numbers.) A string $\boldsymbol{x} = x_1 x_2 \cdots x_n$ of *length* $|\boldsymbol{x}| = n$ is represented by $\boldsymbol{x}[1..n]$, where $x_i = \boldsymbol{x}[i] \in \Sigma$ for $1 \le i \le n$. The set of all nonempty strings over the alphabet Σ is denoted by Σ^+. The *empty string* of zero length is denoted by $\boldsymbol{\varepsilon}$, with $\Sigma^* = \Sigma^+ \cup \boldsymbol{\varepsilon}$. If $\boldsymbol{x} = \boldsymbol{u}\boldsymbol{w}\boldsymbol{v}$ for strings $\boldsymbol{u}, \boldsymbol{w}, \boldsymbol{v} \in \Sigma^*$, then $\boldsymbol{u}$ is a *prefix*, $\boldsymbol{w}$ a *substring* or *factor*, and $\boldsymbol{v}$ a *suffix* of $\boldsymbol{x}$. We denote the suffix starting at position i, $1 \le i \le n$ by S_i. For a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and an integer sequence $0 < i_1 < i_2 < \cdots < i_k < n+1$, the string $\boldsymbol{y} = \boldsymbol{x}[i_1]\boldsymbol{x}[i_2] \cdots \boldsymbol{x}[i_k]$ is said to be a *subsequence* of $\boldsymbol{x}$, *proper* if $k < n$.

If $\boldsymbol{x} = \boldsymbol{u}^k$ (a concatenation of k copies of $\boldsymbol{u}$) for some nonempty string $\boldsymbol{u}$ and some integer $k > 1$, then $\boldsymbol{x}$ is said to be a *repetition*; otherwise, $\boldsymbol{x}$ is *primitive*. A string $\boldsymbol{y} = R_i(\boldsymbol{x})$ is the i^{th} *conjugate* (or *rotation*) of $\boldsymbol{x} = \boldsymbol{x}[1..n]$ if $\boldsymbol{y} = \boldsymbol{x}[i{+}1..n]\boldsymbol{x}[1..i]$ for some $0 \le i < n$ (so that $R_0(\boldsymbol{x}) = \boldsymbol{x}$).

Note that all strings are written in mathbold: $\boldsymbol{x}$, $\boldsymbol{w}$ instead of x, w.

We shall be considering various standard arrays associated with strings. The *suffix array* $\mathrm{SA} = \mathrm{SA}_{\boldsymbol{x}}[1..n]$ of $\boldsymbol{x}$ is an array of integers $1..n$ such that for every $i \in 1..n$, $\mathrm{SA}[i] = j$ if and only if S_j is the i^{th} suffix in lexorder. For example, the suffixes of $\boldsymbol{x} = acab$ are $ab, acab, b, cab$ in lexorder, so that $\mathrm{SA}_{\boldsymbol{x}} = (3, 1, 4, 2)$.

The *Burrows-Wheeler transform* $\mathrm{BWT} = \mathrm{BWT}_{\boldsymbol{x}}$ of a given string $\boldsymbol{y}$ [18] is a permutation formed in two equivalent ways from the string $\boldsymbol{x} = \boldsymbol{y}\$$[1], where \$ is a *sentinel* letter assumed, for processing convenience, to be smaller than any letter in Σ:

(1) Form a matrix $\mathrm{M}_{\boldsymbol{x}}^{lex}$ of the rotations of $\boldsymbol{x}$ sorted in increasing lexorder, then select the final column; or

(2) Compute the suffix array $\mathrm{SA}_{\boldsymbol{x}}$, then for $i \leftarrow 1$ **to** $n+1$, set $\mathrm{BWT}[i] \leftarrow \boldsymbol{x}\big[\mathrm{SA}[i]-1\big]$ whenever $\mathrm{SA}[i] > 1$; $\mathrm{BWT}[i] \leftarrow \boldsymbol{x}[n+1] = \$$, otherwise.

Thus, given $\boldsymbol{x} = acab\$$, we effectively work from

$$\mathrm{M}_{\boldsymbol{x}}^{lex} = \begin{array}{ccccc} \underline{\$} & a & c & a & \boldsymbol{b} \\ \underline{a} & \underline{b} & \underline{\$} & a & \boldsymbol{c} \\ \underline{a} & \underline{c} & \underline{a} & \underline{b} & \boldsymbol{\$} \\ \underline{b} & \underline{\$} & a & c & \boldsymbol{a} \\ \underline{c} & \underline{a} & \underline{b} & \underline{\$} & \boldsymbol{a} \end{array} \tag{1}$$

choosing $\mathrm{BWT}_{\boldsymbol{x}} = bc\$aa$ from the last column. The underlined prefixes in the table correspond to the suffix array entries $\mathrm{SA}[1..5] = (5, 3, 1, 4, 2)$.

In practice, method (2) is always used, since it requres only $\Theta(n)$ time, compared to $\Theta(n^2)$ for method (1). As in the example, the BWT often has the useful property that equal letters are grouped together.

The *FM-Index* [19] is defined in terms of two functions:

- $C_{\boldsymbol{x}}(\ell)$ gives, for each letter $\ell \in \Sigma$ that occurs in $\boldsymbol{x}$, the number of letters in $\boldsymbol{x}$ smaller in lexorder than ℓ. For $\boldsymbol{x} = acab$, $C_{\boldsymbol{x}}(a) = 0, C_{\boldsymbol{x}}(b) = 2, C_{\boldsymbol{x}}(c) = 3$.

[1] The introduction of the \$ ensures that the two methods are indeed equivalent.

- $\text{RANK}_{\boldsymbol{x}}(p, \ell)$ gives the number of occurrences of letter ℓ in the prefix of $\boldsymbol{x}$ of length p. Thus RANK_{acab} takes the following values:

p/ℓ	$\underline{1}$	$\underline{2}$	$\underline{3}$	$\underline{4}$
a	1	1	2	2
b	0	0	0	1
c	0	1	1	1

The functions C and RANK can be efficiently computed and are useful in the processing of the BWT, as discussed in Section 5.

Finally, a *Lyndon word* [20] is a primitive word that is minimum in lexorder $<$ over its conjugacy class. The following Lyndon factorization (*LF*) theorem is fundamental in stringology and will be important in many of our applications:

Theorem 1. *[20] Any word $\boldsymbol{x}$ can be written uniquely as a product $LF_{\boldsymbol{x}} = \boldsymbol{x} = \boldsymbol{u}_1\boldsymbol{u}_2\cdots\boldsymbol{u}_k$ of Lyndon words, with $(\boldsymbol{u}_1 \geq \boldsymbol{u}_2 \geq \cdots \geq \boldsymbol{u}_k)$.*

For further stringological definitions and theory, see [21, 22].

3. Results on V-Order

We now define a non-lexorder called V-order and describe some of its notable properties. Let $\boldsymbol{x} = x_1x_2\cdots x_n$ be a string over Σ. Define $h \in \{1, \ldots, n\}$ by $h = 1$ if $x_1 \leq x_2 \leq \cdots \leq x_n$; otherwise, by the unique value such that $x_{h-1} > x_h \leq x_{h+1} \leq x_{h+2} \leq \cdots \leq x_n$. Let $\boldsymbol{x}^* = x_1x_2\cdots x_{h-1}x_{h+1}\cdots x_n$, where the star * indicates deletion of x_h. Write $\boldsymbol{x}^{s*} = (...(\boldsymbol{x}^*)^*...)^*$ with $s \geq 0$ stars. Let $g = \max\{x_1, x_2, \ldots, x_n\}$, and let k be the number of occurrences of g in $\boldsymbol{x}$. Then the sequence $\boldsymbol{x}, \boldsymbol{x}^*, \boldsymbol{x}^{2*}, ...$ ends $g^k, ..., g^1, g^0 = \boldsymbol{\varepsilon}$. From all strings $\boldsymbol{x}$ over Σ we form the *star tree*, where each string $\boldsymbol{x}$ labels a vertex and there is a directed edge upward from $\boldsymbol{x}$ to $\boldsymbol{x}^*$, with the empty string $\boldsymbol{\varepsilon}$ as the root.

Definition 2 (V-order [2])**.** *We define V-order $\prec$ between distinct strings $\boldsymbol{x}$,*

$\boldsymbol{y}$, *where* $\boldsymbol{x}$, $\boldsymbol{y}$ *do not contain* \$[2]. *First* $\boldsymbol{x} \prec \boldsymbol{y}$ *if in the star tree* $\boldsymbol{x}$ *is in the path* $\boldsymbol{y}, \boldsymbol{y}^*, \boldsymbol{y}^{2*}, \ldots, \boldsymbol{\varepsilon}$. *If* $\boldsymbol{x}, \boldsymbol{y}$ *are not in a path, there exist smallest* s, t *such that* $\boldsymbol{x}^{(s+1)*} = \boldsymbol{y}^{(t+1)*}$. *Let* $\boldsymbol{s} = \boldsymbol{x}^{s*}$ *and* $\boldsymbol{t} = \boldsymbol{y}^{t*}$; *then* $\boldsymbol{s} \neq \boldsymbol{t}$ *but* $|\boldsymbol{s}| = |\boldsymbol{t}| = m$ *say. Let* $j \in [1..m]$ *be the greatest integer such that* $\boldsymbol{s}[j] \neq \boldsymbol{t}[j]$. *If* $\boldsymbol{s}[j] < \boldsymbol{t}[j]$ *in* Σ *then* $\boldsymbol{x} \prec \boldsymbol{y}$; *otherwise,* $\boldsymbol{y} \prec \boldsymbol{x}$. *Clearly* $\prec$ *is a total order on all strings in* Σ^*.

Example 3. *[Star tree] We illustrate the star tree for the* V*-order comparison of the words* $\boldsymbol{x} = lexorder$ *and* $\boldsymbol{y} = matrix$. *The subscript* h *indicates the* V *letter to be deleted (defined above as* $x_{h-1} > x_h \leq x_{h+1} \leq x_{h+2} \leq \cdots \leq x_n$*). The circled letters are those compared in alphabetic order (defined above as* $\boldsymbol{s}[j] \neq \boldsymbol{t}[j]$*).*

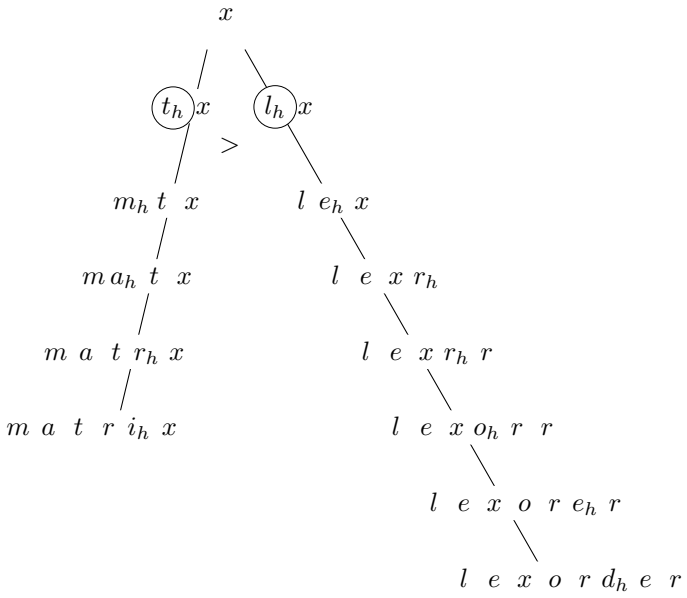

Figure 1: *lexorder* $\prec$ *matrix*

An instance where $\boldsymbol{x}$ and $\boldsymbol{y}$ occur in the same path in the star tree is given by $\boldsymbol{x} = 42527$, $\boldsymbol{y} = 57$, yielding $\boldsymbol{x}^* = 4257$, $\boldsymbol{x}^{2*} = 457$, $\boldsymbol{x}^{3*} = 57$ and so $57 \prec 42527$.

