String Comparison in V-Order: New Lexicographic Properties & On-line Applications

Ali Alatabbi[1], Jacqueline W. Daykin[1,2], M. Sohel Rahman[3], and W. F. Smyth[1,4,5⋆]

[1] Department of Informatics, King's College London, UK
ali.alatabbi@kcl.ac.uk, jackie.daykin@kcl.ac.uk
[2] Department of Computer Science
Royal Holloway College, University of London, UK
J.Daykin@cs.rhul.ac.uk
[3] AℓEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh
msrahman@cse.buet.ac.bd
[4] Algorithms Research Group, Department of Computing & Software
McMaster University, Canada
smyth@mcmaster.ca
[5] School of Engineering & Information Technology
Murdoch University, Western Australia

Abstract. V-order is a global order on strings related to Unique Maximal Factorization Families (UMFFs) [6,7], which are themselves generalizations of Lyndon words [14]. V-order has recently been proposed as an alternative to lexicographical order in the computation of suffix arrays and in the suffix-sorting induced by the Burrows-Wheeler transform. Efficient V-ordering of strings thus becomes a matter of considerable interest. In this paper we present new and surprising results on V-order in strings, then go on to explore the algorithmic consequences.

1 Introduction

This paper extends current knowledge on the non-lexicographic string ordering technique known as V-order [5]. New combinatorial insights are obtained which are linked to computational settings. In particular, we relate V-order string comparison to lexicographic by showing how it is

⋆ This work was supported in part by the Natural Sciences & Engineering Research Council of Canada.

possible to traverse the strings from left to right, respectively right to left, at each stage determining in $O(1)$ time the order of prefixes, respectively suffixes. This improves on existing ordering algorithms [1, 2, 7] in various ways: it removes any dependence on an "indexed" alphabet, it orders prefixes and suffixes in addition to the original strings, and it reduces dependence on additional data structures. Furthermore, we introduce an input-sensitive variant for V-order comparison.

Regarding practical applications of V-order, in [9] a novel variant of the classic lexicographic Burrows-Wheeler transform, the V-transform (V-BWT), was introduced which was based on V-order – instances of enhanced data clustering were demonstrated. Linear V-sorting of all the rotations of a string $\boldsymbol{x} = \boldsymbol{x}[1 \dots n]$, as required for an efficient transform, was achieved by linear time and space V-order string comparison (Daykin et al. 2011) [7] along with $\Theta(n)$ suffix-sorting (Ko and Aluru, 2003) [13]. Lyndon-like factorization of a string into V-words is likewise linear in time and space [7]. For V-words, [9] showed how to compute the V-transform in $\Theta(n)$ time and space; in addition, inverting the V-transform to recover the input V-word was achieved in time $O(n^2 \log k')$, using $O(n+k')$ additional storage, where k' is the number of sequences of largest letters in $\boldsymbol{x}$. A bijective algorithm was also outlined in the case that $\boldsymbol{x}$ is arbitrary.

We apply the new combinatorial insights gained to modify ideas given in [15] for Lyndon factorizations, suffix arrays and the Burrows Wheeler transform, to similarly obtain on-line processing for V-order.

2 Preliminaries

Consider a finite totally ordered alphabet Σ which consists of a set of characters (equivalently letters or symbols) with cardinality $|\Sigma|$. A string is a sequence of zero or more characters over Σ. A string $\boldsymbol{s}$ of length $|\boldsymbol{s}| = n$ is represented by $\boldsymbol{s}[1 \dots n]$, where $\boldsymbol{s}[i] \in \Sigma$ for $1 \leq i \leq n$. The set of all non-empty strings over the alphabet Σ is denoted by Σ^+. The empty string with zero length is denoted by ϵ, with $\Sigma^* = \Sigma^+ \cup \epsilon$; A string $\boldsymbol{w}$ is a substring, or factor, of $\boldsymbol{s}$ if $\boldsymbol{s} = \boldsymbol{uwv}$, where $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^*$.

Words $\boldsymbol{w}[1\ldots i]$ are prefixes of $\boldsymbol{w}$, and words $\boldsymbol{w}[i\ldots n]$ are suffixes of $\boldsymbol{w}$. For further stringological definitions, theory and algorithmics see [4].

Some of our applications are derived from Lyndon words, which we now introduce. A string $\boldsymbol{y} = \boldsymbol{y}[1\ldots n]$ is a *conjugate* (or cyclic rotation) of $\boldsymbol{x} = \boldsymbol{x}[1\ldots n]$ if $\boldsymbol{y}[1\ldots n] = \boldsymbol{x}[i\ldots n]\boldsymbol{x}[1\ldots i-1]$ for some $1 \leq i \leq n$ (for $i = 1,\ \boldsymbol{y} = \boldsymbol{x}$). A *Lyndon word* is a primitive word which is minimal for the lexicographical order (lexorder) of its conjugacy class.

Theorem 1. *[3] Any word $\boldsymbol{w}$ can be written uniquely as a non-increasing product $\boldsymbol{w} = \boldsymbol{u}_1\boldsymbol{u}_2\cdots\boldsymbol{u}_k$ of Lyndon words.*

Theorem 1 shows that there is a unique decomposition of any word into non-increasing Lyndon words ($\boldsymbol{u}_1 \geq \boldsymbol{u}_2 \geq \cdots \geq \boldsymbol{u}_k$). We proceed to define a non-lexicographic order, V-order, and then establish useful new lexicographic characteristics for V-order.

