Simple Linear Comparison of Strings in V-Order[⋆]
(Extended Abstract)

Ali Alatabbi[1], Jackie Daykin[1,2],
M. Sohel Rahman[1,3,⋆⋆], and William F. Smyth[1,⋆⋆⋆]

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

Abstract. In this paper we focus on a total (but non-lexicographic) ordering of strings called V-order. We devise a new linear-time algorithm for computing the V-comparison of two finite strings. In comparison with the previous algorithm in the literature, our algorithm is both conceptually simpler, based on recording letter positions in increasing order, and more straightforward to implement, requiring only linked lists.

Keywords: algorithm, array, comparison, complexity, data structure, lexicographic order, linear, linked-list, V-order, Lyndon word, string, total order, word.

1 Introduction

An important task required in many combinatorial computations is deciding the relative order of two members of a totally ordered set [KS-98, R-03], for instance

⋆ Part of this research was carried out when Rahman was visiting King's College as a Commonwealth Fellow. This research work is also partially supported by a CodeCrafters-Investortools Research Grant for CSE BUET. (http://www.codecraftersintl.com/researchgrant.html)

⋆⋆ Partially Supported by a Commonwealth Fellowship and an ACU Titular Award.

⋆⋆⋆ Supported in part by the Natural Sciences & Engineering Research Council of Canada.

organizing words in a natural language dictionary. Binary comparison of finite strings (words) thus arises as a primitive operation, a building block, in more complex procedures, which therefore requires efficient implementation.

In this paper we first discuss some known techniques for totally ordering sets, and then introduce our contribution: a new linear string comparison algorithm using V-order.

Given an integer $n \geq 1$ and a nonempty set of symbols Σ (bounded or unbounded), a string of length n over Σ takes the form $x = x_1 \dots x_n$ with each $x_i \in \Sigma$. The classic and commonly used method for organizing sets of strings is lexicographic (dictionary) order. Formally, if Σ is a totally ordered alphabet then *lexicographic ordering* (lexorder) $u < v$ with $u, v \in \Sigma^+$ is defined if and only if either u is a proper prefix of v, or $u = ras,\ v = rbt$ for some $a, b \in \Sigma$ such that $a < b$ and for some $r, s, t \in \Sigma^*$.

Lexorder is a very natural method for deciding precedence and organizing information which also finds many uses in computer science, typically in constructing data structures and related applications:

– Building indexes for information retrieval, particularly self-indexes which replace the text and support almost optimal space and search time [NM-07].
– Constructing suffix arrays, which record string suffix starting positions in the lexorder of the suffixes, and thus support binary search [KA-03, KSB-06, NZC-09].
– The Burrows-Wheeler Transform (BWT), which applies suffix sorting, and exhibits data clustering properties, hence is suitable for preprocessing data prior to compression activities [ABM-08, CDP-05].
– The application of automata for bioinformatics sequence alignment. The BWT is extended for finite automata representing the multiple alignment problem - the paths in the automaton are sorted into lexorder thus extending the suffix sorting framework related to the classic BWT [SVM-11].
– An important class in the study of combinatorics on words is *Lyndon words* [L-83] - strings (words) which are lexicographically least amongst the cyclic rotations of their letters (characters) – see also [S-03]; furthermore, any string can be uniquely factored into Lyndon words [CFL-58] - Duval's algorithm cleverly detects the lexicographic order between factors in linear time [Du-83, D-11]. The Lyndon decomposition allows for efficient 'divide-and-conquer' of a string into patterned factors; numerous applications include: periodic musical structures [C-04], string matching [BGM-11, CP-91], and algorithms for digital geometry [BLPR-09].
– Hybrid Lyndon structures, introduced in [DDS-13], based on two methods of ordering strings one of which is lexorder.

Naively, lexorder $u < v$ can be decided in time linear in the length of the shorter string, and space linear in the length of the longer string; various data structures may be used for enhancing this string comparison. In [DIS-94] the Four Russians technique [IS-92] is proposed to compare strings of length n on a bounded alphabet in $O(1)$ time, while for an unbounded alphabet the parallel construction of a merged suffix tree using the CRCW PRAM model [IS-92] is proposed that can be constructed in $O(\log n \log\log n)$ time using $O(n/\log n)$ processors; using this tree, sequential comparison requires $O(\log\log n)$ time.

