A Faster V-order String Comparison Algorithm

Ali Alatabbi[1], Jacqueline W. Daykin[1,2,3], Neerja Mhaskar[4], M. Sohel Rahman[5], and William F. Smyth[1,4,6]

[1] Department of Informatics, King's College London, UK
ali.alatabbi@kcl.ac.uk, jackie.daykin@kcl.ac.uk
[2] Department of Computer Science
Aberystwyth University, Wales & Mauritius
[3] Department of Information Science
Stellenbosch University, South Africa
[4] Algorithms Research Group, Department of Computing & Software
McMaster University, Canada
pophlin@mcmaster.ca, smyth@mcmaster.ca
[5] Department of Computer Science & Engineering
Bangladesh University of Engineering & Science
msrahman@cse.buet.ac.bd
[6] School of Engineering & Information Technology
Murdoch University, Western Australia

Abstract. V-order is a total order on strings that determines an instance of Unique Maximal Factorization Families (UMFFs) [7–10], a generalization of Lyndon words [12]. V-order has also recently 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) [11]. The central problem of efficient V-ordering of strings was considered in [2–4, 9, 10], culminating in a remarkably simple, linear time, constant space comparison algorithm [1]. In this paper we improve on this result to achieve significant speed-up in almost all cases of interest.

Keywords: combinatorics, experiments, lexorder, linear, on-line algorithm, probability, string comparison, V-comparison, V-order

1 Introduction

This paper extends current knowledge on the non-lexicographic ordering technique known as V-order [6]. New combinatorial insights are obtained which are linked to computational settings. The first of these leads to an improvement on the linear-time V-order string comparison algorithm described in [1].

2 Preliminaries

We are given a finite totally ordered set of cardinality $\sigma = |\Sigma|$, called the *alphabet*, whose elements are *characters* (equivalently *letters*). A *string* is a sequence of zero or more characters over Σ. A string $\boldsymbol{x} = x_1 x_2 \cdots x_n$ of *length* $|\boldsymbol{x}| = n$ is represented by $\boldsymbol{x}[1..n]$, where $\boldsymbol{x}[i] \in \Sigma$ for $1 \leq i \leq n$. The set of all non-empty strings over the alphabet Σ is denoted by Σ^+. The *empty string* of zero length is denoted by ε, with $\Sigma^* = \Sigma^+ \cup \varepsilon$. If $\boldsymbol{x} = \boldsymbol{uwv}$ for strings $\boldsymbol{u}, \boldsymbol{w}, \boldsymbol{v} \in \Sigma^*$, then $\boldsymbol{u}$ is a *prefix*, $\boldsymbol{w}$ is a *substring* or *factor*, and $\boldsymbol{v}$ is a *suffix* of $\boldsymbol{x}$. We denote by $\boldsymbol{s}[i \ldots j]$, or $s_i \cdots s_j$, the substring of $\boldsymbol{s}$ that starts at position i and ends at position j. Notably, if $i > j$, $s[i \ldots j] = \varepsilon$; that is, $s[i \ldots j]$ is the empty string. If $\boldsymbol{x} = \boldsymbol{u}^k$ (a concatenation of k

Ali Alatabbi, Jacqueline W. Daykin, Neerja Mhaskar, M. Sohel Rahman, William F. Smyth: A Faster V-order String Comparison Algorithm, pp. 38–49.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

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*. For further stringological definitions, theory and algorithmics see [5, 13].

We define an order that is not lexorder (dictionary order), called V-order, as well as some of its notable properties, both combinatorial and algorithmic. V-order was explored in [6] and subsequently studied extensively in the literature both combinatorially and algorithmically [1–4, 7, 9–11].