[2]It turns out that, in V-order, it is useful to require, not only that \$ be the smallest letter, but moreover that \$ $\prec \boldsymbol{\varepsilon}$. To avoid this situation, we transform strings containing \$ into equivalent \$-free strings to which Definition 2 applies. See Definition 17.

Definition 4 (V***-form*** [2, 23, 3, 4])**.** *The V**-form** of a string $\boldsymbol{x}$ is defined as*

$$V_k(\boldsymbol{x}) = \boldsymbol{x} = \boldsymbol{x_0} g \boldsymbol{x_1} g \cdots \boldsymbol{x_{k-1}} g \boldsymbol{x_k}$$

for strings $\boldsymbol{x_i}$, $i = 0, 1, \ldots, k$, where g is the largest letter in $\boldsymbol{x}$ — thus we suppose that g occurs exactly k times. Note that, in this representation, any $\boldsymbol{x_i}$ may be the empty string ε. For clarity, when more than one string is involved, we use the notation $\mathcal{L}_{\boldsymbol{x}} = g$, $\mathcal{C}_{\boldsymbol{x}} = k$.

Lemma 5. *[2, 23, 3, 4] Suppose we are given distinct strings $\boldsymbol{x}$ and $\boldsymbol{y}$ with corresponding V-forms as follows:*

$$\begin{aligned} \boldsymbol{x} &= \boldsymbol{x}_0 \mathcal{L}_{\boldsymbol{x}} \boldsymbol{x}_1 \mathcal{L}_{\boldsymbol{x}} \boldsymbol{x}_2 \cdots \boldsymbol{x}_{j-1} \mathcal{L}_{\boldsymbol{x}} \boldsymbol{x}_j, \\ \boldsymbol{y} &= \boldsymbol{y}_0 \mathcal{L}_{\boldsymbol{y}} \boldsymbol{y}_1 \mathcal{L}_{\boldsymbol{y}} \boldsymbol{y}_2 \cdots \boldsymbol{y}_{k-1} \mathcal{L}_{\boldsymbol{y}} \boldsymbol{y}_k, \end{aligned}$$

where $j = \mathcal{C}_{\boldsymbol{x}}$, $k = \mathcal{C}_{\boldsymbol{y}}$.

Let $h \in 0..\max(j, k)$ be the least integer such that $\boldsymbol{x}_h \neq \boldsymbol{y}_h$. Then $\boldsymbol{x} \prec \boldsymbol{y}$ if and only if one of the following conditions holds:

(C1) $\mathcal{L}_{\boldsymbol{x}} < \mathcal{L}_{\boldsymbol{y}}$

(C2) $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$ *and* $\mathcal{C}_{\boldsymbol{x}} < \mathcal{C}_{\boldsymbol{y}}$

(C3) $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, $\mathcal{C}_{\boldsymbol{x}} = \mathcal{C}_{\boldsymbol{y}}$ *and* $\boldsymbol{x}_h \prec \boldsymbol{y}_h$.

Observe the recursive nature of determining $\prec$ in (C3); that is, each substring pair $\boldsymbol{x}_h$, $\boldsymbol{y}_h$ can likewise be decomposed into V-forms.

Example 6. *[C1-C3] The following examples illustrate the three conditions in Lemma 5:*

- *Using C1, if $\boldsymbol{x} = 345$ and $\boldsymbol{y} = 456$, we have $\boldsymbol{x} \prec \boldsymbol{y}$ as $\mathcal{L}_{\boldsymbol{x}} = 5 < 6 = \mathcal{L}_{\boldsymbol{y}}$.*
- *Using C2, if $\boldsymbol{x} = 1818181$ and $\boldsymbol{y} = 78787$, we have $\boldsymbol{x} \succ \boldsymbol{y}$ as $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}} = 8$ and $\mathcal{C}_{\boldsymbol{x}} = 3 > 2 = \mathcal{C}_{\boldsymbol{y}}$.*

- *Using C3, if* $\boldsymbol{x} = 9616921$ *and* $\boldsymbol{y} = 9616912$*, we have* $\boldsymbol{x} \prec \boldsymbol{y}$ *because* $\boldsymbol{x}_0 = \boldsymbol{y}_0 = \boldsymbol{\varepsilon}$*,* $\boldsymbol{x}_1 = \boldsymbol{y}_1 = 616$ *while* $\boldsymbol{x}_2 = 21 \prec 12 = \boldsymbol{y}_2$*, since the* V*-form* $\boldsymbol{x}_0$ *prefix of 21 is* $\boldsymbol{\varepsilon}$ *while for 12 it is 1 and* $\boldsymbol{\varepsilon} \prec 1$*. It is instructive to extend this example to* $\boldsymbol{x'} = 961692198888$ *and* $\boldsymbol{y'} = 961691294$*, where* $\boldsymbol{x'} \prec \boldsymbol{y'}$ *since the order is still determined by* $\boldsymbol{x'}_2 = 21$ *and* $\boldsymbol{y'}_2 = 12$*.*

Lemma 7. *[3, 4] For given strings* $\boldsymbol{x}$ *and* $\boldsymbol{y}$*, if* $\boldsymbol{y}$ *is a proper subsequence of* $\boldsymbol{x}$*, then* $\boldsymbol{y} \prec \boldsymbol{x}$*.*

For example, using this lemma we immediately have 9374 $\prec$ 93748336, 937336 $\prec$ 93748336, and 9786 $\prec$ 93748336.

The remarkable Lemma 7 has many consequences, not least the trivial ordering of suffixes in V-order discussed in Section 4.

Lemma 8. *[6, 8] For any strings* $\boldsymbol{u}$*,* $\boldsymbol{v}$*,* $\boldsymbol{x}$*,* $\boldsymbol{y}$*:* $\boldsymbol{x} \prec \boldsymbol{y} \Leftrightarrow \boldsymbol{uxv} \prec \boldsymbol{uyv}$*.*

By Lemma 8, a comparison of two strings in V-order can ignore equal prefixes (or suffixes). Therefore, on this basis, Algorithm COMPARE($\boldsymbol{x}$,$\boldsymbol{y}$), first presented in [8] and later upgraded in [9], ignores common prefixes of strings $\boldsymbol{x}$ and $\boldsymbol{y}$ while comparing them. COMPARE also makes use of the following:

Lemma 9. *[8] For any two strings* $\boldsymbol{x}, \boldsymbol{y}$ *with* $\boldsymbol{x} \prec \boldsymbol{y}$ *and any two letters* λ, μ *such that* $\boldsymbol{y} \prec \boldsymbol{x}\lambda$*:*

(i) *if* $\lambda \leq \mu$*, then* $\boldsymbol{x}\lambda \prec \boldsymbol{y}\mu$*;*

(ii) *if* $\lambda > \mu$*, then* $\boldsymbol{y}\mu \prec \boldsymbol{x}\lambda$*.*

In contrast to comments made in the Conclusion of [4], Lemma 9 implies that V-order comparison can be conducted in a positional manner just like lexorder comparison. In fact, COMPARE($\boldsymbol{x}$,$\boldsymbol{y}$) is an on-line algorithm that requires only a one-character window!

The improved V-order comparison algorithm given in [9], called COMPARE-Sensitive, was obtained by simple modifications of COMPARE. COMPARE-Sensitive considers the structure of the input strings in order to achieve improved

running times. A probabilistic argument shows that COMPARE-Sensitive runs faster than COMPARE in almost all cases of interest, at least twice as fast according to the experiments described in [9].

Lemma 10. *[3, 4, 5, 7, 8, 9] V-order comparison of given strings $\boldsymbol{x}$ and $\boldsymbol{y}$ requires linear time and constant space.*

We now introduce the V-order equivalent of the lexorder Lyndon word:

Definition 11 (V-Word [23])**.** *A string $\boldsymbol{x}$ over an ordered alphabet Σ is said to be a* V-word *if it is the unique minimum in V-order $\prec$ in the conjugacy class of $\boldsymbol{x}$.*

Thus, like a Lyndon word, a V-word is necessarily primitive.

Example 12. *[$\prec$] We can apply Definition 2, equivalently the structure given by Lemma 5, to conclude that*

$$7173 \prec 7371 \prec 1737 \prec 3717,$$

so that 7173 *is a V-word, while on the other hand* 1737 *is a Lyndon word. Similarly,* 71727174 *and* 818382 *are V-words, while conjugates 17271747 and 183828 are Lyndon words.*

Now consider two V-words $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ on an alphabet Σ. It may be that $\boldsymbol{x_1 x_2}$ is also a V-word, a fact we record by writing $\boldsymbol{x_1} <_{\mathcal{V}} \boldsymbol{x_2}$; if not, then we write $\boldsymbol{x_1} \geq_{\mathcal{V}} \boldsymbol{x_2}$. Thus, corresponding to the Lyndon factorization into Lyndon words using $\geq$ (Theorem 1), we arrive at a *V-order factorization* expressed in terms of an ordering $\geq_{\mathcal{V}}$ of V-words:

Lemma 13. *[3, 4] Using only linear time and space (see Algorithm VF in [4][3]), a string $\boldsymbol{x}$ can be factored uniquely into V-words $\boldsymbol{x} = \boldsymbol{x_1 x_2 \cdots x_m}$, where by definition $\boldsymbol{x_1} \geq_{\mathcal{V}} \boldsymbol{x_2} \geq_{\mathcal{V}} \cdots \geq_{\mathcal{V}} \boldsymbol{x_m}$.*

[3] Although no claim is made in this reference, VF is an on-line algorithm, since it does no backtracking and outputs the V-word factors in order from left to right.