A class of lexorder-type total orders is easily obtained from permuting the usual order $1, 2, \ldots n$ of pairwise comparison of letters, along with interchanging $<$ with $>$ and so on; for example *relex order* (reverse lexicographic) [R-03], and *co-lexorder* (lexorder of reversed strings) studied and applied to string factorization in [DDS-09].

Non-lexicographic methods include deciding precedence by minimal change such as Gray's *reflected binary code*, where two successive values differ in only one bit, hence well-suited for error correction in digital communications [G-53, S-97]. A more recent example is *V-order* [D-85, DaD-96, DaD-97] which is the focus of this paper: we first introduce this technical method for comparing strings and then consider it algorithmically.

Let Σ be a totally ordered alphabet, and let $\boldsymbol{u} = u_1u_2...u_n$ be a string over Σ. Define $h \in \{1, \ldots, n\}$ by $h = 1$ if $u_1 \leq u_2 \ldots \leq u_n$; otherwise, by the unique value such that $u_{h-1} > u_h \leq u_{h+1} \leq u_{h+2} \leq \ \ldots \leq u_n$. Let $\boldsymbol{u}^* = u_1u_2...u_{h-1}u_{h+1}...u_n$, where the star $*$ indicates deletion of the letter u_h. Write $\boldsymbol{u}^{s*}$ for $(...(\boldsymbol{u}^*)^*...)^*$ with $s \geq 0$ stars [1]. Let $g = \max\{u_1, u_2, ..., u_n\}$, and let k be the number of occurrences of g in $\boldsymbol{u}$. Then the sequence $\boldsymbol{u}, \boldsymbol{u}^*, \boldsymbol{u}^{2*}, ...$ ends $g^k, ..., g^2, g^1, g^0 = \boldsymbol{\varepsilon}$. In the *star tree* each string $\boldsymbol{u}$ over Σ labels a vertex, and there is a directed edge from $\boldsymbol{u}$ to $\boldsymbol{u}^*$, with $\boldsymbol{\varepsilon}$ as the root.

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

Example 1. Over the binary alphabet with $0 < 1$: in lexorder, $0101 < 01110$; in V-order, $0101 \prec 01110$.
Over the naturally ordered integers: in lexorder, $123456 < 2345$; in V-order, $2345 \prec 123456$.
Over the naturally ordered Roman alphabet: in lexorder, *eabecd* $<$ *ebaedc*; in V-order, *ebaedc* $\prec$ *eabecd*.

String comparison in V-order $\prec$ was first considered algorithmically in [DDS-11, DDS-13] - the dynamic longest matching suffix of the pair of input strings, together with a doubly-linked list which simulated letter deletions and hence paths in the star tree, enabled deciding order; these techniques achieved V-comparison in worst-case time and space proportional to string length - thus asymptotically the same as naive comparison in lexorder.

Currently known applications of V-order, utilizing linear-time V-comparison, and generally derived from lexorder or Lyndon cases are as follows:

[1] Note that this star operator, as defined in [DaD-96], [DD-03] etc, is distinct from the Kleene star operator: Kleene star is applied to sets, while this V-star is applied to strings.

- A V-order structure, an instance of a hybrid Lyndon word and known as a V-word [DD-03], similarly to the classic Lyndon case, gives an instance of an African musical rhythmic pattern [CT-03].
- Linear factorization of a string into factors (V-words) sequentially [DDS-11] and in parallel [DDIS-13] - yielding factors which are distinct from the Lyndon factorization of the given string [DDS-13].
- Modification of a linear suffix array construction [KA-03] from lexorder to V-order [DS-13] thus allowing efficient V-ordering of the cyclic rotations of a string.
- Applying the above suffix array modification to compute a novel Burrows-Wheeler transform (V-BWT) using, not the usual lexorder, but rather V-order [DS-13] - achieving instances of enhanced data clustering.