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 the letter x_h. Write $\boldsymbol{x}^{s*}$ for $(...(\boldsymbol{x}^*)^*...)^*$ with $s \geq 0$ stars. Let $g = \max\{x_1, x_2, ..., x_n\}$, and let k be the number of occurrences of g in $\boldsymbol{x}$. Then the sequence $\boldsymbol{x}, \boldsymbol{x}^*, \boldsymbol{x}^{2*}, ...$ ends with $g^k, ..., g^2, g^1, g^0 = \boldsymbol{\varepsilon}$. In the *star tree* each string $\boldsymbol{x}$ over Σ labels a vertex, and there is a directed edge from $\boldsymbol{x}$ to $\boldsymbol{x}^*$, with the empty string $\boldsymbol{\varepsilon}$ as the root.

Definition 1. *We define V-*order $\prec$ *between distinct strings $\boldsymbol{x}$,$\boldsymbol{y}$ with $\boldsymbol{x} \prec \boldsymbol{y}$. First $\boldsymbol{x} \prec \boldsymbol{y}$ if $\boldsymbol{x}$ is in the path $\boldsymbol{y}, \boldsymbol{y}^*, \boldsymbol{y}^{2*}, ..., \boldsymbol{\varepsilon}$. If $\boldsymbol{x}, \boldsymbol{y}$ are not in a path, there exist smallest s, t such that $\boldsymbol{x}^{(s+1)*} = \boldsymbol{y}^{(t+1)*}$. Put $\boldsymbol{s} = \boldsymbol{x}^{s*}$ 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 $\boldsymbol{x} \prec \boldsymbol{y}$. Clearly $\prec$ is a total order.*

For instance, using the natural ordering of integers, if $\boldsymbol{x} = 32415$, then $\boldsymbol{x}^* = 3245$, $\boldsymbol{x}^{2*} = 345$, $\boldsymbol{x}^{3*} = 45$ and so $45 \prec 32415$.

Definition 2. *[5–8] 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. For clarity, when more than one string is involved, we use the notation $g = \mathcal{L}_{\boldsymbol{x}}$, $k = \mathcal{C}_{\boldsymbol{x}}$.

Lemma 1. *[5–8] Suppose we are given distinct strings $\boldsymbol{x}$ and $\boldsymbol{y}$ with corresponding V-forms as follows:*

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

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

Let $h \in \{0 \ldots \max(j, k)\}$ 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* $\boldsymbol{x}_h \prec \boldsymbol{y}_h$.

Lemma 2. *[6, 7] For given strings $\boldsymbol{v}$ and $\boldsymbol{x}$, if $\boldsymbol{v}$ is a proper subsequence of $\boldsymbol{x}$, then $\boldsymbol{v} \prec \boldsymbol{x}$.*

Example 1. We compare two dictionaries for a set of English words over the ordered Roman alphabet.
Lexorder($<$) dictionary: *catastrophe* $<$ *sop* $<$ *strop* $<$ *strophe* $<$ *top*. The well-known lexorder positional technique seeks the first difference from the left and then applies the ordering of the alphabet.
V-order ($\prec$) dictionary: *sop* $\prec$ *top* $\prec$ *strop* $\prec$ *strophe* $\prec$ *catastrophe*. The first V-order comparison is determined by Lemma 1(C1) and the following three by the useful Lemma 2.

3 New Results on V-Order

A main interest of this paper is to consider positional lexorder-type ordering techniques for V-order, for which we first establish some basics. Given an ordered alphabet $\Sigma = \{1 < 2 < \cdots\}$ and a string $\boldsymbol{x} \in \Sigma^+$ with $|\boldsymbol{x}| > 1$, then from conditions (C1, C2) we have, as for lexorder, $1 \prec \boldsymbol{x} \prec \boldsymbol{x^i}$ for all $i > 1$. For strings $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \Sigma^+$ with $\boldsymbol{u} \prec \boldsymbol{v} \prec \boldsymbol{w}$, we find by Lemma 2 that, again as for lexorder, both $\boldsymbol{u} \prec \boldsymbol{uv}$ and $\boldsymbol{vw} \not\prec \boldsymbol{w}$ (in contrast to Lyndon words). In general, for $i, j > 1$, we can say that

$$1 \prec \boldsymbol{u} \prec \boldsymbol{u^2} \prec \cdots \prec \boldsymbol{u^i} \prec \boldsymbol{u^i v} \prec \cdots \prec \boldsymbol{u^i v^j} \prec \cdots \prec \boldsymbol{u^i v^j w} \prec \cdots$$

We begin by generalizing Lemma 2.5 in [9]:

Lemma 3. *For any two strings $\boldsymbol{x}$, $\boldsymbol{y}$ and $\lambda \in \Sigma$, $\boldsymbol{x} \prec \boldsymbol{y} \Leftrightarrow \boldsymbol{x}\lambda \prec \boldsymbol{y}\lambda$.*

Proof. Let $\boldsymbol{x'} = \boldsymbol{x}\lambda$, $\boldsymbol{y'} = \boldsymbol{y}\lambda$. First observe that if $\mathcal{L}_{\boldsymbol{x}} < \mathcal{L}_{\boldsymbol{y}}$, then by (C1), $\boldsymbol{x} \prec \boldsymbol{y}$. Furthermore:

- if $\lambda < \mathcal{L}_{\boldsymbol{y}}$, then $\boldsymbol{x'} \prec \boldsymbol{y'}$ by (C1), because $\mathcal{L}_{\boldsymbol{x}} \leq \mathcal{L}_{\boldsymbol{x'}} < \mathcal{L}_{\boldsymbol{y}} = \mathcal{L}_{\boldsymbol{y'}}$;
- if $\lambda = \mathcal{L}_{\boldsymbol{y}}$, then $\boldsymbol{x'} \prec \boldsymbol{y'}$ by (C2), because $\mathcal{L}_{\boldsymbol{x'}} = \mathcal{L}_{\boldsymbol{y'}} = \lambda$ and $\mathcal{C}_{\boldsymbol{x'}} = 1 < \mathcal{C}_{\boldsymbol{y'}}$;
- if $\lambda > \mathcal{L}_{\boldsymbol{y}}$, then $\boldsymbol{x'} \prec \boldsymbol{y'}$ by (C3), because $\mathcal{L}_{\boldsymbol{x'}} = \mathcal{L}_{\boldsymbol{y'}} = \lambda$, $\mathcal{C}_{\boldsymbol{x'}} = \mathcal{C}_{\boldsymbol{y'}} = 1$, and $\boldsymbol{x} \prec \boldsymbol{y}$.

Thus the lemma holds for $\mathcal{L}_{\boldsymbol{x}} < \mathcal{L}_{\boldsymbol{y}}$ and, by the complementary argument, it holds also for $\mathcal{L}_{\boldsymbol{y}} < \mathcal{L}_{\boldsymbol{x}}$. We may assume therefore that $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$.

Suppose then that $\mathcal{C}_{\boldsymbol{x}} < \mathcal{C}_{\boldsymbol{y}}$, so that by (C2), $\boldsymbol{x} \prec \boldsymbol{y}$. Furthermore:

- if $\lambda \leq \mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, then $\boldsymbol{x'} \prec \boldsymbol{y'}$ by (C2), because $\mathcal{C}_{\boldsymbol{x'}} = \mathcal{C}_{\boldsymbol{x}} + \delta < \mathcal{C}_{\boldsymbol{y}} + \delta = \mathcal{C}_{\boldsymbol{y'}}$, where $\delta = 0$ ($\lambda < \mathcal{L}_{\boldsymbol{x}}$) or 1 ($\lambda = \mathcal{L}_{\boldsymbol{x}}$);
- if $\lambda > \mathcal{L}_{\boldsymbol{x}}$, then $\boldsymbol{x'} \prec \boldsymbol{y'}$ by (C3), because $\mathcal{L}_{\boldsymbol{x'}} = \mathcal{L}_{\boldsymbol{y'}} = \lambda$, $\mathcal{C}_{\boldsymbol{x'}} = \mathcal{C}_{\boldsymbol{y'}} = 1$, and $\boldsymbol{x} \prec \boldsymbol{y}$.

Thus the lemma holds for $\mathcal{C}_{\boldsymbol{x}} < \mathcal{C}_{\boldsymbol{y}}$, and as above also for $\mathcal{C}_{\boldsymbol{y}} < \mathcal{C}_{\boldsymbol{x}}$.

Suppose therefore that $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, $\mathcal{C}_{\boldsymbol{x}} = \mathcal{C}_{\boldsymbol{y}}$. Then whether or not $\boldsymbol{x} \prec \boldsymbol{y}$ depends on the least value h of Lemma 1 such that $\boldsymbol{x}_h \prec \boldsymbol{y}_h$ or $\boldsymbol{y}_h \prec \boldsymbol{x}_h$:

- If $\lambda = \mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, then h is unchanged by appending λ to $\boldsymbol{x}$ and to $\boldsymbol{y}$, so that, in this case, $\boldsymbol{x} \prec \boldsymbol{y} \Leftrightarrow \boldsymbol{x'} \prec \boldsymbol{y'}$, as required.
- For $\lambda > \mathcal{L}_{\boldsymbol{x}}$, we find as above that $\mathcal{L}_{\boldsymbol{x'}} = \mathcal{L}_{\boldsymbol{y'}} = \lambda$, $\mathcal{C}_{\boldsymbol{x'}} = \mathcal{C}_{\boldsymbol{y'}} = 1$, the ordering of $\boldsymbol{x'}$ and $\boldsymbol{y'}$ is equivalent to the ordering of $\boldsymbol{x}$ and $\boldsymbol{y}$.
- Finally, suppose that $\lambda < \mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$. If $h < \mathcal{C}_{\boldsymbol{x}}$, then as above the ordering of $\boldsymbol{x'}$,$\boldsymbol{y'}$ corresponds to the ordering of $\boldsymbol{x}$,$\boldsymbol{y}$, unaffected by appending λ. If on the other hand $h = \mathcal{C}_{\boldsymbol{x}}$, then the problem reduces recursively to ordering $\boldsymbol{x_h}\lambda, \boldsymbol{y_h}\lambda$ based on the ordering of $\boldsymbol{x_h}$,$\boldsymbol{y_h}$, where $\mathcal{L}_{\boldsymbol{x_h}} < \mathcal{L}_{\boldsymbol{x}}$ and $\mathcal{L}_{\boldsymbol{y_h}} < \mathcal{L}_{\boldsymbol{y}}$. Thus, after a finite number of such reductions, one of the above cases must hold.

  This completes the proof. □

Lemma 4. *For any two strings* $\boldsymbol{x}$, $\boldsymbol{y}$ *and* $\lambda \in \Sigma$, $\boldsymbol{x} \prec \boldsymbol{y} \Leftrightarrow \lambda\boldsymbol{x} \prec \lambda\boldsymbol{y}$.