Let $\boldsymbol{x} = x_1x_2\cdots x_n$ be a string over Σ. Define $h \in \{1,\ldots,n\}$ by $h = 1$ if $x_1 \le x_2 \le \cdots \le x_n$; otherwise, by the unique value such that $x_{h-1} > x_h \le x_{h+1} \le x_{h+2} \le \cdots \le x_n$. 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. We write $\boldsymbol{x}^{s*}$ for $(\ldots(\boldsymbol{x}^*)^*\ldots)^*$ with $s \ge 0$ 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 $\boldsymbol{x}, \boldsymbol{x}^*, \boldsymbol{x}^{2*}, \ldots$ ends in $g^k, \ldots, g^1, g^0 = \boldsymbol{\varepsilon}$. From all strings $\boldsymbol{x}$ over Σ, we use this process to 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 1. *[6, 7, 9, 10] We define V-*order $\prec$ *between distinct strings $\boldsymbol{x}$, $\boldsymbol{y}$. 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 not, then 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 $\boldsymbol{s} \ne \boldsymbol{t}$ but $|\boldsymbol{s}| = |\boldsymbol{t}| = m$ say. Let $j \in 1..m$ be the greatest integer such that $\boldsymbol{s}[j] \ne \boldsymbol{t}[j]$. 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^*.*

For instance, using the natural ordering of integers, if $\boldsymbol{x} = 2631$, then $\boldsymbol{x}^* = 263$, $\boldsymbol{x}^{2*} = 26$, $\boldsymbol{x}^{3*} = 6$, $\boldsymbol{x}^{4*} = \boldsymbol{\varepsilon}$, and so $26 \prec 2631$ while $2631 \prec 94$.

Definition 2. *[6, 7, 9, 10] 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) 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 3. *[6, 7, 9, 10] 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 $h \in 0..\max(j, k)$ be the least integer such that $\boldsymbol{x}_h \ne \boldsymbol{y}_h$. Then $\boldsymbol{x} \prec \boldsymbol{y}$ if, and only if, one of the following conditions hold:

(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.

Lemma 4. *[9, 10] For given strings $\boldsymbol{x}$ and $\boldsymbol{v}$, if $\boldsymbol{v}$ is a proper subsequence of $\boldsymbol{x}$, then $\boldsymbol{v} \prec \boldsymbol{x}$.*

Theorem 5. *[1, 4] For any strings $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{x}$, $\boldsymbol{y}$: $\boldsymbol{x} \prec \boldsymbol{y} \Leftrightarrow \boldsymbol{uxv} \prec \boldsymbol{uyv}$.*

Note that, according to this result, a comparison of two strings can ignore equal prefixes (or suffixes) — see Algorithm `COMPARE`.

Lemma 6. *[1] 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 $\boldsymbol{y}\mu \prec \boldsymbol{x}\lambda$.*

Lemma 6 led to Algorithm `COMPARE` (Figure 1), described in [1][1]. Note that, in contrast to comments made in the Conclusion of [10], this tells us that in fact V-order comparison can be conducted in a positional manner as in lexorder comparison — in fact, `COMPARE` is an on-line algorithm [13] that requires only a one-character window. Notably, the first V-order comparison algorithm was presented in [9], and this was followed up by a couple of other interesting algorithms [2, 3, 10] before algorithm `COMPARE` was presented in [1].

```
procedure COMPARE(x, y, δ)
    i ← 1; j ← 1; δ ← 0
    while j ≤ |y| and i ≤ |x| and x[i] = y[j] do
        i ← i + 1; j ← j + 1
    if i > |x| or j > |y| then
        if |x| = |y| then
            δ ← 0; return
        if i > |x|  then
            δ ← −1; return
        δ ← 1; return
    while true do
        while j ≤ |y| and x[i] > y[j] do
            j ← j + 1
        if j > |y| then
            δ ← 1; return
        else
            i ← i + 1
        while i ≤ |x| and x[i] < y[j] do
            i ← i + 1
        if i > |x| then
            δ ← −1; return
        else
            j ← j + 1
```

Figure 1. Comparing $\boldsymbol{x}$ of length m and $\boldsymbol{y}$ of length n in V-order: $\boldsymbol{x} \prec, =, \succ \boldsymbol{y}$ according as $\delta = -1, 0, 1$.