These initial avenues suggest that further uses of V-order, analogous to the practical functions listed for lexorder and Lyndon words, will continue to arise, including for instance those for suffix trees - thus necessitating efficient implementations of the primitive V-comparison.

We introduce here a new algorithm for computing the V-comparison of two finite strings - the advantage is that it is both conceptually simpler, based on recording letter positions in increasing order, and more straightforward to implement, requiring only linked lists. The time complexity is $O(n + |\Sigma|)$ and similarly the space complexity is $O(n + |\Sigma|)$. However, in computational practice the alphabet, like the input, can be assumed to be finite - at most $O(n)$ - and so the algorithm runs in essentially linear time.

2 V-Order String Comparison Algorithm

In this section, we present a novel linear-time algorithm for V-order string comparison. Before going into the algorithmic details, we present relevant definitions and results from the literature useful in describing and analyzing our algorithm, starting with a unique representation of a string.

Definition 2. *([DD-03, DDS-11, DDS-13]) 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 possibly empty $\boldsymbol{x_i}$, $i = 0, 1, \ldots, k$, where g is the largest letter in $\boldsymbol{x}$ – thus we suppose that g occurs exactly k times.

The following lemma is the key to our algorithm.

Lemma 1. *([DaD-96, DD-03, DDS-11, DDS-13]) Suppose we are given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$ with the corresponding V-forms as follows:*

$$\boldsymbol{v} = \boldsymbol{v}_0 \mathcal{L}_v \boldsymbol{v}_1 \mathcal{L}_v \boldsymbol{v}_2 \cdots \boldsymbol{v}_{j-1} \mathcal{L}_v \boldsymbol{v}_j$$

$$\boldsymbol{x} = \boldsymbol{x}_0 \mathcal{L}_x \boldsymbol{x}_1 \mathcal{L}_x \boldsymbol{x}_2 \cdots \boldsymbol{x}_{k-1} \mathcal{L}_x \boldsymbol{x}_k$$

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

- $\mathcal{L}_v < \mathcal{L}_x$
- $\mathcal{L}_v = \mathcal{L}_x$ *and* $j < k$
- $\mathcal{L}_v = \mathcal{L}_x$, $j = k$ *and* $\boldsymbol{v}_h \prec \boldsymbol{x}_h$.

Lemma 2. *([DDS-11, DDS-13]) Suppose we are given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$. If $\boldsymbol{v}$ ($\boldsymbol{x}$) is a subsequence of $\boldsymbol{x}$ ($\boldsymbol{v}$) then $\boldsymbol{v} \prec \boldsymbol{x}$ ($\boldsymbol{x} \prec \boldsymbol{v}$).*

We will use some simple data structures, which are initialized by preprocessing steps. We use $Map_u(a)$ to store, in increasing order, the positions of the character a in a string $\boldsymbol{u}$. $Map_u(\Sigma)$ records the 'maps' of all $a \in \Sigma$. To construct $Map_u(\Sigma)$ we take an array of size Σ. For each $a \in \Sigma$, we construct a linked list that stores the positions $i \in [1..|\boldsymbol{u}|]$ in increasing order such that $\boldsymbol{u}[i] = a$.

Example 2. Suppose we have a string $\boldsymbol{u}$ as follows:

$$\begin{array}{rccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \boldsymbol{u} = & 8 & 5 & 8 & 2 & 1 & 8 & 7 & 6 & 5 & 4 & 3 \end{array}$$

$Map_u(\Sigma)$ is shown below for the string $\boldsymbol{u}$ defined above.

$$\begin{tikzcd}
1 \arrow[d] & 2 \arrow[d] & 3 \arrow[d] & 4 \arrow[d] & 5 \arrow[d] & 6 \arrow[d] & 7 \arrow[d] & 8 \arrow[d] \\
5 & 4 & 11 & 10 & 2 \arrow[d] & 8 & 7 & 1 \arrow[d] \\
 & & & & 9 & & & 3 \arrow[d] \\
 & & & & & & & 6
\end{tikzcd}$$

This leads to the following lemma.