Example 14. *[$\geq_{\mathcal{V}}$] For* $\boldsymbol{x} = 33132421$*, the Lyndon decomposition is* $3 \geq 3 \geq 13242 \geq 1$*, while the* V*-order factorization identifies nonextendible* V*-words 33132 and 421 with* $33132 \geq_{\mathcal{V}} 421$*. (Note however that* $33132 \prec 421$*! See [24] for more background on this phenomenon.) Similarly, from Example 12, the string*

$$\boldsymbol{x} = \boldsymbol{uvw} = (7173)(71727174)(818382)$$

has the unique V*-order factorization* $\boldsymbol{u} \geq_{\mathcal{V}} \boldsymbol{v} \geq_{\mathcal{V}} \boldsymbol{w}$*, even though* $\boldsymbol{u} \prec \boldsymbol{v} \prec \boldsymbol{w}$*.*

As we now discover, it will also be useful to order strings $\boldsymbol{x}$, $\boldsymbol{y}$ based on a lexicographic approach to their V-order factorizations. We call this ordering, denoted $\prec_{LEX(V)}$, *lex-extension* order, expressed here with respect to V-order factorization, while another choice of factorization is given by the unique V-form of a string.

Definition 15 (Lex-extension order [23, 4, 10])**.** *Suppose that the* V*-order factorization of distinct strings* $\boldsymbol{x}, \boldsymbol{y} \in \Sigma^+$ *yields nonempty factors*

$$\boldsymbol{x} = \boldsymbol{x_1 x_2} \cdots \boldsymbol{x_p}, \quad \boldsymbol{y} = \boldsymbol{y_1 y_2} \cdots \boldsymbol{y_q}.$$

If either

(1) $\boldsymbol{x}$ *is a proper prefix of* $\boldsymbol{y}$ *(that is,* $\boldsymbol{x_i} = \boldsymbol{y_i}$ *for* $1 \leq i \leq p < q$*)* or else

(2) for some $i \in 1..min(\boldsymbol{p}, \boldsymbol{q})$*,* $\boldsymbol{x_j} = \boldsymbol{y_j}$ *for every* $1 \leq j < i$*, and* $\boldsymbol{x_i} \prec \boldsymbol{y_i}$*,*

then we write $\boldsymbol{x} \prec_{LEX(V)} \boldsymbol{y}$*. Otherwise,* $\boldsymbol{x} \succ_{LEX(V)} \boldsymbol{y}$ *(*$\boldsymbol{y} \prec_{LEX(V)} \boldsymbol{x}$*).*

Intuitively, in $\prec_{LEX(V)}$ ordering, we first factor the two strings to be compared into V-factors. Then we treat each V-factor as a letter, and compare strings by comparing the V-factors at the same position. Consider Example 16, where $\boldsymbol{y} \prec \boldsymbol{x}$, but $\boldsymbol{x} \prec_{LEX(V)} \boldsymbol{y}$.

Example 16. *[$\prec_{LEX(V)}$] Given* V*-order factorizations*

$$\boldsymbol{x} = \boldsymbol{x_1 x_2 x_3} = (33132)(422)(5), \ \boldsymbol{y} = \boldsymbol{y_1 y_2} = (33132)(413),$$

we find $\boldsymbol{x_1} = \boldsymbol{y_1}$*, but since* $\boldsymbol{x_2} \prec \boldsymbol{y_2}$*, therefore* $\boldsymbol{x} \prec_{LEX(V)} \boldsymbol{y}$ *(even though* $\boldsymbol{x} \succ \boldsymbol{y}$*).*

4. Computing the Extended Suffix Array in V-Order

In this section we discuss the V-order version, V-BWT, of the Burrows-Wheeler transform [10]. As in the lexorder case, described in Section 2, it is convenient to append a special sentinel letter \$, less in lexorder than every letter in Σ, to each string being processed. However, it turns out that, in V-order, it could become necessary also to require that \$ be less than the empty string $\boldsymbol{\varepsilon}$. For example, this requirement would arise if, while executing a V-order comparison of strings containing \$, an element $\boldsymbol{x_i}$ corresponding to \$ in the V-form representation (Definition 4) were empty.

In order to avoid this situation, we introduce Definition 17, which transforms $\boldsymbol{x}$, $\boldsymbol{y}$, each containing at most a single \$, into \$-free equivalent strings $\boldsymbol{x'}$, $\boldsymbol{y'}$ such that $\boldsymbol{x'} \prec \boldsymbol{y'} \Rightarrow \boldsymbol{x} \prec \boldsymbol{y}$. This is achieved by introducing new letters α and $\alpha' > \alpha$, both less than any letter in Σ, but *not* less than $\boldsymbol{\varepsilon}$, to replace \$ in $\boldsymbol{x}$ and $\boldsymbol{y}$.

Definition 17. *[Transforming strings containing* \$ *to equivalent* \$*-free strings.] Given distinct strings* $\boldsymbol{x}$*,* $\boldsymbol{y}$*, each containing at most one occurrence of* \$*, we construct new strings* $\boldsymbol{x'}$*,* $\boldsymbol{y'}$ *as defined below, by introducing new letters* α *and* $\alpha' > \alpha$*, both less than any letter in* Σ*, to replace* \$*; that is, we replace* $\Sigma \cup \{\$\}$ *by* $\Sigma' = \Sigma \cup \{\alpha, \alpha'\}$*. The following rules are applied:*

1. *If* $\boldsymbol{x}[r] = \boldsymbol{y}[t] = \$$*,* $1 \le r < t \le n$*:*
   (a) $\boldsymbol{x'} \leftarrow \boldsymbol{x}[1..r-1]\alpha\boldsymbol{x}[r+1..|\boldsymbol{x}|]$ *(*α *replaces* \$*),*
   $\boldsymbol{y'} \leftarrow \boldsymbol{y}[1..r-1]\alpha'\boldsymbol{y}[r..|\boldsymbol{y}|]$ *(insert* α'*);*
   (b) $\boldsymbol{y'} \leftarrow \boldsymbol{y'}[1..t]\alpha\boldsymbol{y'}[t+2..|\boldsymbol{y'}|]$ *(*α *replaces* \$*),*
   if $|\boldsymbol{x'}| \ge t$ *then* $\boldsymbol{x'} \leftarrow \boldsymbol{x'}[1..t]\alpha'\boldsymbol{x'}[t+1..|\boldsymbol{x'}|]$ *(insert* α'*).*
2. *If* $\boldsymbol{x}[r] = \$$*,* $1 \le r \le |\boldsymbol{x}|$*, and* $\forall i, 1 \le i \le |\boldsymbol{y}| : \boldsymbol{y}[i] \ne \$$*:*
   $\boldsymbol{x'} \leftarrow \boldsymbol{x}[1..r-1]\alpha\boldsymbol{x}[r+1..|\boldsymbol{x}|]$*, (*$\alpha$ *replaces* \$*),*
   if $|\boldsymbol{y}| \ge r-1$ *then* $\boldsymbol{y'} \leftarrow \boldsymbol{y}[1..r-1]\alpha'\boldsymbol{y}[r..|\boldsymbol{y}|]$ *(insert* α'*).*

3. *If* $\boldsymbol{x}[r] = \boldsymbol{y}[r] = \$$, $1 \leq r \leq |\boldsymbol{x}|, |\boldsymbol{y}|$*:*
   $\boldsymbol{x'} \leftarrow \boldsymbol{x}[1..r-1]\boldsymbol{x}[r+1..|\boldsymbol{x}|]$, $\boldsymbol{y'} \leftarrow \boldsymbol{y}[1..r-1]\boldsymbol{y}[r+1..|\boldsymbol{y}|]$
   (remove $\$$ *from both* $\boldsymbol{x}$ *and* $\boldsymbol{y}$*).*

Then $\boldsymbol{x'} \prec \boldsymbol{y'} \Rightarrow \boldsymbol{x} \prec \boldsymbol{y}$.

Since the ordering of letters between the strings $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{x'}$, $\boldsymbol{y'}$ is preserved, and because $\alpha < \alpha'$, both less than every element in Σ, the ordering between strings $\boldsymbol{x}, \boldsymbol{y}$ and strings $\boldsymbol{x'}, \boldsymbol{y'}$, respectively, is preserved. Thus we extend the application of Definition 2 to comparison of distinct strings $\boldsymbol{x}$, $\boldsymbol{y}$, each containing at most one occurrence of $\$ \prec \boldsymbol{\varepsilon}$. This also ensures the inequalities between strings are well defined in all cases. Furthermore, since strings $\boldsymbol{x'}$ and $\boldsymbol{y'}$ are strings over an alphabet Σ' which does not include \$, and V-order (Definition 2) is defined for arbitrary alphabets (excluding \$), all known results on V-order, including all the lemmas presented in this paper, apply immediately to $\boldsymbol{x'}$ and $\boldsymbol{y'}$. In particular, the transitivity property for V-order on Σ naturally holds on Σ' also.

We apply Definition 17 to the example strings $\boldsymbol{x} = \$acab$ and $\boldsymbol{y} = acab\$$ introduced above. From Rule 1(a), we get $\boldsymbol{x'} = \alpha acab$ and $\boldsymbol{y'} = \alpha' acab\$$; next, applying 1(b), we find $\boldsymbol{x'} = \alpha acab\alpha'$ and $\boldsymbol{y'} = \alpha' acab\alpha$. Then using the star tree construction of Definition 2 to compare $\boldsymbol{x'}$ and $\boldsymbol{y'}$, we find as above that $s = \boldsymbol{x'}^{3*} = \alpha ac$ and $t = \boldsymbol{y'}^{3*} = \alpha' ac$, yielding $\boldsymbol{x'} \prec \boldsymbol{y'}$. Hence, by Definition 17, $\boldsymbol{x} \prec \boldsymbol{y}$.

Applying this methodology to $\boldsymbol{x} = acab\$$, we can sort all its conjugates into V-order – $cab\$a \prec \$acab \prec acab\$ \prec b\$aca \prec ab\$ac$ – yielding the V-order Burrows-Wheeler matrix $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{x}}$ instead of the lexorder version $\mathrm{M}^{lex}_{\boldsymbol{x}}$ given in (1):

$$\mathrm{M}^{\mathcal{V}}_{\boldsymbol{x}} = \begin{array}{ccccc} c & a & b & \$ & a \\ \$ & a & c & a & b \\ a & c & a & b & \$ \\ b & \$ & a & c & a \\ a & b & \$ & a & c \end{array} \tag{2}$$

Finally we form V-BWT$_{\boldsymbol{x}} = ab\$ac$ from the last column L of $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{x}}$.

We go on now to introduce the extended suffix array and its V-order implementation, making use of the V-BWT in a manner suggested recently by the lexorder methodology of Mantaci *et al.* [12].

We begin by showing how to translate the results of [12] from lexorder into V-order. These authors describe a strategy for obtaining the suffix array SA$_{\boldsymbol{x}}$ of a string $\boldsymbol{x}$ from its Lyndon factorization LF$_{\boldsymbol{x}}$. To do the equivalent calculation in V-order — SA$_{\boldsymbol{x}}$ from the V-order factorization of $\boldsymbol{x}$ (Lemma 13) — is not however of interest, because, as remarked earlier, sorting the suffixes of $\boldsymbol{x}$ in V-order is trivial! By virtue of Lemma 7, the V-order of the suffixes is

$$\boldsymbol{x}[n] \prec \boldsymbol{x}[n-1..n] \prec \cdots \prec \boldsymbol{x}[1..n],$$

so that $\mathrm{SA}_{\boldsymbol{x}}[i] = n - i + 1$ for all $\boldsymbol{x}$.