Proof. The argument is analogous to that given for Lemma 3. Note that the recursive case $\lambda\boldsymbol{x_0}, \lambda\boldsymbol{y_0}$ is likewise based on the ordering of $\boldsymbol{x_0}, \boldsymbol{y_0}$, where $\mathcal{L}_{\boldsymbol{x_0}} < \mathcal{L}_{\boldsymbol{x}}$ and $\mathcal{L}_{\boldsymbol{y_0}} < \mathcal{L}_{\boldsymbol{y}}$. □

Interestingly, although Lemma 4 holds for lexorder, Lemma 3 does not as shown by: $a < ab$ in lexorder but $ac \not\prec abc$.

We can now combine the above lemmas into a more general result:

Theorem 2. *For any strings* $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{x} \prec \boldsymbol{y} \Leftrightarrow \boldsymbol{uxv} \prec \boldsymbol{uyv}$.

Proof. This follows from repeated applications of Lemmas 3 & 4, where we append one letter at a time to suffixes and prepend one letter at a time to prefixes. □

We can establish extensions and applications of these results:

Lemma 5. *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be strings with V-forms*

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

For any letter $\lambda \leq \max(\mathcal{L}_{\boldsymbol{x}}, \mathcal{L}_{\boldsymbol{y}})$ and any integer $i \in \{0 \ldots \max(j,k)\}$, let

$$\boldsymbol{x'} = \boldsymbol{x}_0\mathcal{L}_{\boldsymbol{x}}\cdots\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_i\lambda\mathcal{L}_{\boldsymbol{x}}\cdots\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_j,$$
$$\boldsymbol{y'} = \boldsymbol{y}_0\mathcal{L}_{\boldsymbol{y}}\cdots\mathcal{L}_{\boldsymbol{y}}\boldsymbol{y}_i\lambda\mathcal{L}_{\boldsymbol{y}}\cdots\mathcal{L}_{\boldsymbol{y}}\boldsymbol{y}_k,$$
$$\boldsymbol{x''} = \boldsymbol{x}_0\mathcal{L}_{\boldsymbol{x}}\cdots\mathcal{L}_{\boldsymbol{x}}\lambda\boldsymbol{x}_i\mathcal{L}_{\boldsymbol{x}}\cdots\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_j,$$
$$\boldsymbol{y''} = \boldsymbol{y}_0\mathcal{L}_{\boldsymbol{y}}\cdots\mathcal{L}_{\boldsymbol{y}}\lambda\boldsymbol{y}_i\mathcal{L}_{\boldsymbol{y}}\cdots\mathcal{L}_{\boldsymbol{y}}\boldsymbol{y}_k.$$

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

Proof. First suppose that $\boldsymbol{x'} \prec \boldsymbol{y'}$, so that one of the conditions (C1)-(C3) of Lemma 1 must hold:

- Assume that $\mathcal{L}_{\boldsymbol{x'}} < \mathcal{L}_{\boldsymbol{y'}}$. Then $\lambda < \mathcal{L}_{\boldsymbol{y}}$ and $\mathcal{L}_{\boldsymbol{x}} \leq \mathcal{L}_{\boldsymbol{x'}} < \mathcal{L}_{\boldsymbol{y'}} = \mathcal{L}_{\boldsymbol{y}}$, so that $\boldsymbol{x} \prec \boldsymbol{y}$ by (C1).
- Assume that $\mathcal{L}_{\boldsymbol{x'}} = \mathcal{L}_{\boldsymbol{y'}}$, with $\mathcal{C}_{\boldsymbol{x'}} < \mathcal{C}_{\boldsymbol{y'}}$. If $\lambda = \mathcal{L}_{\boldsymbol{y}}$, then either $\mathcal{L}_{\boldsymbol{x}} < \mathcal{L}_{\boldsymbol{y}}$ or $\lambda = \mathcal{L}_{\boldsymbol{x}}$ and $\mathcal{C}_{\boldsymbol{x}} = \mathcal{C}_{\boldsymbol{x'}} - 1 < \mathcal{C}_{\boldsymbol{y'}} - 1 = \mathcal{C}_{\boldsymbol{y}}$; otherwise, $\lambda < \mathcal{L}_{\boldsymbol{y}}$, so that $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$ with $\mathcal{C}_{\boldsymbol{x}} = \mathcal{C}_{\boldsymbol{x'}} < \mathcal{C}_{\boldsymbol{y'}} = \mathcal{C}_{\boldsymbol{y}}$. In all three cases, $\boldsymbol{x} \prec \boldsymbol{y}$ by (C2).
- If $\mathcal{L}_{\boldsymbol{x'}} = \mathcal{L}_{\boldsymbol{y'}}$ and $\mathcal{C}_{\boldsymbol{x'}} = \mathcal{C}_{\boldsymbol{y'}}$, then whether or not $\boldsymbol{x} \prec \boldsymbol{y}$ depends on the least value h of Lemma 1 such that $\boldsymbol{x}_h \prec \boldsymbol{y}_h$:
  - if $h \neq i$, then the ordering of $\boldsymbol{x}, \boldsymbol{y}$ corresponds to the ordering of $\boldsymbol{x'}, \boldsymbol{y'}$, unaffected by removing λ;
  - if $h = i$, then the ordering of $\boldsymbol{x}, \boldsymbol{y}$ reduces to the ordering of $\boldsymbol{x_h}\lambda, \boldsymbol{y_h}\lambda$, so that $\boldsymbol{x} \prec \boldsymbol{y}$ by Theorem 1.