Lemma 3. *Given a string $\boldsymbol{u}$ of length n we can build $Map_u(\Sigma)$ in $O(n + |\Sigma|)$ time and space.*

Proof. Proof will be provided in the journal version. □

We will now prove a number of new lemmas that will be used in the string comparison algorithm – first we will introduce some notation. Let $firstMiss(\boldsymbol{u}, \boldsymbol{v})$ denote the first mismatch entry between $\boldsymbol{u}, \boldsymbol{v}$. More formally, we say $\ell = firstMiss(\boldsymbol{u}, \boldsymbol{v})$ if and only if $\boldsymbol{u}[\ell] \neq \boldsymbol{v}[\ell]$ and $\boldsymbol{u}[i] = \boldsymbol{v}[i]$, for all $1 \leq i < \ell$. In what follows, the notion of a global mismatch and a local mismatch is useful in the context of two strings $\boldsymbol{u}, \boldsymbol{v}$ and their respective substrings $\boldsymbol{u'}, \boldsymbol{v'}$. In particular, $firstMiss(\boldsymbol{u}, \boldsymbol{v})$ would be termed as the global mismatch in this context and $firstMiss(\boldsymbol{u'}, \boldsymbol{v'})$ would be termed as a local mismatch, i.e., local to the corresponding substrings. For this global/local notion, the context $\mathcal{C}$ is important and is defined with respect to the two strings and their corresponding substrings, i.e., the context here would be denoted by $\mathcal{C}\langle(\boldsymbol{u}, \boldsymbol{u'}), (\boldsymbol{v}, \boldsymbol{v'})\rangle$. Also, for the V-form of a string $\boldsymbol{u}$ we will use the following convention: $\mathcal{L}_{u,\ell}$ denotes the ℓ-th $\mathcal{L}_u$

in the V-form of $\boldsymbol{u}$ and $pos(\mathcal{L}_{u,\ell})$ will be used to denote its index/position in $\boldsymbol{u}$. With this extended notation, the V-form of $\boldsymbol{u}$ can be rewritten as follows:

$$\boldsymbol{u} = \boldsymbol{u}_0\ \mathcal{L}_{u,1}\ \boldsymbol{u}_1\ \mathcal{L}_{u,2}\ \boldsymbol{u}_2\ \cdots\ \boldsymbol{u}_{j-1}\ \mathcal{L}_{u,j}\ \boldsymbol{u}_j.$$

Moreover, within the context $\mathcal{C}$, the strings $\boldsymbol{u}, \boldsymbol{v}$ are referred to as the *superstrings* and $\boldsymbol{u'}, \boldsymbol{v'}$ as the *substrings.*

Lemma 4. *Suppose we are given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$ with the corresponding V-forms as follows:*

$$\boldsymbol{v} = \boldsymbol{v}_0\mathcal{L}_v\boldsymbol{v}_1\mathcal{L}_v\boldsymbol{v}_2\cdots\boldsymbol{v}_{j-1}\mathcal{L}_v\boldsymbol{v}_j$$

$$\boldsymbol{x} = \boldsymbol{x}_0\mathcal{L}_x\boldsymbol{x}_1\mathcal{L}_x\boldsymbol{x}_2\cdots\boldsymbol{x}_{k-1}\mathcal{L}_x\boldsymbol{x}_k$$

Assume that $\mathcal{L}_v = \mathcal{L}_x$ and $j = k$. Let $h \in \{0\ldots\max(j,k)\}$ be the least integer such that $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. Now assume that $\ell_h = firstMiss(\boldsymbol{v}_h, \boldsymbol{x}_h)$ and $\ell_f = firstMiss(\boldsymbol{v}, \boldsymbol{x})$. In other words, ℓ_h is the index of the first mismatch entry between the substrings $\boldsymbol{v}_h, \boldsymbol{x}_h$, whereas ℓ_f is the index of the first mismatch entry between the two strings $\boldsymbol{v}$ and $\boldsymbol{x}$. Then we must have $\ell_f = \sum_{i=0}^{h-1}(|\boldsymbol{v}_i| + 1) + \ell_h$. (Or equivalently, $\ell_f = \sum_{i=0}^{h-1}(|\boldsymbol{x}_i| + 1) + \ell_h$.)