Thus, in the V-order version of this problem, we replace the suffixes with the "extended suffixes" (conjugates of $\boldsymbol{x}$) employed to determine the V-BWT. Then the i-th *extended suffix* is defined by

$$ES_i \equiv \boldsymbol{x}[i..n]\boldsymbol{x}[1..i-1]. \tag{3}$$

The following example, from [10], shows that the ordering of the extended suffixes is no longer straightforward:

Example 18. *For* $\boldsymbol{x} = 9191919293$*, consider suffixes* $S_5 = 919293$ *and* $S_3 = 91919293$*, with* $S_5 \prec S_3$ *by Lemma 7. However, extending these suffixes and applying Lemma 5 (C3) yields the opposite order:*

$$\underline{9191929}391 \prec \underline{9192939}191 \Longleftrightarrow ES_3 \prec ES_5.$$

Thus we show here how to use the V-order factorization of $\boldsymbol{x}$ (in particular, the on-line algorithm VF mentioned in Lemma 13) to generate the Burrows-Wheeler matrix $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{x}}$ and thus the corresponding V-BWT$_{\boldsymbol{x}}$. We call the cor-

responding suffix array EVSA = EVSA$_{\boldsymbol{x}}$, equivalent to the rotations of $\boldsymbol{x}$ as ordered in $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{x}}$. For example, in Equation (2), for $\boldsymbol{x'} = acab\$$, EVSA$_{\boldsymbol{x'}} = (2, 5, 1, 4, 3)$ and $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{x'}}$ can be derived from each other, each yielding the conclusion that V-BWT$_{\boldsymbol{x'}} = ab\$ac$.

We remark that the difference between Lyndon and V-order factorization of $\boldsymbol{x}$ [3, 4] provides more options for efficient string processing. Examples 12 and 14 give an idea of the variation that can occur between lexorder and V-order.

Now we turn to the idea of "compatibility" of sorted suffixes as introduced in [12]. Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string and $\boldsymbol{u} = \boldsymbol{x}[i..j]$, $1 \leq i \leq j \leq n$, a substring of $\boldsymbol{x}$. Then the sorting of suffixes $\boldsymbol{s_p} = \boldsymbol{x}[p..j]$, $\boldsymbol{s_q} = \boldsymbol{x}[q..j]$ of $\boldsymbol{u}$ is *compatible* with the sorting of the corresponding suffixes $\boldsymbol{x}[p..n]$, $\boldsymbol{x}[q..n]$ of $\boldsymbol{x}$ if these two (p, q) pairs have the same order in both $\boldsymbol{u}$ and $\boldsymbol{x}$. In lexorder, compatibility of arbitrarily chosen $\boldsymbol{u}$ and $\boldsymbol{v}$ does not always hold [12], but does hold when they are substrings of Lyndon factors in the Lyndon factorization of $\boldsymbol{x}$. However, in V-order, by Lemma 7, compatibility holds for every choice of p, q. Moreover, the shorter suffix is always lesser, thus allowing comparison in terms of indexes:

Lemma 19. *Let $\boldsymbol{x} \in \Sigma^+$ and $\boldsymbol{u}$ be a substring of $\boldsymbol{x}$ with $\boldsymbol{s_1}$ a suffix of $\boldsymbol{u}$. If $\boldsymbol{s_2}$ is a proper suffix of $\boldsymbol{s_1}$ then $\boldsymbol{s_2} \prec \boldsymbol{s_1}$ with respect to both $\boldsymbol{u}$ and $\boldsymbol{x}$.*

Proof. Consider the suffixes $\boldsymbol{s_1 t_1}$ and $\boldsymbol{s_2 t_2}$ of $\boldsymbol{x}$ for possibly empty $\boldsymbol{t_1}$, $\boldsymbol{t_2}$. Applying Lemma 7, we find that $\boldsymbol{s_2} \prec \boldsymbol{s_1}$ with respect to $\boldsymbol{u}$ and $\boldsymbol{s_2 t_2} \prec \boldsymbol{s_1 t_1}$ with respect to $\boldsymbol{x}$, as required. □

As we have seen, Lemma 19 is not sufficient for extended suffixes, but since each conjugate has the same number of maximum g's, condition (C3) implicitly applies. However, as a special case of this lemma, we obtain the following V-order analogy of the result (Theorem 3.2) given in [12] for Lyndon decomposition:

Corollary 20. *Let $\boldsymbol{x} \in \Sigma^+$ with V-order factorization $\boldsymbol{x} = \boldsymbol{v_1} \cdots \boldsymbol{v_k}$, and let $\boldsymbol{u} = \boldsymbol{v_i} \cdots \boldsymbol{v_j}$, for $1 \leq i \leq j \leq k$. Then the sorting of the suffixes of $\boldsymbol{u}$ is compatible with the sorting of the corresponding suffixes of $\boldsymbol{x}$.*

The intricate differences between lexorder and V-order, along with the superficial nature of the V-order suffix array, lead to the following lemma that aims toward incrementally building the extended V-order suffix array, EVSA, in a manner similar to that employed in [12] for building SA. Thus, for $k = 2$, the next lemma partly describes the structure of $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{u}}$, hence of $\mathrm{EVSA}_{\boldsymbol{u}}$:

Lemma 21. *Let $\boldsymbol{u} = \boldsymbol{v_1 v_2}\$ \in \Sigma^+$, where $\boldsymbol{v_1}$, $\boldsymbol{v_2}$ are V-words with $\boldsymbol{v_1} \geq_{\mathcal{V}} \boldsymbol{v_2}$ and $\mathcal{L}_{\boldsymbol{v_1}} = \mathcal{L}_{\boldsymbol{v_2}} = \mathcal{L}$. Let $R_0, R_1, \ldots, R_{\mathcal{C}_{\boldsymbol{v_2}}}$ be the $\mathcal{C}_{\boldsymbol{v_2}} + 1$ consecutive rotations of $\boldsymbol{v_2}\$$ occurring in the V-order Burrows-Wheeler matrix $M^{\mathcal{V}}_{\boldsymbol{v_2}\$}$ that start with $\$$ or $\mathcal{L}$. Then in $M^{\mathcal{V}}_{\boldsymbol{u}}$:*

(a) for every $j \in 0..\mathcal{C}_{\boldsymbol{v_2}}$, there exists a unique row E_r, with (nonempty) prefix P_r and (possibly empty) suffix S_r, such that $P_r S_r = R_j$ and $S_r P_r = \boldsymbol{v_2}\$$;

(b) E_0 has prefix $\$$ and the remaining $\mathcal{C}_{\boldsymbol{v_2}}$ entries R_j identified in (a) occur in rows $E_j, 1 \leq j \leq \mathcal{C}_{\boldsymbol{v_2}}$, in ascending order of j.

Proof.

For both cases (a) and (b), appending $\$$ to $\boldsymbol{v_2}$ ensures that working with suffix arrays is equivalent to working with $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{u}}$ and similarly for $\mathrm{EVSA}_{\boldsymbol{u}}$ (see commentaries on the sentinel $\$$ in Section 2 and prior to Equation 2).

(a) This follows from Corollary 20 and the fact that $M^{\mathcal{V}}_{\boldsymbol{u}}$ includes all the rotations of $\boldsymbol{u}$.

(b) Since the first row of $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{u}}$ must be least in V-order, and $\$ \prec \boldsymbol{\varepsilon}$ and $\boldsymbol{v_1}$ being a V-word beginning with $\mathcal{L}$, it has prefix $\$$ — this unique row is identified as E_0.

For the sake of rigour, consider converting to $\$$-free strings. Since $\boldsymbol{u}$ contains exactly one $\$$, then it can only appear in the first position in any row of $M^{\mathcal{V}}_{\boldsymbol{u}}$, once. Also since $\boldsymbol{v_1}$ is a V-word it must begin with $\mathcal{L}$. So let r be the row of $M^{\mathcal{V}}_{\boldsymbol{u}}$ starting $\$\mathcal{L}$. Comparing row r with other rows in $M^{\mathcal{V}}_{\boldsymbol{u}}$, and applying Definition 17 part 1, row r now starts with α and every other row starts with α', where $\alpha < \alpha'$. Consider the $\boldsymbol{x_0}$ prefix for the V-form

of each row in $M^{\mathcal{V}}_{\boldsymbol{u}}$, then for row r we have $\boldsymbol{x_0} = \alpha$ and hence it is unique out of all the $\boldsymbol{x_0}$ substrings. Furthermore, each of the other $\boldsymbol{x_0}$ substrings must contain some letter greater than α, so that Lemma 5 (C1) applies, thus establishing that $r = E_0$. We now return to working with \$.

Consider those suffixes P_r in the EVSA suffix array of $\boldsymbol{v_2}\$$ which start with \$ or $\mathcal{L}$, and suppose that they are V-ordered so that their subscripts are in ascending order of j. The rows E_r in $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{u}}$ are then identified as those of the form $P_r\boldsymbol{v_1}S_r$.

Suppose that $P_k \prec P_l$ which is determined by applying lex-extension Definition 15 with the factorizations given by the V-forms of P_k and P_l. Since $\boldsymbol{v_2}\$$ only contains one \$, then part 1 of this definition does not apply, for if P_k was a proper prefix of P_l then P_l would contain two \$'s. From part 2 there is a pair of substrings $\boldsymbol{x_i}$, $\boldsymbol{y_i}$ with minimal i where $\boldsymbol{x_i}$ is in P_k and $\boldsymbol{y_i}$ is in P_l with $\boldsymbol{x_i} \prec \boldsymbol{y_i}$. Then $P_k \prec P_l$ implies $P_k\boldsymbol{v_1}S_k \prec P_l\boldsymbol{v_1}S_l$. In other words, the V-order of $P_k\boldsymbol{v_1}S_k$ and $P_l\boldsymbol{v_1}S_l$ is determined before the start of $\boldsymbol{v_1}$ by the order of P_k and P_l which both end in \$. Hence the order of the P_r in $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{v_2}\$}$ is preserved in $\mathrm{M}^{\mathcal{V}}_{\boldsymbol{u}}$.