Next suppose that $\boldsymbol{x} \prec \boldsymbol{y}$. Again we consider the conditions (C1)-(C3) of Lemma 1:

- Assume that $\mathcal{L}_{\boldsymbol{x}} < \mathcal{L}_{\boldsymbol{y}}$. If $\lambda = \mathcal{L}_{\boldsymbol{y}}$, then $\lambda = \mathcal{L}_{\boldsymbol{x'}} = \mathcal{L}_{\boldsymbol{y'}}$ with $\mathcal{C}_{\boldsymbol{x'}} = 1 < \mathcal{C}_{\boldsymbol{y'}}$, so that $\boldsymbol{x'} \prec \boldsymbol{y'}$ by (C2); while if $\lambda < \mathcal{L}_{\boldsymbol{y}}$, then $\boldsymbol{x'} \prec \boldsymbol{y'}$ by (C1), because $\mathcal{L}_{\boldsymbol{x}} \leq \mathcal{L}_{\boldsymbol{x'}} < \mathcal{L}_{\boldsymbol{y}} = \mathcal{L}_{\boldsymbol{y'}}$.
- Assume that $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, with $\mathcal{C}_{\boldsymbol{x}} < \mathcal{C}_{\boldsymbol{y}}$. If $\lambda = \mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, then $\mathcal{C}_{\boldsymbol{x'}} = \mathcal{C}_{\boldsymbol{x}}+1 < \mathcal{C}_{\boldsymbol{y}}+1 = \mathcal{C}_{\boldsymbol{y'}}$; if $\lambda < \mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, then $\mathcal{C}_{\boldsymbol{x'}} = \mathcal{C}_{\boldsymbol{x}} < \mathcal{C}_{\boldsymbol{y}} = \mathcal{C}_{\boldsymbol{y'}}$. In both cases, $\boldsymbol{x'} \prec \boldsymbol{y'}$ by (C2).
- If $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$ and $\mathcal{C}_{\boldsymbol{x}} = \mathcal{C}_{\boldsymbol{y}}$, then again whether or not $\boldsymbol{x'} \prec \boldsymbol{y'}$ depends on the least value h of Lemma 1 such that $\boldsymbol{x}_h \prec \boldsymbol{y}_h$:
  - if $h \neq i$, then the ordering of $\boldsymbol{x'}, \boldsymbol{y'}$ corresponds to the ordering of $\boldsymbol{x}, \boldsymbol{y}$, unaffected by adding λ;
  - if $h = i$, then the ordering of $\boldsymbol{x'}, \boldsymbol{y'}$ reduces to the ordering of $\boldsymbol{x_h}\lambda, \boldsymbol{y_h}\lambda$, so that $\boldsymbol{x'} \prec \boldsymbol{y'}$ by Theorem 2.

This completes the proof that $\boldsymbol{x'} \prec \boldsymbol{y'} \Leftrightarrow \boldsymbol{x} \prec \boldsymbol{y}$. The proof that $\boldsymbol{x''} \prec \boldsymbol{y''} \Leftrightarrow \boldsymbol{x} \prec \boldsymbol{y}$ is similar. □

To see that Lemma 5 does not hold for $\lambda > \max(\mathcal{L}_{\boldsymbol{x}}, \mathcal{L}_{\boldsymbol{y}})$, consider

$$\boldsymbol{x} = 1323 \prec \boldsymbol{y} = 3133, \ \lambda = 4, \ \text{but } \boldsymbol{y'} = 43133 \prec \boldsymbol{x'} = 14323.$$

Remark 1. Lemma 5 is easily generalized by replacing λ by any string $\boldsymbol{u} = u_1 u_2 \cdots u_m$ such that, for $1 \leq j \leq m$, $u_m \leq \max(\mathcal{L}_{\boldsymbol{x}}, \mathcal{L}_{\boldsymbol{y}})$, and inserting such a $\boldsymbol{u}$ at any or all positions $i \in \{0 \ldots \max(j, k)\}$.

Lemma 6. *For any two strings $\boldsymbol{x}, \boldsymbol{y}$ and letters $\lambda, \mu \in \Sigma$, $\lambda \leq \mu$:*

(i) $\boldsymbol{x} \prec \boldsymbol{y} \Rightarrow \lambda\boldsymbol{x} \prec \mu\boldsymbol{y}$;
(ii) $\boldsymbol{x} \prec \boldsymbol{y} \Rightarrow \boldsymbol{x}\lambda \prec \boldsymbol{y}\mu$.

Proof. For $\lambda = \mu$, (i) reduces to Lemma 4, while (ii) reduces to Lemma 3. Thus we may assume $\lambda < \mu$.

Suppose $\boldsymbol{x} \prec \boldsymbol{y}$. Then by Lemma 4 $\lambda\boldsymbol{x} \prec \lambda\boldsymbol{y}$, while by Theorem 2 with $\boldsymbol{u} = \boldsymbol{\varepsilon}$, $\lambda\boldsymbol{y} \prec \mu\boldsymbol{y}$. Therefore $\lambda\boldsymbol{x} \prec \mu\boldsymbol{y}$, proving (i). The proof of (ii)

is similar. □

The following examples show that sufficiency does not hold in Lemma 6:

(i) $\boldsymbol{y} = 441 \prec \boldsymbol{x} = 442$, $\lambda = 2 < \mu = 3$, but $\lambda\boldsymbol{x} = 2442 \prec \mu\boldsymbol{y} = 3441$;
(ii) $\boldsymbol{y} = 441 \prec \boldsymbol{x} = 442$, $\lambda = 2 < \mu = 3$, but $\boldsymbol{x}\lambda = 4422 \prec \boldsymbol{y}\mu = 4413$.