Proof. Proof will be provided in the journal version. $\square$

Corollary 1. *If in Case 2 of Lemma 4 we have $\ell_f = pos(\mathcal{L}_{v,\ell})$, then $\boldsymbol{v}_h$ is a proper prefix of $\boldsymbol{x}_h$.*

Interestingly, we can extend Lemma 4 further if we consider the (inner) contexts within (outer) contexts as the following lemma shows. In other words V-form can be applied recursively and independently as shown in [DDS-11]. In what follows, for given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$ with corresponding V-forms, the condition that $\mathcal{L}_v = \mathcal{L}_x$, $j = k$ will be referred to as *Cond-I*$(\boldsymbol{v}, \boldsymbol{x})$.

Lemma 5. *Suppose we are given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$ with corresponding V-forms, and assume that Cond-I$(\boldsymbol{v}, \boldsymbol{x})$ holds. Now consider the (outer) context $\mathcal{C}^0\langle(\boldsymbol{v}, \boldsymbol{v}_{h^0}), (\boldsymbol{x}, \boldsymbol{x}_{h^0})\rangle$, where h^0 is the least integer such that $\boldsymbol{v}_{h^0} \neq \boldsymbol{x}_{h^0}$.*

Now similarly consider the V-forms of $\boldsymbol{v}_{h^0}$ and $\boldsymbol{x}_{h^0}$ and assume that Cond-I$(\boldsymbol{v}_{h^0}, \boldsymbol{x}_{h^0})$ holds. Further, consider the (inner) context $\mathcal{C}^1\langle(\boldsymbol{v}_{h^0}, \boldsymbol{v}_{h^1}), (\boldsymbol{x}_{h^0}, \boldsymbol{x}_{h^1})\rangle$, where h^1 is the least integer such that $\boldsymbol{v}_{h^1} \neq \boldsymbol{x}_{h^1}$.

Then the global mismatch of the context $\mathcal{C}^0$ coincides with the local mismatch of the context $\mathcal{C}^1$.

Proof. Proof will be provided in the journal version. $\square$

Corollary 2. *Given nested contexts $\mathcal{C}^i, 0 \leq i \leq k$ satisfying the hypotheses of Lemma 5, the global mismatch of context $\mathcal{C}^0$ coincides with the local mismatch of context $\mathcal{C}^k$.*

Corollary 2 establishes that the first global mismatch will always be the first mismatch as we go further within inner contexts through the chain of outer and inner contexts.

Now we can focus on the string comparison algorithm: Algorithm $CompareV$. Suppose we are given two distinct strings $\boldsymbol{p}$ and $\boldsymbol{q}$, then the algorithm performs the following steps.

Step 1: **Preprocessing Step.** Compute $Map_p(\Sigma)$ and $Map_q(\Sigma)$. We also compute the first mismatch position ℓ_f between $\boldsymbol{p}$ and $\boldsymbol{q}$. This will be referred to as the global mismatch position and will be independent of any context within the iterations of the algorithm. Then we repeat the following sub-steps in Step 2. During different iterations of the execution of these stages we will be considering different contexts by proceeding from outer to inner contexts. Initially, we will start with the outermost context, i.e., $\mathcal{C}^0\langle(\boldsymbol{p}, \boldsymbol{p}_{h^0}), (\boldsymbol{q}, \boldsymbol{q}_{h^0})\rangle$, where h^0 is the least integer such that $\boldsymbol{p}_{h^0} \neq \boldsymbol{q}_{h^0}$. At each iteration, we will be considering the largest $\alpha \in \Sigma$ that is present within one of the superstrings in the context. In other words, if the current context is $\mathcal{C}^0$, as is the case during the initial iteration, we will consider the largest α such that $\alpha \in \boldsymbol{p}$ or $\alpha \in \boldsymbol{q}$.