Overall, we are essentially merging two lists of suffixes each ordered in lex-extension order into a further lex-extension ordered list of suffixes, hence maintaining the lists' original orders. □

We illustrate Lemma 21 with the following example:

Example 22. *Let V-words $\boldsymbol{v_1} = 83$ and $\boldsymbol{v_2} = 88182$, where $\boldsymbol{v_1} \geq_{\mathcal{V}} \boldsymbol{v_2}$, $\mathcal{L}_{\boldsymbol{v_1}} = \mathcal{L}_{\boldsymbol{v_2}} = 8$, and $\boldsymbol{u} = 8388182\$$. The resulting matrices are given below, where the rotations R_j (starting with \$ or $\mathcal{L} = 8$) are indicated in $M^{\mathcal{V}}_{\boldsymbol{v_2}\$}$ and the P_iS_i are*

underlined in $M^{\mathcal{V}}_{\boldsymbol{u}}$.

$$
M^{\mathcal{V}}_{\boldsymbol{v_2}\$} = \begin{array}{ll} R_0: & \$\ 8\ 8\ 1\ 8\ 2 \\ R_1: & 8\ 8\ 1\ 8\ 2\ \$ \\ R_2: & 8\ 1\ 8\ 2\ \$\ 8 \\ R_3: & 8\ 2\ \$\ 8\ 8\ 1 \\ & 1\ 8\ 2\ \$\ 8\ 8 \\ & 2\ \$\ 8\ 8\ 1\ 8 \end{array} \tag{4}
$$

$$
M^{\mathcal{V}}_{\boldsymbol{u}} = \begin{array}{cccccccc} \underline{\$} & 8 & 3 & \underline{8} & \underline{8} & \underline{1} & \underline{8} & \underline{2} \\ \underline{8} & \underline{8} & \underline{1} & \underline{8} & \underline{2} & \underline{\$} & 8 & 3 \\ \underline{8} & \underline{1} & \underline{8} & \underline{2} & \underline{\$} & 8 & 3 & \underline{8} \\ \underline{8} & \underline{2} & \underline{\$} & 8 & 3 & \underline{8} & \underline{8} & \underline{1} \\ 8 & 3 & 8 & 8 & 1 & 8 & 2 & \$ \\ 1 & 8 & 2 & \$ & 8 & 3 & 8 & 8 \\ 2 & \$ & 8 & 3 & 8 & 8 & 1 & 8 \\ 3 & 8 & 8 & 1 & 8 & 2 & \$ & 8 \end{array} \tag{5}
$$

Equipped with Corollary 20 and Lemma 21, we can modify the clever incremental suffix-sorting/BWT strategy of [12] to construct the extended V-order suffix array, EVSA, of $\boldsymbol{x} = \boldsymbol{x}[1..n]$. We outline the steps:

Step 1: Identify the first factor $\boldsymbol{v_1}$ in the V-order factorization $\boldsymbol{v_1} \geq_{\mathcal{V}} \cdots \geq_{\mathcal{V}} \boldsymbol{v_k}$ of $\boldsymbol{x}$ in linear time [3, 4]; we assume the maximum letter in each factor is $\mathcal{L}$.

Step 2: Compute the lex-extension order suffix array $\text{EVSA}_{\boldsymbol{v_1}\$}$ of $\boldsymbol{v_1}\$$ in linear time [10]. In practice we will only need the suffixes starting with $\mathcal{L}$ in our pattern matching application in Section 6.

Step 3: Extract $\text{BWT}_{\boldsymbol{v_1}\$}$ from $\text{EVSA}_{\boldsymbol{v_1}\$}$.

Step 4: For factor $\boldsymbol{v_i}[1..t]$, $1 < i \leq k$, insert each suffix $\boldsymbol{v_i}[t], \boldsymbol{v_i}[t{-}1..t], \ldots, \boldsymbol{v_i}[1..t]$ into the current EVSA in its V-order — so for $i = 2$, one by one the

suffixes of $\boldsymbol{v_2}\$$ are inserted into $\text{EVSA}_{\boldsymbol{v_1}\$}$ giving the EVSA for $\boldsymbol{v_1 v_2}\$$. Further, as each $\boldsymbol{v_i}$ is processed we can extract $\text{BWT}_{\boldsymbol{v_1}\cdots\boldsymbol{v_i}\$}$: for a suffix $\boldsymbol{x}[i..j]$ the BWT letter is $\boldsymbol{x}[i-1]$.

Note: After the above steps all the V-word factors are incrementally processed and V-$\text{BWT}_{\boldsymbol{x}}$ computed. To reduce complexity, the method avoids merging the suffix arrays of both the extended suffix array computed to date and that of the current factor being processed.

We now describe suffix insertion (Step 4) in more detail. For this we apply Lemma 3.16 in [4] which states that, unlike Lyndon words, the order of the set $\mathcal{V}$ of V-words is in some cases the same as V-order while in other cases it is reversed. This leads to two cases for Step 4: first, the next factor to be processed has a larger maximum letter $\mathcal{L}$ than those in all factors processed so far, and second, the maximum letters are the same. However, the following example shows that the method doesn't work in the first case of distinct maximal letters — hence they are restricted to be equal in Lemma 21.

Example 23. *Let* $\boldsymbol{u} = \boldsymbol{v_1} \geq_{\mathcal{V}} \boldsymbol{v_2}$ *with* $\boldsymbol{v_1} = 321$ *and* $\boldsymbol{v_2} = 5152$. *The ordered conjugates of* $\boldsymbol{v_1}$ *are:* $321 \prec 132 \prec 213$, *but note that the order of these conjugates is changed after processing* $\boldsymbol{v_2}$. *We list the ordered conjugates of* $\boldsymbol{u} = 3215152$ *below with complete factors shown in square brackets and the conjugates of 321 underlined.*

5152[321]
52[321]51
<u>1</u>[5152]<u>32</u>
152[321]5
<u>21</u>[5152]<u>3</u>
<u>321</u>[5152]
2[321]515

On the other hand, as we have seen, we are constrained to extended suffixes

relating to a conjugacy class where each conjugate has the same maximal letter $\mathcal{L}$ which occurs with the same frequency. In particular, Lemma 21 and condition (C3) ensure that the order of an existing suffix array is preserved during this iterative processing of factors for those suffixes starting with $\mathcal{L}$; therefore we now go on to deal with this case.

Suppose a prefix $\boldsymbol{p}$ of p factors of $\boldsymbol{x}$ has been processed resulting in EVSA($\boldsymbol{p}$) with an associated set of ordered conjugates, where all conjugates have the same maximum letter. The task is now to insert the suffixes of factor $\boldsymbol{v_{p+1}}$ into EVSA($\boldsymbol{p}$). Applying properties established in [10], if $g = \mathcal{L}_{\boldsymbol{w}}$ and $k = \mathcal{C}_{\boldsymbol{w}}$ for a string $\boldsymbol{w}$ (which does not have the sentinel \$ appended), then in V-order the first k conjugates of $\boldsymbol{w}$ start with g. Processing will be as follows:

- First all existing suffixes computed prior to processing $\boldsymbol{v_{p+1}}$ will have $\boldsymbol{v_{p+1}}$ appended to them.
- The technique for calculating the index for inserting each proper suffix of $\boldsymbol{v_{p+1}}$ (increasing one V-form $\boldsymbol{x_i}$ type substring at a time from right to left) into EVSA is given with the FM-index in Section 5.
- Inserting the last improper suffix, $\boldsymbol{v_{p+1}}$, is straightforward. We establish that, analogous to the lexorder case in [12], this suffix is least in V-order and can be entered directly into the extended suffix array. Consider $\boldsymbol{v_1}$: since it is a V-word it is least in EVSA($\boldsymbol{v_1}$) and likewise for $\boldsymbol{v_2}$. If we suppose that $\boldsymbol{v_1} \neq \boldsymbol{v_2}$, then $\boldsymbol{v_2 v_1} \in \mathcal{V}$ (see the generalized form of properties of V-words described by Theorem 2.7(3) in [4]), and $\boldsymbol{v_2 v_1}$ is least in EVSA($\boldsymbol{v_2 v_1}$), and further, $\boldsymbol{v_2} \prec \boldsymbol{v_2 v_1}$. Similarly, if $\boldsymbol{v_1} = \boldsymbol{v_2}$, then $\boldsymbol{v_2} \prec \boldsymbol{v_2 v_2}$. So suppose this assumption holds for all the first $p+1$ factors. Then additionally, by Lemma 7, we have $\boldsymbol{v_{p+1}} \prec \boldsymbol{v_1 \cdots v_p v_{p+1}}$ holding in the extended suffix array. This property holds iteratively as we continue processing the factors of $\boldsymbol{x}$.

As expressed in [12] for the Lyndon case, this technique is suitable for integration with the on-line V-order factoring algorithm: suffix-sorting can proceed

in tandem as soon as the first V-factor is identified.

5. Computing the FM-Index in V-Order

Here we describe algorithms to compute the components of the FM-Index in V-order. We assume that for a given string $\boldsymbol{x}$, the V-BWT matrix $\mathrm{M}_{\boldsymbol{x}}^{\mathcal{V}}$ (sorted conjugates in V-order) has been computed using the methodology described in Section 4. Since the FM-index stores the first (denoted F) and last (denoted L) columns of the Burrows-Wheeler matrix efficiently, our ultimate goal is to implement, in the context of V-order, BWT-type pattern matching using FM-index functions VC and VRANK analogous to those defined in Section 2.

It will be convenient for this discussion to assume that the given string $\boldsymbol{x}$ is defined on an integer alphabet $\Sigma = \{0, 1, \ldots, g\}$, where the least letter 0 corresponds to the sentinel letter \$, thus ensuring that every rotation of the string is distinct ($\boldsymbol{x}$ is not a repetition). In order to adapt $RANK$ and C from lexorder to V-order, we first establish a collection of results, Lemmas 24 – 29, on properties of $\mathrm{M}_{\boldsymbol{x}}^{\mathcal{V}}$, and illustrated by references to the example $\boldsymbol{x} = \boldsymbol{w'}$ in Figure 2, which shows $\mathrm{M}_{\boldsymbol{w'}}^{\mathcal{V}}$ corresponding to the following string:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{w'} =$	1	9	2	3	9	2	6	5	9	2	3	0

Lemma 24. *The Last First Mapping holds for V-order.*

Proof. We will establish the claim using the well-known BWT context sorting argument.

Given a string $\boldsymbol{x}$, let $\boldsymbol{r_i} = \boldsymbol{u}\lambda$ and $\boldsymbol{r_j} = \boldsymbol{v}\lambda$ be a pair of distinct rotations of $\boldsymbol{x}$ occurring in the rows i, j of $M_{\boldsymbol{x}}^{\mathcal{V}}$ such that $\boldsymbol{r_i} \prec \boldsymbol{r_j}$, $\boldsymbol{u}, \boldsymbol{v} \neq \boldsymbol{\varepsilon}$ and $\lambda \in \Sigma$. The context therefore is given by $\boldsymbol{u}$ and $\boldsymbol{v}$.

By Lemma 8 we have $\boldsymbol{u}\lambda \prec \boldsymbol{v}\lambda \implies \boldsymbol{u} \prec \boldsymbol{v}$ and further $\boldsymbol{u} \prec \boldsymbol{v} \implies \lambda\boldsymbol{u} \prec \lambda\boldsymbol{v}$. That is, the relative order of rows $\boldsymbol{r_i}$ and $\boldsymbol{r_j}$ is maintained under the rotation of the letter λ.