4 Applications

Some of the results presented above lead us to some interesting applications. In this section, we first present a brief sketch of an idea for a new string comparison algorithm in V-order and then proceed to consider applications of our results to suffix arrays (SAs) and the Burrows Wheeler transform (BWT).

4.1 V-Order String Comparison

Recently, Alatabbi et al. presented an interesting V-order string comparison algorithm in [1, 2] (referred to as the ADRS algorithm henceforth), where a mapping of the position of each letter in the string is exploited to check for the conditions stated in Lemma 1. Note that there are three conditions in Lemma 1 and things get most interesting when we reach Condition (C3) because of its recursive nature. Now, the efficiency of ADRS algorithm depends on a key result (cf. Corollary 2.9 of [2]) which proves that the mismatch position of the two strings under comparison remains the same as we go deep into the recursion. This fact along with the result presented in Lemma 5 gives us yet another idea for an efficient string comparison algorithm in V-order. Essentially, the idea builds upon the idea of the map in the ADRS algorithm as we will now outline.

Suppose we are given two strings, $\boldsymbol{x}$ and $\boldsymbol{y}$, with V-forms

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

Step 1: We first scan the input strings from left to right to identify $\mathcal{L}_{\boldsymbol{x}}$ and $\mathcal{L}_{\boldsymbol{y}}$ and compute $\mathcal{C}_{\boldsymbol{x}}$ and $\mathcal{C}_{\boldsymbol{y}}$. At this point, if we can determine the order using conditions (C1) and/or (C2) of Lemma 1, then we terminate immediately returning the order.

Step 2: We compute the first mismatch position, h, between $\boldsymbol{x}$ and $\boldsymbol{y}$; that is, for $1 \leq i < h$, we have $\boldsymbol{x_i} = \boldsymbol{y_i}$ and $\boldsymbol{x_h} \neq \boldsymbol{y_h}$. Now, by applying Lemma 5, we can ignore the letters to its left, because they are equal in $\boldsymbol{x}$ and $\boldsymbol{y}$. Note that the case when h lies within $\boldsymbol{x}_0(\boldsymbol{y}_0)$ can be handled easily.

Step 3: Assume that the nearest $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$ to the right of h is at position $\ell_x + 1$ ($\ell_y + 1$) in $\boldsymbol{x}$ ($\boldsymbol{y}$). The case when h lies within $\boldsymbol{x}_j(\boldsymbol{y}_j)$ again can be handled easily.

Step 4: Now we focus on $\boldsymbol{x'} = \boldsymbol{x}_h..\boldsymbol{x}_{\ell_x}$ and $\boldsymbol{y'} = \boldsymbol{y}_h..\boldsymbol{y}_{\ell_y}$. Essentially, we will construct a map as is done in the ADRS algorithm. But we will not construct the map completely; rather we will construct only the part of the map that is relevant to the computation in a different way. To do this we count the number of occurrences of each letter $\alpha \in \Sigma$ within an appropriate range as follows. We start with the highest letter and continue downward. Assuming that $\sigma = |\Sigma|$, we use two σ-length arrays $count_{\boldsymbol{x}}[1..\sigma]$ and $count_{\boldsymbol{y}}[1..\sigma]$ as follows. Suppose we are counting the number of $\alpha \in \Sigma$. Then we check the leftmost occurrence p of $\beta > \alpha$ in the range $\boldsymbol{x}[h..\ell_x]$ such that there is no occurrence of $\gamma > \beta$ before p. And we count the number of occurrences of α in the range $\boldsymbol{x}[h..p-1]$ and store it in $count_{\boldsymbol{x}}[\alpha]$. Similarly we compute $count_{\boldsymbol{y}}[\alpha]$.

Step 5: At this point, in $count_{\boldsymbol{x}}[1..\sigma]$ ($count_{\boldsymbol{y}}[1..\sigma]$) we have the frequency of each letter $\alpha \in \Sigma$ in the appropriate range. Now the rest is quite easy. We scan $count_{\boldsymbol{x}}$, $count_{\boldsymbol{y}}$ from the higher to lower letters of Σ as follows:

for $\alpha = highest(\Sigma)$ to $lowest(\Sigma)$ **do**
  if $count_{\boldsymbol{x}}[\alpha] == count_{\boldsymbol{y}}[\alpha]$ **then**

▷ *This means either α is nonexistent (when count is zero) or we are in Condition (C3). So we need to check the next letter.*

continue

else

▷ *If $count_{\boldsymbol{x}}[\alpha] \neq count_{\boldsymbol{y}}[\alpha]$, then either α is nonexistent in $\boldsymbol{x}$ — when $count_{\boldsymbol{x}}[\alpha]$ is zero — or in $\boldsymbol{y}$ — when $count_{\boldsymbol{y}}[\alpha]$ is zero. That is, we are in Condition (C1) or (C2). So we have $count_{\boldsymbol{x}}[\alpha] < count_{\boldsymbol{y}}[\alpha]$ ($count_{\boldsymbol{y}}[\alpha] < count_{\boldsymbol{x}}[\alpha]$, respectively).*

return $\boldsymbol{x} \prec \boldsymbol{y}$ ($\boldsymbol{y} \prec \boldsymbol{x}$, respectively)

At this point a brief discussion is in order. Recall that the ADRS algorithm runs in $O(n + \sigma)$ time. Because σ is $O(n)$, this running time is optimal. Therefore, we cannot get improvement asymptotically and the theoretical time complexity of the new algorithm matches that of the ADRS algorithm. However, the use of Lemma 5 gives us an opportunity to work much less from a practical point of view, especially for favourable input strings. And this is why, despite the same theoretical time complexity, our new algorithm is an ***input sensitive*** algorithm and in practice should perform better than the ADRS algorithm.