Step 2: Throughout this step we will assume that the current context is $\mathcal{C}\langle(\boldsymbol{v}, \boldsymbol{v}_h), (\boldsymbol{x}, \boldsymbol{x}_h)\rangle$, where h is the least integer such that $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. So, initially we have $\mathcal{C} = \mathcal{C}^0$. Suppose we are now considering $\alpha \in \Sigma$, then it must be the largest $\alpha \in \Sigma$ such that either $\alpha \in \boldsymbol{v}$ or $\alpha \in \boldsymbol{x}$. We proceed to the following sub-steps:

Step 2.a: We compute $Map_v(\alpha)$ from $Map_p(\alpha)$ where $Map_v(\alpha)$ contains the positions that are only within the range of $\boldsymbol{v}$ in the current context $\mathcal{C}$. Similarly, we compute $Map_x(\alpha)$ from $Map_q(\alpha)$ where $Map_x(\alpha)$ contains the positions that are only within the range of $\boldsymbol{x}$ in the current context $\mathcal{C}$. Now we compare $Map_v(\alpha)$ and $Map_x(\alpha)$, which yields two cases.

Step 2.a.(i): In this case, $Map_v(\alpha) = Map_x(\alpha)$.
This means that within the current context $\mathcal{C}$, considering the V-form of the superstrings $\boldsymbol{v}$ and $\boldsymbol{x}$, we must have $\mathcal{L}_v = \mathcal{L}_x$ and $j = k$. So, we need to check Condition 3 of Lemma 1. We identify h such that h is the least integer with $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. By Lemmas 4, 5 and Corollary 2 we know that this h can be easily identified because it is identical to the global mismatch position ℓ_f.
Then we iterate to Step 2 again with the inner context $\mathcal{C}^1\langle(\boldsymbol{v}_h, \boldsymbol{v}_{h^1}), (\boldsymbol{x}_h, \boldsymbol{x}_{h^1})\rangle$, where h^1 is the least integer such that $\boldsymbol{v}_{h^1} \neq \boldsymbol{x}_{h^1}$. In other words, we assign $\mathcal{C} = \mathcal{C}^1$ and then repeat Step 2 for $\beta \in \Sigma$ where $\beta < \alpha$.

Step 2.a.(ii): In this case, $Map_v(\alpha) \neq Map_x(\alpha)$.
[C1] If $Map_v(\alpha) = \emptyset$ ($Map_x(\alpha) = \emptyset$), we have Condition 1 of Lemma 1 satisfied (ε is the least string in V-order) and hence

we return $\boldsymbol{v} \prec \boldsymbol{x}$ ($\boldsymbol{x} \prec \boldsymbol{v}$). Note that this effectively decides $\boldsymbol{p} \prec \boldsymbol{q}$ ($\boldsymbol{q} \prec \boldsymbol{p}$) and the algorithm terminates.
[C2] If $|Map_v(\alpha)| < |Map_x(\alpha)|$ ($|Map_x(\alpha)| < |Map_v(\alpha)|$), we have Condition 2 of Lemma 1 satisfied and hence we return $\boldsymbol{v} \prec \boldsymbol{x}$ ($\boldsymbol{x} \prec \boldsymbol{v}$). Similarly, this effectively decides $\boldsymbol{p} \prec \boldsymbol{q}$ ($\boldsymbol{q} \prec \boldsymbol{p}$) and the algorithm terminates.
[C3] Otherwise, we have $\mathcal{L}_v = \mathcal{L}_x$ and $j = k$. So, we need to check Condition 3 of Lemma 1, and identify h such that h is the least integer such that $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. By Lemmas 4, 5 and Corollary 2 we know that h can be easily identified because it is identical to the global mismatch position ℓ_f. Now we do a final check as to whether $\boldsymbol{v}_h$ is a subsequence (in fact, a prefix) of $\boldsymbol{x}_h$ according to Corollary 1. If so, then by Lemma 2 we return $\boldsymbol{v} \prec \boldsymbol{x}$ ($\boldsymbol{x} \prec \boldsymbol{v}$), which decides that $\boldsymbol{p} \prec \boldsymbol{q}$ ($\boldsymbol{q} \prec \boldsymbol{p}$) and the algorithm terminates. Otherwise, we return to Step 2 with the inner context $\mathcal{C}^1\langle(\boldsymbol{v}_h, \boldsymbol{v}_{h^1}), (\boldsymbol{x}_h, \boldsymbol{x}_{h^1})\rangle$, where h^1 is the least integer such that $\boldsymbol{v}_{h^1} \neq \boldsymbol{x}_{h^1}$. In other words, we assign $\mathcal{C} = \mathcal{C}^1$ and then repeat Step 2 again.