G_{-1}	9	2	3	9	2	6	5	9	2	3	0	1
	9	2	3	0	1	9	2	3	9	2	6	5
	9	2	6	5	9	2	3	0	1	9	2	3
G_1	0	$\underline{1}$	9	2	3	9	2	6	5	9	2	3
	$\underline{1}$	9	2	3	9	2	6	5	9	2	3	0
G_3	$\underline{3}$	9	2	6	5	9	2	3	0	1	9	2
	$\underline{3}$	0	1	9	2	3	9	2	6	5	9	2
	2	$\underline{3}$	9	2	6	5	9	2	3	0	1	9
	2	$\underline{3}$	0	1	9	2	3	9	2	6	5	9
G_5	$\underline{5}$	9	2	3	0	1	9	2	3	9	2	6
G_6	$\underline{6}$	5	9	2	3	0	1	9	2	3	9	2
	2	$\underline{6}$	5	9	2	3	0	1	9	2	3	9

Figure 2: $M^{\mathcal{V}}_{\boldsymbol{w'}}$ for $\boldsymbol{w'} = 192392659230$, where \$ = 0. This matrix can be partitioned into five nonempty groups $G_{-1}, G_1, G_3, G_5, G_6$ as shown in the table. For each nonempty group G_j, let μ_j denote the maximum letter in the prefix $\boldsymbol{w'_0}$ before the first occurrence of $g = 9$ in every row of G_j. Then $\mu_{-1} = \boldsymbol{\varepsilon}, \mu_1 = 1, \mu_3 = 3, \mu_5 = 5$ and $\mu_6 = 6$. For $j > -1$, the μ_j are underlined in each row. The groups G_0, G_2, G_4, G_7, G_8 are empty and not shown in the figure. EVSA $= (2, 9, 5, 12, 1, 4, 11, 3, 10, 8, 7, 6)$, describing the ordering of the strings in $M^{\mathcal{V}}_{\boldsymbol{w'}}$, while $L = V\text{-BWT}_{\boldsymbol{w'}} = 153302299629$. Note that although $\boldsymbol{w'}$ contains no letter repetitions, in contrast $V\text{-BWT}_{\boldsymbol{w'}}$ does, as is typical with this transformation scheme.

Consider now the sequence $S = \boldsymbol{r_{h_1}}, \boldsymbol{r_{h_2}}, \cdots, \boldsymbol{r_i}, \cdots, \boldsymbol{r_j}, \cdots, \boldsymbol{r_{h_t}}$ of all rows in $M^{\mathcal{V}}_{\boldsymbol{x}}$ having suffix λ. Then the above argument holds for each adjacent pair in the sequence S and therefore the relative order of all rows in the sequence is maintained under the rotation of the letter λ.

Hence the $i-th$ occurrence of the suffix letter λ in the last column L is the same as the $i-th$ occurrence of the prefix letter λ in the first column F. An analogous argument shows the converse holds which establishes the claim. □

For example, consider rows 3 and 4 of $M^{\mathcal{V}}_{\boldsymbol{w'}}$ where $\boldsymbol{u} = 92659230192$, $\boldsymbol{v} = 01923926592$, and both end with $\lambda = 3$. Since row 3 is above row 4, so that $\boldsymbol{u}\lambda \prec \boldsymbol{v}\lambda$, therefore the rotations $\lambda\boldsymbol{u}$ and $\lambda\boldsymbol{v}$, which correspond respectively to rows 6 and 7, and which begin with $\lambda = 3$, satisfy $\lambda\boldsymbol{u} \prec \lambda\boldsymbol{v}$.

Lemma 25. *Let $\boldsymbol{x}$ (not containing \$) be a string with k maximum letters g and its associated V-BWT matrix be $M^{\mathcal{V}}_{\boldsymbol{x}}$. Then, the k letters g occur as the first k letters in the first column F of $M^{\mathcal{V}}_{\boldsymbol{x}}$.*

Proof. We apply Lemma 5 (C3). Observe that any rotation $\boldsymbol{x^r}$ of $\boldsymbol{x}$ with $\boldsymbol{x_0^r} = \boldsymbol{\varepsilon}$ in V-form must precede any rotation $\boldsymbol{x^{r'}}$ of $\boldsymbol{x}$ with $\boldsymbol{x_0^{r'}} \neq \boldsymbol{\varepsilon}$. Hence the k rotations of $\boldsymbol{x}$ starting g will occur first in column F of $M_{\boldsymbol{x}}^{\mathcal{V}}$. (Also see [10] for a statement of this fact.) □

Definition 26 (Groups)**.** *Consider the rows of $M_{\boldsymbol{x}}^{\mathcal{V}}$ in their V-forms (Definition 4), and examine the (possibly empty) $\boldsymbol{x_0}$ substring of each row. The rows of $M_{\boldsymbol{x}}^{\mathcal{V}}$ are partitioned into collections of adjacent rows called **groups**, where each group is characterized by a letter $\mu < g$ which is the maximum within every $\boldsymbol{x_0}$ substring in the group, and is denoted by G_μ. When $\mu = \boldsymbol{\varepsilon}$, we denote its corresponding group by G_{-1}.*

Lemma 27. *Let $\boldsymbol{x}$ be a string, and $M_{\boldsymbol{x}}^{\mathcal{V}}$ be its associated V-BWT matrix.*

(i) If the first letter of column F is not \$, then the first group in $M_{\boldsymbol{x}}^{\mathcal{V}}$ is G_{-1}; otherwise, it is G_0.

(ii) If the first letter of column F is \$, then the first two groups must be G_0, G_{-1} respectively, followed by groups in the increasing order of their characteristic letters μ.

Proof. The proof is a direct consequence of the fact that $\$ \prec \boldsymbol{\varepsilon}$, condition (C3) of Lemma 5 and Definition 26. □

Lemma 28. *Consider the group G_{-1} in $M_{\boldsymbol{x}}^{\mathcal{V}}$ of all rows beginning with $\boldsymbol{\varepsilon}g$. Let $|G_{-1}| = k$ and denote by $k' \leq k$ the number of distinct letters in column L of G_{-1} that do not equal g.*

(i) Then there must exist k' groups in $M_{\boldsymbol{x}}^{\mathcal{V}}$ corresponding to these distinct letters.

(ii) The k' distinct groups will occur in ascending order of the k' letters.

Proof.

(*i*) Let $g\boldsymbol{u_1}\lambda_1, g\boldsymbol{u_2}\lambda_2, \ldots, g\boldsymbol{u_{k'}}\lambda_{k'}$ be the rows in $M^{\mathcal{V}}_{\boldsymbol{x}}$ ending in the k' suffix letters $\lambda_1, \lambda_2, \ldots, \lambda_{k'}$. Since $M^{\mathcal{V}}_{\boldsymbol{x}}$ contains every rotation of $\boldsymbol{x}$, the rotations beginning with the k' suffix letters (which are now prefixes), that is, $\lambda_1 g\boldsymbol{u_1}, \lambda_2 g\boldsymbol{u_2}, \ldots, \lambda_{k'} g\boldsymbol{u_{k'}}$, must also exist in $M^{\mathcal{V}}_{\boldsymbol{x}}$. Considering these rows in V-form, the x_0 portion in each row is a single letter (possibly equal to \$). Hence, by Definition 26, groups $G_{\lambda_1}, \ldots, G_{\lambda_{k'}}$ exist, corresponding to rows $\lambda_i g\boldsymbol{u_1}, \ldots, \lambda_{k'} g\boldsymbol{u_{k'}}$, respectively. Therefore, k' groups in $M^{\mathcal{V}}_{\boldsymbol{x}}$ exist corresponding to the k' distinct suffix letters.

(*ii*) The proof of Part(*ii*) is a direct consequence of condition (C3) of Lemma 5 and Lemma 27. □

For example, in Figure 2, the suffix letters of the first $k' = k = 3$ rows are $1, 5, 3$ corresponding to groups G_1, G_3, and G_5 with $\mu_1 = 1,\ \mu_3 = 3$, and $\mu_5 = 5$.

Observation. It is possible for a letter λ in column F to occur (but not as the maximum letter) in the $\boldsymbol{x_0}$ portion of distinct groups in a V-BWT matrix. For example, in Figure 2, the letter 2 occurs in $\boldsymbol{x_0}$ for two groups G_3 and G_6.

For the following lemma, recall that the V-form of a string $\boldsymbol{x}$ is $V_k(\boldsymbol{x}) = \boldsymbol{x} = \boldsymbol{x_0} g \boldsymbol{x_1} g \cdots \boldsymbol{x_{k-1}} g \boldsymbol{x_k}$, where, if $\boldsymbol{x}$ is over the alphabet $\Sigma = \{1 < 2 < \cdots\}$ and the final letter of $\boldsymbol{x_k}$ is \$, then \$ is conveniently represented by 0.

Lemma 29. *Given a string* $\boldsymbol{x} = \boldsymbol{x_0} g \boldsymbol{x_1} g \cdots \boldsymbol{x_{k-1}} g \boldsymbol{x_k}$, *let* $\boldsymbol{x'} = \boldsymbol{x'_0} g \boldsymbol{x'_1} g \cdots \boldsymbol{x'_{k-1}} g$ *be the rotation of* $\boldsymbol{x}$ *such that* $\boldsymbol{x'_0} = \boldsymbol{x_k x_0}$ *and* $\boldsymbol{x'_i} = \boldsymbol{x_i}$ *for* $0 < i < k$. *Suppose* $\boldsymbol{z} = \boldsymbol{z}[1..j_i] = \boldsymbol{x'_i}$ *is nonempty for some* $0 \le i < k$, *and consider a letter* $\mu = \boldsymbol{z}[j]$ *in* $\boldsymbol{z}$, $1 \le j \le j_i$.

(i) Then the group G_μ *exists corresponding to the rotation of* $\boldsymbol{x}$ *starting with* μ, *that is, the row* $\boldsymbol{z}[j..j_i] g \boldsymbol{x'_{i+1}} g \cdots \boldsymbol{x'_{k-1}} g \boldsymbol{x'_0} g \boldsymbol{x'_1} g \cdots \boldsymbol{x'_{i-1}} g \boldsymbol{z}[1..j-1]$ *in* $M^{\mathcal{V}}_{\boldsymbol{x}}$, *if and only if* $\mu \ge \boldsymbol{z}[j']$ *for every* $j' \in j{+}1..j_i$.

(ii) Let μ *occur* t *number of times in* $\boldsymbol{x_i}$. *Let* $p_1, p_2, \ldots, p_t$ *be the lengths of the maximum prefix preceding each of the* t *occurrences of* μ *in* $\boldsymbol{x_i}$

respectively, such that every letter in the prefix is strictly less than μ. Then the total number of rows in G_μ corresponding to μ in $\boldsymbol{x_i}$ is equal to $p_1 + p_2 + \ldots + p_t + t$.