Lemma 7. *[1–3, 9, 10] V-comparison requires linear time and constant space.*

[1] This version corrects the comparisons in lines 5 & 8 of the original presentation.

3 V-order String Comparison

In this section, we present a new algorithm to compare strings in V-order, which is sensitive to the structure of the input strings – the `COMPARE-Sensitive` algorithm (Figure 2). We show that probabilistically `COMPARE-Sensitive` runs faster than `COMPARE` in almost all cases of interest. We also show experimental results comparing the two algorithms. From these experiments we see that `COMPARE-Sensitive` runs at least twice as fast as `COMPARE`, in almost all cases of interest. Moreover, the new algorithm can easily be converted to work in an on-line setting, just like `COMPARE`. In addition to the new algorithm, we give a minor correction to the `COMPARE` algorithm (Figure 1).

3.1 `COMPARE-Sensitive` Algorithm

Suppose we are asked to compare two strings $\boldsymbol{x}$, $\boldsymbol{y}$ with V-forms

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

In this representation we can "most of the time" determine the order simply by applying conditions (C1) and (C2) of Lemma 3 to $\mathcal{L}_{\boldsymbol{x}}$ and $\mathcal{L}_{\boldsymbol{y}}$: if one maximum is greater than the other, or if the maxima are equal but have different frequencies of occurrences, then we are done. The `COMPARE-Sensitive` Algorithm (Figure 2) uses this observation to compare strings $\boldsymbol{x}$ and $\boldsymbol{y}$. The first few lines of the algorithm show the very simple preprocessing required for each string $\boldsymbol{x}, \boldsymbol{y}$ to check for conditions (C1) and (C2).

```
procedure COMPARE-Sensitive(x, y, δ)
    SCAN(x, |x|; L_x, occx)
    SCAN(y, |y|; L_y, occy)
    if L_x ≠ L_y then
        δ ← SIGN(L_x − L_y)
    else
        if occx ≠ occy then
            δ ← SIGN(occx − occy)
        else
            COMPARE-C3(x, y, L_x, δ)
```

Figure 2. Apply conditions (C1) and (C2) to $\boldsymbol{x}$ and $\boldsymbol{y}$: for $n \gg \sigma$, almost always $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, while at the same time usually $occx \neq occy$, so that the routine `COMPARE-C3` does not need to be executed at all.

```
procedure SCAN(x, |x|; L_x, occx)
    L_x ← x[1]; occx ← 1
    for i ← 2 to |x| do
        if x[i] ≮ L_x then
            if x[i] = L_x then
                occx ← occx + 1
            else
                L_x ← x[i]; occx ← 1
```

Figure 3. Scan traverses the string $\boldsymbol{x}$ to compute the maximum $\mathcal{L}_{\boldsymbol{x}}$ and its frequency $occx$.

```
procedure COMPARE-C3(x, y, L_x, δ)
    i ← 1; j ← 1; δ ← 0
    while j ≤ |y| and i ≤ |x| and x[i] = y[j] do
        i ← i + 1; j ← j + 1
    if i > |x| or j > |y| then
        if |x| = |y| then
            δ ← 0; return
        if i > |x| then
            δ ← −1; return
        δ ← 1; return
    if x[i] = L_x then δ ← −1; return                          ▷ s(x_h) = ε
    if y[j] = L_x then δ ← 1; return                           ▷ s(y_h) = ε
    while true do
        while j ≤ |y| and x[i] > y[j] and y[j] < L_x do
            j ← j + 1
        if y[j] = L_x or j > |y| then                          ▷ L_x = L_y and y[j] ⊁ L_x
            δ ← 1; return
        else
            i ← i + 1
        while i ≤ |x| and x[i] < y[j] and x[i] < L_x do
            i ← i + 1
        if x[i] = L_x or i > |x| then                          ▷ x[i] ⊁ L_x
            δ ← −1; return
        else
            j ← j + 1
```

Figure 4. Comparing $\boldsymbol{x}$ of length m and $\boldsymbol{y}$ of length n in V-order, when $\mathcal{L}_x = \mathcal{L}_y$ and $occx = occy$: $\boldsymbol{x} \prec, =, \succ \boldsymbol{y}$ according as $\delta = -1, 0, 1$.