To prove the correctness of the algorithm we need the following lemmas.

Lemma 6. *Step 2 of Algorithm CompareV can be realized through a loop that considers each character $\alpha \in \Sigma$ in decreasing order, skipping the ones that are absent in both $\boldsymbol{v}$ and $\boldsymbol{x}$ or in the current context.*

Proof. Proof will be provided in the journal version. □

Lemma 7. *Algorithm CompareV terminates at some point.*

Proof. Note that Algorithm *CompareV* can terminate only by conditions [C1] and [C2] of Step 2.a.(ii). Also recall that the input of the algorithm is two distinct strings. Furthermore, we have computed a global mismatch position ℓ_f. Hence, clearly at some point we will reach either [C1] or [C2] of Step 2.a.(ii). Therefore, the algorithm will definitely terminate. □

The correctness of the algorithm follows immediately from Lemmas 1, 2, 6 and 7. Finally we analyze the running time of Algorithm *CompareV* as follows.

Lemma 8. *Algorithm CompareV runs in $O(n + |\Sigma|)$ time and space.*

Proof. Proof will be provided in the journal version. □

Note that a naive $O(n^2)$ rendition of this algorithm was proposed by a reviewer in [DDS-13].

3 Conclusion

Lexicographic orderings have also been considered in the case of parallel computations: for instance, an optimal algorithm for lexordering n integers is given in [I-86], and parallel Lyndon factorization in [DIS-94, DDIS-13]. Analogously, we propose future research into parallel forms of V-ordering strings.

References

[ABM-08] Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows-Wheeler trans- form: data compression, suffix arrays, and pattern matching, p. 352. Springer (2008)

[BGM-11] Breslauer, D., Grossi, R., Mignosi, F.: Simple real-time constant-space string matching. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 173–183. Springer, Heidelberg (2011)

[BLPR-09] Brlek, S., Lachaud, J.-O., Provençal, X., Reutenauer, C.: Lyndon + Christoffel = digitally convex. Pattern Recognition 42(10), 2239–2246 (2009)

[C-04] Chemillier, M.: Periodic musical sequences and Lyndon words. Soft Computing - A Fusion of Foundations, Methodologies and Applications 8(9), 611–616 (2004) ISSN: 1432-7643 (Print), 1433-7479 (Online)

[CT-03] Chemillier, M., Truchet, C.: Computation of words satisfying the "rhythmic oddity property" (after Simha Arom's works). Inf. Proc. Lett. 86, 255–261 (2003)

[CFL-58] Chen, K.T., Fox, R.H., Lyndon, R.C.: Free differential calculus, IV - The quotient groups of the lower central series. Ann. Math. 68, 81–95 (1958)

[CDP-05] Crochemore, M., Désarménien, J., Perrin, D.: A note on the Burrows-Wheeler transformation. Theor. Comput. Sci. 332(1-3), 567–572 (2005)

[CP-91] Crochemore, M., Perrin, D.: Two-way string-matching. J. Assoc. Comput. Mach. 38(3), 651–675 (1991)

[D-85] Daykin, D.E.: Ordered ranked posets, representations of integers and inequalities from extremal poset problems. In: Rival, I. (ed.) Graphs and Order, Proceedings of a Conference in Banff, Canada. NATO Advanced Sciences Institutes Series C: Mathematical and Physical Sciences, vol. 147, pp. 395–412. Reidel, Dordrecht-Boston (1984, 1985)

[D-11] Daykin, D.E.: Algorithms for the Lyndon unique maximal factorization. J. Combin. Math. Combin. Comput. 77, 65–74 (2011)