(iii) If the suffix letter of $\boldsymbol{x_k}$ is \$, then G_μ exists corresponding to $\mu = 0$ if and only if $\boldsymbol{x_0} = \boldsymbol{\varepsilon}$.

Proof. Part (i) $(\Rightarrow)$. The proof is by contradiction. Suppose, there exists a largest $j' \in j{+}1..j_i$ such that $\mu < \boldsymbol{z}[j']$ and $\boldsymbol{z}[j']$ is maximal in $j{+}1..j_i$. Let

$$\boldsymbol{u} = \boldsymbol{x'_{i+1}} g \cdots \boldsymbol{x'_{k-1}} g \boldsymbol{x'_0} g \boldsymbol{x'_1} g \cdots \boldsymbol{x'_{i-1}} g \boldsymbol{z}[1..j-1].$$

Then,

$$r_h = \boldsymbol{z}[j..j'..j_i] g \boldsymbol{u}, r_{h+1} = \boldsymbol{z}[j{+}1..j'..j_i] g \boldsymbol{u} \boldsymbol{z}[j], \ldots, r_{h+j'-j} = \boldsymbol{z}[j'..j_i] g \boldsymbol{u} \boldsymbol{z}[j..j'{-}1]$$

are the rows in the $M^{\mathcal{V}}_{\boldsymbol{x}}$ matrix corresponding to the rotations starting with letters $\boldsymbol{z}[j], \ldots, \boldsymbol{z}[j']$ respectively, and all these prefixes contain $\boldsymbol{z}[j']$. By Definition 26, Lemma 27, and the hypothesis $\mu < \boldsymbol{z}[j']$, the rows $r_h, r_{h+1}, \ldots, r_{h+j'-j}$ are in the vicinity of each other forming a group $G_{\boldsymbol{z}[j']}$ characterized by the maximum letter $\boldsymbol{z}[j']$. Clearly, $G_{\boldsymbol{z}[j']}$ contains r_h starting with $\mu = \boldsymbol{z}[j]$. Therefore, a group G_μ does not exist containing r_h, that is, the rotation of $\boldsymbol{x}$ starting with $\mu = \boldsymbol{z}[j]$, as this rotation belongs to $G_{\boldsymbol{z}[j']}$ – hence a contradiction is established. The converse can be shown analogously.

Part(ii): By Part (i), all t rotations starting with μ in $\boldsymbol{x'_i}$ occur in G_μ and therefore contribute to t rows. Further, any rotation of $\boldsymbol{x'}$ starting with a prefix preceding an occurrence (say the r-th occurrence) of μ in $\boldsymbol{x'_i}$ such that each letter in the prefix is strictly less than μ, also belongs to G_μ by Definition 26 and Lemma 5. Then the total number of such rows contributing to the size of G_μ by the r-th occurrence of μ in $\boldsymbol{x_i}$ is equal to the maximum length of the associated prefix, that is, p_r. Therefore, the total number of rows in G_μ corresponding to all t occurrences of μ in $\boldsymbol{x_i}$ is equal to $p_1 + p_2 + \ldots + p_t + t$.

Part(iii) $(\Rightarrow)$ Suppose G_0 exists in $M^{\mathcal{V}}_{\boldsymbol{x}}$. Then by Lemma 27, G_0 is the first

group in $M^{\mathcal{V}}_{\boldsymbol{x}}$ followed by G_{-1}. Then, there exists a rotation starting with $\$g$ in $M^{\mathcal{V}}_{\boldsymbol{x}}$. This is possible only when $\boldsymbol{x_0} = \boldsymbol{\varepsilon}$. Therefore ($\Rightarrow$) holds. The converse can be shown analogously. □

As an example, in Figure 2, consider $\boldsymbol{w'_2} = 265$, which contributes to groups with $\mu = 5$ and $\mu = 6$, but NOT $\mu = 2$. Further, as an example of case (*iii*) of the previous lemma, in the same example, $\boldsymbol{w'_0} = 1 \neq \boldsymbol{\varepsilon}$, and so the three rows starting with 9 appear at the start of $M^{\mathcal{V}}_{\boldsymbol{w'}}$, and the group G_0 does not exist. See Example 22 for an instance where $\boldsymbol{x_0} = \boldsymbol{\varepsilon}$.

We make use of Lemmas 24 – 29 to determine the number of rotations of the given string $\boldsymbol{x}$ that occur within each of the ordered groups $G_0, G_1, \ldots, G_{g-1}$. Using it, we compute $VC = VC[0..g-1]$, the V-order version of the C array defined originally for lexorder:

Definition 30. *(Array $VC[0, 1, \ldots, g-1]$) Suppose $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is defined on an integer alphabet $\Sigma = \{0, 1, \ldots, g\}$, where $\boldsymbol{x}[n] = 0$ corresponds to the sentinel letter \$. Then for each letter $\mu \in \Sigma$ that corresponds to a nonempty group G_μ, $0 \leq \mu < g$, $VC[\mu]$ is the sum of the sizes of the groups G_j, $0 \leq j \leq \mu-1$; that is,*

$$VC[\mu] = \sum_{j=0}^{\mu-1} |G_j|;$$

for all other letters μ, $VC[\mu] = 0$.

For instance, the VC array for the string $\boldsymbol{w'} = 192392659230$ is $VC = (0, 0, 0, 2, 0, 6, 7, 0, 0)$.

The computation of the VC array is given by Procedure COMPUTE_VC in Figure 3. Based on Lemmas 24 – 29 and Definitions 26 & 30, we state:

Lemma 31. *Procedure* COMPUTE_VC *correctly computes the VC array.*

Proof. The proof is a direct consequence of Lemma 29. □

Lemma 32. *Procedure* COMPUTE_VC *computes the VC array in $O(n)$ time.*

```
procedure COMPUTE_VC(x, n)
▷ Scan x to compute g and k
g ← −1
for i ← 1 to n do
    if x[i] > g then g ← x[i]; k ← 1
    elseif x[i] = g then k ← k+1
    i ← i+1
▷ Compute x' = rotated x (Lemma 29): x'[n] = g
h ← 0
while x[n−h] < g do h ← h+1
x' ← x[n−h+1..n]x[1..n−h]
▷ Compute G (Lemma 29)
G[0..g−1] ← 0^g
for i = n downto 1 do
    if x'[i] = g then μ ← −1
    else
        if μ < x'[i] then μ ← x'[i]
        G[μ] ← G[μ]+1
    i ← i−1
▷ Compute VC[0..g−1]
VC[0..g−1] ← 0^g; μ0 ← 0
while μ0 < g and G[μ0] = 0 do μ0 ← μ0+1
lastmu ← μ0
for μ = μ0+1 to g−1 do
    if G[μ] ≠ 0 then
        VC[μ] ← VC[lastmu]+G[lastmu]; lastmu ← μ
    μ ← μ+1
```

Figure 3: Computing the VC array.

Proof. The procedure COMPUTE_VC has **for** and **while** loops that are executed at most $O(n)$ time. Therefore, the running time of the procedure is $O(n)$. □

We now discuss the FM-index RANK function with respect to V-order; that is, the VRANK function defined as follows:

Definition 33 (VRANK Function)**.** *$VRANK_{\boldsymbol{x}}(I, \ell)$ gives the number of occurrences of letter ℓ in the substring of $\boldsymbol{x}$ defined by the interval $I = (i, j)$, where $1 \leq i \leq j \leq |\boldsymbol{x}|$.*

As can be observed, the only difference between RANK and VRANK is that RANK returns the rank of the letter ℓ in a certain prefix of $\boldsymbol{x}$, while VRANK

		L	Tally Table (T_L)						
			0	1	2	3	5	6	9
	1	1	0	1	0	0	0	0	0
	2	5	0	1	0	0	1	0	0
	3	3	0	1	0	1	1	0	0
	4	0	1	1	0	1	1	0	0
	5	3	1	1	0	2	1	0	0
$i =$	6	2	1	1	1	2	1	0	0
	7	2	1	1	2	2	1	0	0
	8	9	1	1	2	2	1	0	1
$j =$	9	9	1	1	2	2	1	0	2
	10	6	1	1	2	2	1	1	2
	11	2	1	1	3	2	1	1	2
	12	9	1	1	3	2	1	1	3

Figure 4: $\boldsymbol{w'} = 192392659230$, and $L = V\text{-BWT}_{\boldsymbol{w'}} = 153302299629$.

returns the rank of ℓ in the substring $\boldsymbol{x}[i..j]$ of $\boldsymbol{x}$. Then VRANK of a letter λ in the substring $\boldsymbol{x}[i..j]$ of $\boldsymbol{x}$ can be computed as follows:

$$\text{VRANK}_{\boldsymbol{x}}((i,j),\lambda) = \begin{cases} \text{VRANK}_{\boldsymbol{x}}((1,j),\lambda), & \text{if } i = 1 \\ \text{VRANK}_{\boldsymbol{x}}((1,j),\lambda) - \text{VRANK}_{\boldsymbol{x}}((1,i-1),\lambda), & \text{if } i > 1 \end{cases} \tag{6}$$

Analogous to RANK, VRANK can be computed easily using the Tally table for $L(T_L)$ — a matrix where each column represents a letter in the string and the i-th row represents the number of occurrences of the corresponding letter in the prefix of length i of L. Then a cell $T_L(i,j)$ in the Tally table stores the number of occurrences of letter j in the prefix of length i of L. Therefore, $T_L(i,j) = \text{VRANK}_L((1,i),j)$. For example, consider the Tally table given in Figure 4: by Equation (6),

$$\text{VRANK}_{\boldsymbol{w'}}((6,9),2) = \text{VRANK}_{\boldsymbol{w'}}((1,9),2) - \text{VRANK}_{\boldsymbol{w'}}((1,5),2) = 2 - 0 = 2.$$

The Tally table can be computed in linear time in the length of the string n, and the space required by it is $\mathcal{O}(\sigma n)$. As discussed in [19], this space can be

significantly reduced by storing only certain rows, starting at the first one, and occurring at a predetermined constant offset (α) in the Tally table. A row that is no longer stored in the Tally table is then computed in constant time by using existing rows and simple arithmetic calculations (for more details see [19]). The total size of the reduced Tally table is $\mathcal{O}(\frac{\sigma n}{\alpha})$.

Application of VC and VRANK to fully implement BWT-type pattern matching in V-order, analogous to the results in [12], remains an open problem.

6. V-Order Substring Matching

In this section we introduce V-order pattern matching using the backward search technique for substrings rather than letters. For clarity of concept, we exclude the use of the sentinel \$ in this section.