4.2 Suffix sorting and Burrows Wheeler transformation

The suffix permutation [11] of a word $\boldsymbol{w} = w_1 w_2 \dots w_n$ is the permutation $\pi_{\boldsymbol{w}}$ over $\{1, \dots, n\}$, where π_{w_i} is the rank of the suffix $\boldsymbol{w}[i, n]$ in the set of the lexicographically sorted suffixes of $\boldsymbol{w}$. In [12] it is shown how to deduce the Lyndon factorization (Theorem 1) of a text from its suffix permutation; conversely, a strategy is given in [15] for obtaining the suffix array from the Lyndon factorization of a text.

We will outline how our new results from Section 3 can be applied to obtaining a lex-extension suffix array from the V-order factorization of a text – the distinctness of factors in a Lyndon versus V-order factorization of a given string [6, 7] opens more avenues for string processing (such as choosing the factorization with more/less factors for efficiency).

To elaborate, there are three main cases to be handled for the V-factorization algorithm VF in [6, 7] as follows. To determine the V-order factorization $\boldsymbol{x_1} \geq \boldsymbol{x_2} \cdots \geq \boldsymbol{x_k}$ of a string $\boldsymbol{x}$, algorithm VF applies Lemma 3.16 in [6] to substrings $\boldsymbol{x_i}$, $\boldsymbol{x_j}$:

- If (C1) holds for $\boldsymbol{x_i}$, $\boldsymbol{x_j}$ ($\mathcal{L}_{\boldsymbol{x_i}} < \mathcal{L}_{\boldsymbol{x_j}}$) then $\boldsymbol{x_i} > \boldsymbol{x_j}$ in the factorization - the algorithm tracks maximal elements.
- If (C2) holds for $\boldsymbol{x_i}$, $\boldsymbol{x_j}$ then, $\boldsymbol{x_i} < \boldsymbol{x_j}$ if $\boldsymbol{x_i x_j}$ is a Hybrid Lyndon (that is a Lyndon word under lex-extension [6]), and $\boldsymbol{x_i x_j}$ is a factor in the factorization – the algorithm checks for concatenating repetitions.
- If (C3) holds for $\boldsymbol{x_i}$, $\boldsymbol{x_j}$, and if $\boldsymbol{x_i} \prec \boldsymbol{x_j}$ then $\boldsymbol{x_i x_j}$ is a factor in the factorization – the algorithm compares substrings between maximal elements.

As each factor is identified by algorithm VF, its rightmost position is recorded (procedure output) and then all housekeeping variables are re-initialized (procedure RESET) – this essentially converts the remaining suffix of the string into a new string to be factored with no re-visiting of the previously factored elements required. Hence, similarly to Duval's Lyndon decomposition algorithm [10], the linear V-order factoring technique can be used for on-line scenarios which is the setting of our applications.

Now, we are interested in the notion of compatibility for sorting suffixes as introduced in [15]. Let $\boldsymbol{x}$ be a word and $\boldsymbol{u}$ be a substring (factor) of $\boldsymbol{x}$. The sorting of suffixes $\boldsymbol{s_1}$, $\boldsymbol{s_2}$ of $\boldsymbol{u}$, with respect to $\boldsymbol{u}$, is *compatible* with the sorting of the suffixes of $\boldsymbol{x}$ for which $\boldsymbol{s_1}$, $\boldsymbol{s_2}$ are prefixes, with respect to $\boldsymbol{x}$, if they have the same order in both $\boldsymbol{u}$ and $\boldsymbol{x}$. It is shown in [15] that, although compatibility doesn't always hold for lex-order suffix-sorting, when $\boldsymbol{u}$ is chosen to be a substring of Lyndon factors in a factorization then it does hold. In contrast, compatibility always holds for sorting suffixes in V-order, and furthermore, the shorter suffix is always lesser:

Lemma 7. *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 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 2 then both $\boldsymbol{s_2} \prec \boldsymbol{s_1}$ with respect to $\boldsymbol{u}$ and $\boldsymbol{s_2 t_2} \prec \boldsymbol{s_1 t_1}$ with respect to $\boldsymbol{x}$. □

Lemma 2 further shows that suffixes are totally V-ordered by their given order: for any string $\boldsymbol{x} = \boldsymbol{x}[1 \dots n]$, we have $x_n \prec x_{n-1}x_n \prec \cdots \prec \boldsymbol{x}$.

However, to address applications involving conjugates of strings, such as the Burrows Wheeler transform, Lemma 7 doesn't suffice for V-order: when using suffixes to sort all rotations of a string, since each rotation has the same number of maximal elements, therefore implicitly condition (C3) applies — for ordering these suffixes we need the first distinct prefix substrings of the V-forms of the suffixes. We will use *lex-extension ordering* which compares factors in a factorization pair-wise from left to right while each comparison is made in V-order.