[DaD-96] Danh, T.-N., Daykin, D.E.: The structure of V-order for integer vectors. In: Hilton, A.J.W. (ed.) Congr. Numer., vol. 113, pp. 43–53. Utilas Mat. Pub. Inc., Winnipeg (1996)

[DaD-97] Danh, T.-N., Daykin, D.E.: Ordering integer vectors for coordinate deletions. J. London Math. Soc. 55(2), 417–426 (1997)

[DD-03] Daykin, D.E., Daykin, J.W.: Lyndon-like and V-order factorizations of strings. J. Discrete Algorithms 1, 357–365 (2003)

[DD-08] Daykin, D.E., Daykin, J.W.: Properties and construction of unique maximal factorization families for strings. Internat. J. Found. Comput. Sci. 19(4), 1073–1084 (2008)

[DDIS-13] Daykin, D.E., Daykin, J.W., Iliopoulos, C.S., Smyth, W.F.: Generic algorithms for factoring strings. In: Aydinian, H., Cicalese, F., Deppe, C. (eds.) Ahlswede Festschrift. LNCS, vol. 7777, pp. 402–418. Springer, Heidelberg (2013)

[DDS-09] Daykin, D.E., Daykin, J.W., (Bill) Smyth, W.F.: Combinatorics of unique maximal factorization families (UMFFs). In: Janicki, R., Puglisi, S.J., Rahman, M.S. (eds.) Fund. Inform, vol. 97-3, pp. 295–309 (2009); Special Issue on Stringology

[DDS-11] Daykin, D.E., Daykin, J.W., Smyth, W.F.: String comparison and lyndon-like factorization using V-order in linear time. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 65–76. Springer, Heidelberg (2011)

[DDS-13] Daykin, D.E., Daykin, J.W., Smyth, W.F.: A linear partitioning algorithm for Hybrid Lyndons using V-order. Theoret. Comput. Sci. 483, 149–161 (2013)

[DIS-94] Daykin, J.W., Iliopoulos, C.S., Smyth, W.F.: Parallel RAM algorithms for factorizing words. Theoret. Comput. Sci. 127, 53–67 (1994)

[DS-13] Daykin, J.W., Smyth, W.F.: A bijective variant of the Burrows-Wheeler transform using V-Order (submitted)

[Du-83] Duval, J.P.: Factorizing words over an ordered alphabet. J. Algorithms 4, 363–381 (1983)

[G-53] F. Gray, Pulse code communication, U.S. patent no. 2,632,058 (March 17, 1953)

[I-86] Iliopoulos, C.S.: Optimal cost parallel algorithms for lexicographical ordering, Purdue University, Tech. Rep. 86-602 (1986)

[IS-92] Iliopoulos, C.S., Smyth, W.F.: Optimal algorithms for computing the canonical form of a circular string. Theoret. Comput. Sci. 92(1), 87–105 (1992)

[KSB-06] Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)

[KA-03] Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003)

[KS-98] Kreher, D.L., Stinson, D.R.: Combinatorial Algorithms: Generation, Enumeration, and Search. CRC Press (1998)

[L-83] Lothaire, M.: Combinatorics on Words. Addison-Wesley, Reading (1983); 2nd edn. Cambridge University Press, Cambridge (1997)

[NM-07] Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys - CSUR 39(1), 2-es (2007)

[NZC-09] Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proc. 2009 Data Compression Conf., pp. 193–202 (2009)

[R-03] Ruskey, F.: Combinatorial Generation (Unpublished book). CiteSeerX: 10.1.1.93.5967, on combinatorics (2003)

[S-97] Savage, C.: A survey of combinatorial Gray codes. SIAM Rev. 39(4), 605–629 (1997)

[SVM-11] Sirén, J., Välimäki, N., Mäkinen, V.: Indexing finite language representation of population genotypes. In: Przytycka, T.M., Sagot, M.-F. (eds.) WABI 2011. LNCS, vol. 6833, pp. 270–281. Springer, Heidelberg (2011)

[S-03] Smyth, B.: Computing patterns in strings, p. 423. Pearson (2003)