To achieve substring matching, the last column in the V-BWT matrix will be regarded as a column of substrings rather than letters, which we denote as VL. Specifically, VL consists of the last suffix $g\boldsymbol{x_k}$ of the V-form of each row in the V-BWT matrix. Likewise, we get the first column VF of prefix substrings in the V-BWT matrix, where the first k entries will be $\boldsymbol{\varepsilon} g$.

The substrings to be searched for (successfully) in a given string $\boldsymbol{x}$ take the form of a substring $\boldsymbol{X}$ of the $\boldsymbol{x_i}$ substrings in the V-form of $\boldsymbol{x}$; thus $\boldsymbol{X} = g\boldsymbol{x_i} \cdots g\boldsymbol{x_j}$ for $1 \leq i \leq j \leq k$. So the greatest letters g are basically for demarcation only. This demarcation ensures that string comparison is restricted to the $\boldsymbol{x_i}$ substrings and avoids the complex intermingling illustrated in Figure 5. Furthermore, if useful for particular required demarcation, such as length of substrings, an artificial greatest letter can be inserted in specified positions of the string $\boldsymbol{x}$.

We now show a property for substrings in V-order analogous to the lexorder BWT Last First mapping, in which the i-th occurrence of a letter in L is the same as the i-th occurrence of the letter in F (see Lemma 24).

Lemma 34. *The Last First Mapping holds for substrings in V-order.*

Proof. The proof follows immediately from the proof of Lemma 24 by replacing the letter λ with a substring and applying Lemma 8. □

In order to work with just the first k rows in the V-BWT matrix, we apply this lemma to modify the *FM-Index* [19] functions in the obvious way (see definitions of these functions for letters in Section 2):

- $C_{\boldsymbol{x}}(g\boldsymbol{\ell})$ gives, for each substring $g\boldsymbol{\ell} \in \Sigma^+$ that occurs in $\boldsymbol{x}$, the number of substrings $g\boldsymbol{x_i}$ in $\boldsymbol{x}$ smaller in V-order than $g\boldsymbol{\ell}$;
- $\text{RANK}_{\boldsymbol{x}}(p, g\boldsymbol{\ell})$ gives the number of occurrences of substring $g\boldsymbol{\ell}$ in the prefix of $\boldsymbol{x}$ of length p.

9 21 9 12 9 21 9 1314
9 21 9 1314 9 21 9 12
9 12 9 21 9 1314 9 21
9 1314 9 21 9 12 9 21
1 9 12 9 21 9 1314 9 2
1 9 1314 9 21 9 12 9 2
2 9 21 9 1314 9 21 9 1
21 9 12 9 21 9 1314 9
21 9 1314 9 21 9 12 9
12 9 21 9 1314 9 21 9
4 9 21 9 12 9 21 9 131
14 9 21 9 12 9 21 9 13
314 9 21 9 12 9 21 9 1
1314 9 21 9 12 9 21 9

Figure 5: V-BWT$_{\boldsymbol{x}}$, where the columns VF and VL are underlined.

Example 35 (Substring matching)**.** *For* $\boldsymbol{x} = 49219129219131$*, the V-BWT matrix $M^{\mathcal{V}}_{\boldsymbol{x}}$ is shown in Figure 5. Consider the substring* 921 *in the fourth row of the column VL. To calculate its index in VF we have $C_{\boldsymbol{x}}(921) = 0$ and $RANK_{VL}(4, 921) = 2$ (alternatively, $RANK_{VL}(3, 921)+1 = 1+1 = 2$ depending on the formulation of the definition). Thus, the Last First mapping technique maps* 921 *to index 0+2, namely the second position in VF.*

The substrings 921 in Example 35 are clustered into a run of length 2. Working with substrings potentially may allow for more effective preprocessing

for compression as the substrings can be encoded, such as $A = 921$, and we propose this avenue to be further examined.

Note that to compute the V-BWT matrix for substrings — that is, the first k rows — we encode each substring as above to generate a new alphabet and apply a linear sorting algorithm such as radix sort along with linear V-order string comparison to order the alphabet. Then we apply the techniques in [10] to compute the V-BWT matrix with respect to substrings in linear time overall. Generally this will result in space reduction for the columns VF and VL.

7. Future Research

As mentioned in Section 5, an immediate research problem is the application of VC and VRANK to BWT-type pattern matching in V-order, analogous to the lexorder results in [12].

Computing the V-BWT for substrings invites thorough experimental investigation in terms of the numbers of runs of similar substrings (rather than the usual letters) obtained. Furthermore, lex-extension order opens avenues for various techniques for substring clustering with appropriately defined BWTs: as discussed in [4], this hybrid ordering method can combine lexorder with any other total order on the factors induced in the factorization — here we combined lexorder with applying V-order to the factors, and propose considering other combinations of ordering methods.

The natural ordering of suffixes in V-order by their length suggests that further simple suffix computations and applications should be achievable. For instance, V-letters [4] are defined as V-words where the first letter in a string is strictly greater than the subsequent letters; for example, the V-word 812321. Then applying the trivial suffix-sorting to V-letters, the BWT V-transform is obtained simply by reversing the V-letter, and a V-letter can be similarly recovered from the V-transform [10]. We seek further efficient applications of these ideas. The computation of parallel V-ordering is also open.

Finally we propose the following problem. Suppose that $\boldsymbol{x}, \boldsymbol{y} \in \Sigma^+$ with

$\boldsymbol{x} \prec \boldsymbol{y}$. Under what permutations π — that is, $\boldsymbol{x} \to \pi(\boldsymbol{x})$ and $\boldsymbol{y} \to \pi(\boldsymbol{y})$ — does $\pi(\boldsymbol{x}) \prec \pi(\boldsymbol{y})$ hold? For instance, for $21 \prec 12$ no permutation works, whereas interchanging the first and last letters does work for $142 \prec 243$, since $241 \prec 342$. This example generalizes to requiring that the rightmost substrings of the V-forms are in V-order.

Acknowledgements

We thank the reviewers for carefully reading the manuscript and providing insightful comments which led to substantial improvements.

Funding: The first author was part-funded by the European Regional Development Fund through the Welsh Government [Grant Number 80761-AU-137 (West)]:

The second and third authors were funded by the Natural Sciences & Engineering Research Council of Canada [Grant Number 10536797].

References

[1] A. Alatabbi, J. W. Daykin, N. Mhaskar, M. S. Rahman, W. F. Smyth, Applications of V-order: Suffix arrays, the Burrows-Wheeler transform & the FM-index, in: G. K. Das, P. S. Mandal, K. Mukhopadhyaya, S. ichi Nakano (Eds.), International Workshop on Algorithms & Computation (WALCOM), Vol. 11355 of Lecture Notes in Computer Science, Springer, 2019, pp. 329–338.

[2] T.-N. Danh, D. E. Daykin, The structure of V-Order for integer vectors, Congr. Numer. Ed. A.J.W. Hilton. Utilas Mat. Pub. Inc., Winnipeg, Canada, 113 (1996) (1996) 43–53.

[3] D. E. Daykin, J. W. Daykin, W. F. Smyth, String comparison and Lyndon–like factorization using V-Order in linear time, in: Symp. on Combinatorial Pattern Matching, Vol. 6661, 2011, pp. 65–76.

[4] 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.

[5] A. Alatabbi, J. W. Daykin, M. S. Rahman, W. F. Smyth, Simple linear comparison of strings in V-Order – (extended abstract), in: International Workshop on Algorithms & Computation (WALCOM), Vol. 8344 of Lecture Notes in Computer Science, Springer, 2014, pp. 80–89.

[6] A. Alatabbi, J. W. Daykin, M. S. Rahman, W. F. Smyth, String comparison in V-Order: New lexicographic properties & on-line applications, arXiv:1507.07038 (2015).

[7] A. Alatabbi, J. W. Daykin, M. S. Rahman, W. F. Smyth, Simple linear comparison of strings in V-Order, Fundam. Inform. 139 (2) (2015) 115–126. doi:10.3233/FI-2015-1228.
URL https://doi.org/10.3233/FI-2015-1228

[8] A. Alatabbi, J. W. Daykin, J. Kärkkäinen, M. S. Rahman, W. F. Smyth, V-Order: New combinatorial properties & a simple comparison algorithm, Discrete Appl. Math. 215 (2016) 41–46.

[9] A. Alatabbi, J. W. Daykin, N. Mhaskar, M. S. Rahman, W. F. Smyth, A faster V-Order string comparison algorithm, in: Proc. Prague Stringology Conference, 2018, pp. 38–49.

[10] J. W. Daykin, W. F. Smyth, A bijective variant of the Burrows– Wheeler transform using V-Order, Theoret. Comput. Sci. 531 (2014) 77–89.

[11] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, in: R. A. Baeza-Yates, E. Chávez, M. Crochemore (Eds.), Symp. on Combina-

torial Pattern Matching, Vol. 2676 of Lecture Notes in Computer Science, Springer, 2003, pp. 200–210.

[12] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, Suffix array and Lyndon factorization of a text, J. Discrete Algorithms 28 (2014) 2–8. doi:10.1016/j.jda.2014.06.001.
URL http://dx.doi.org/10.1016/j.jda.2014.06.001

[13] U. Baier, Linear-time suffix sorting — a new approach for suffix array construction, in: Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), 2016, pp. 23:1–23:12.

[14] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, K. Tsuruta, The "runs" theorem, SIAM J. Comput. 46 (5) (2017) 1501–1514.

[15] J. W. Daykin, F. Franek, J. Holub, A. S. M. Sohidull Islam, W. F. Smyth, Reconstructing a string from its Lyndon arrays, Theor. Comput. Sci. 710 (2018) 44–51.

[16] F. Franek, A. Paracha, W. F. Smyth, The linear equivalence of the suffix array and the parially sorted Lyndon array, in: Proc. Prague Stringology Conf., 2017, pp. 85–92.

[17] F. Franek, A. S. M. Sohidull Islam, M. S. Rahman, W. F. Smyth, Algorithms to compute the Lyndon array, in: Proc. Prague Stringology Conf., 2016, pp. 172–184.

[18] D. Adjeroh, T. Bell, A. Mukherjee, The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching, Springer Publishing Company, 2008.

[19] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. 41st Annual Symposium on Foundations of Computer Science, (FOCS 2000), 2000, pp. 390–398.

[20] K. T. Chen, R. H. Fox, R. C. Lyndon, Free differential calculus, iv – the quotient groups of the lower central series, Ann. Math. (1958) 68:81–95.

[21] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, New York, NY, USA, 2007.

[22] B. Smyth, Computing Patterns in Strings, Pearson/Addison–Wesley, 2003.

[23] D. E. Daykin, J. W. Daykin, Lyndon–like and V-Order factorizations of strings, J. Discrete Algorithms 1 (3–4) (2003) 357–365.

[24] D. E. Daykin, J. W. Daykin, W. F. Smyth, Combinatorics of unique maximal factorization families (UMFFs), Fund. Inform. 97–3, Special Issue on Stringology, R. Janicki, S. J. Puglisi and M. S. Rahman, editors (2009) 295–309.