Theorem 3. *Let $\boldsymbol{x} \in \Sigma^+$ with V-order factorization $\boldsymbol{x} = \boldsymbol{x_1} \cdots \boldsymbol{x_k}$, and let $\boldsymbol{u} = \boldsymbol{x_i} \cdots \boldsymbol{x_j}$, for $1 \le i \le j \le k$. Then the sorting of the suffixes of $\boldsymbol{u}$ is compatible with the sorting of the suffixes of $\boldsymbol{x}$.*

Proof. The case of the Lyndon factorization is Theorem 3.2 in [15]. The V-order proof thus follows from the Lyndon-like properties of the V-order factorization and by replacing lexorder with lex-extension ordering. □

Equipped with this theorem, the clever incremental suffix sorting & BWT strategy introduced in [15] can be modified for V-order:

Step 1: Compute the V-order factorization of $\boldsymbol{x} = \boldsymbol{v_1} \cdots \boldsymbol{v_k}$ in linear time [6, 7].

Step 2: Compute the lex-extension order suffix array of each of $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ in linear time [9].

Step 3: Obtain the BWT($\boldsymbol{v_i}$) from each SA($\boldsymbol{v_i}$): for a suffix $\boldsymbol{v_i} = \boldsymbol{x}[h \ldots m]$ the BWT character is $\boldsymbol{x}[h-1]$.

Step 4: Merge the sorted suffixes in Step 2 using ADRS algorithm [2] to obtain the suffix array of $\boldsymbol{v_1 v_2}$. For the merge, if $\boldsymbol{v_j} \succ \boldsymbol{v_k}$, then the chosen suffix for the new array is $\boldsymbol{v_k}$, otherwise it is $\boldsymbol{v_j v_k}$.

Step 5: Obtain the BWT of the merged sorted suffixes in Step 4. If the chosen suffix for the new array was $\boldsymbol{v_k}$, then the BWT character is given by BWT($\boldsymbol{v_k}$); otherwise it is BWT($\boldsymbol{v_j}$) since the prefix $\boldsymbol{x}[1 \ldots h-1]$ in $\boldsymbol{x}$ is rotated as $\boldsymbol{v_j v_k} \ldots \boldsymbol{x}[1 \ldots h-1]$.

Step 6: Compute the lex-extension order suffix array of $\boldsymbol{v_3}$ and merge it with the suffix array of $\boldsymbol{v_1 v_2}$ from Step 4 and obtain the BWT.

Step 7: Repeat until all the V-factors have been incrementally processed.

Overall, for iterating over k factors, the time complexity is $O(k^2 n)$, with each iteration taking $O(kn)$. As expressed in [15] 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. Note that in Step 4 above, the new string comparison algorithm presented in Section 4.1 can be applied when input-sensitivity is relevant.

5 Future Research

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 integers, $21 \prec 12$ and no permutation works; whereas interchanging the first and last letters does for $142 \prec 243$ since $241 \prec 342$, which generalizes to requiring that the rightmost substrings of their V-forms are in V-order.

We propose studying such permutations in the context of the gene team problem: to find a set of genes that appear in two or more species, possibly in a different order, but within a given distance in each chro-

mosome – this has impact in understanding genome evolution and function [16].

References

1. Ali Alatabbi, Jacqueline W. Daykin, M. Sohel Rahman, and William F. Smyth. Simple linear comparison of strings in V–order – (extended abstract). In *International Workshop on Algorithms & Computation (WALCOM)*, volume 8344 of *Lecture Notes in Computer Science*, pages 80–89. Springer, 2014.
2. Ali Alatabbi, Jacqueline W. Daykin, M. Sohel Rahman, and William F. Smyth. Simple linear comparison of strings in V–order. *Fundamenta Informaticae*, To Appear, 2015.
3. K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, iv – the quotient groups of the lower central series. *Ann. Math.*, pages 68:81–95, 1958.
4. Maxime Crochemore and Wojciech Rytter. *Jewels of stringology.* World Scientific, 2002.
5. T.-N. Danh and 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)*, pages 43–53, 1996.
6. D. E. Daykin, J. W. Daykin, and W. F. Smyth. A linear partitioning algorithm for hybrid Lyndons using V–order. *Theoret. Comput. Sci.*, 483:149–161, 2013.
7. David E. Daykin, Jacqueline W. Daykin, and William F. Smyth. String comparison and Lyndon–like factorization using V–order in linear time. In *Symp. on Combinatorial Pattern Matching*, volume 6661, pages 65–76, 2011.
8. D.E. Daykin and J.W. Daykin. Lyndon–like and V–order factorizations of strings. *J. Discrete Algorithms*, (1):357–365, 2003.
9. J. W. Daykin and W. F. Smyth. A bijective variant of the Burrows–Wheeler transform using V–order. *Theoret. Comput. Sci.*, 531:7789, 2014.
10. Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
11. Jean-Pierre Duval and Arnaud Lefebvre. Words over an ordered alphabet and suffix permutations. *RAIRO Theor. Inform. Appl.*, 36(3):249–259, 2002.
12. Christophe Hohlweg and Christophe Reutenauer. Lyndon words, permutations and trees. *Theor. Comput. Sci.*, 307(1):173–178, 2003.
13. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In Ricardo A. Baeza-Yates, Edgar Ch ávez, and Maxime Crochemore, editors, *Symp. on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2003.
14. M. Lothaire. *Combinatorics on Words.* Reading, MA (1983); 2nd Edition, Cambridge University Press, Cambridge (1997). Addison–Wesley, 1983.

15. Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Suffix array and Lyndon factorization of a text. *J. Discrete Algorithms*, 28:2–8, 2014.
16. Biing-Feng Wang and Chien-Hsin Lin. Improved algorithms for finding gene teams and constructing gene team trees. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(5):1258–1272, 2011.