However if conditions (C1) and (C2) of Lemma 3 fail, we need to invoke the COMPARE algorithm to check for condition (C3). A potential disadvantage in invoking COMPARE is that it must rescan at least one of the strings in its full length. We propose a modification to the COMPARE algorithm, the COMPARE-C3 algorithm, which takes advantage of the values $\mathcal{L}_x = \mathcal{L}_y$ computed by SCAN to avoid this rescanning. COMPARE-C3 is motivated by the following observations:

Observation 1 *Recall that the condition (C3) of Lemma 3 first eliminates a common prefix (up to the beginning of substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$) in strings $\boldsymbol{x}$ and $\boldsymbol{y}$, and then compares only the substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ containing the first mismatching letter in $\boldsymbol{x}$ and $\boldsymbol{y}$ while scanning the strings from left to right. Therefore, instead of comparing the entire strings $\boldsymbol{x}$ and $\boldsymbol{y}$ as in* COMPARE*, it suffices to first identify the substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ and only compare them.*

Observation 2 *Observe that the substring $\boldsymbol{x_h}$ ($\boldsymbol{y_h}$) is either followed by the empty string (when $\boldsymbol{x_h}$ ($\boldsymbol{y_h}$) is a suffix of $\boldsymbol{x}$ ($\boldsymbol{y}$)), or $\mathcal{L}_x = \mathcal{L}_y$ (when $\boldsymbol{x_h}$ ($\boldsymbol{y_h}$) is not a suffix of $\boldsymbol{x}$ ($\boldsymbol{y}$)). We use this observation to identify the end of substring $\boldsymbol{x_h}$ ($\boldsymbol{y_h}$) in $\boldsymbol{x}$ ($\boldsymbol{y}$).*

Similar to COMPARE, the first **while** loop in COMPARE-C3 identifies a common prefix (alternatively the first mismatching position in strings $\boldsymbol{x}$ and $\boldsymbol{y}$ from the left). Note that this prefix might include a common prefix of substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ identified under condition (C3) of Lemma 3. We denote the suffixes of $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ without a common prefix by $s(\boldsymbol{x_h})$ and $s(\boldsymbol{y_h})$, respectively. Then the second **while** loop in

`COMPARE-C3`, unlike `COMPARE`, only compares the suffixes $s(\boldsymbol{x_h})$ and $s(\boldsymbol{y_h})$, to compare strings $\boldsymbol{x}$ and $\boldsymbol{y}$. Therefore, it terminates when it encounters $\mathcal{L}_{\boldsymbol{x}}$ or ε, which marks the end of substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ as seen in Observation 2. The two **if** statements before the second **while** loop ensure that `COMPARE-C3` returns correct results when either $s(\boldsymbol{x_h}) = \varepsilon$ or $s(\boldsymbol{y_h}) = \varepsilon$. For completeness we give the pseudocode for SCAN [2] (Figure 3), which is used in `COMPARE-Sensitive` to scan the string $\boldsymbol{x}$ ($\boldsymbol{y}$) to compute the maximum $\mathcal{L}_{\boldsymbol{x}}$ ($\mathcal{L}_{\boldsymbol{y}}$) and its frequency *occx* (*occy*).

If the conditions (C1) and (C2) of Lemma 3 do not hold, then we need to execute `COMPARE-C3` to compare strings $\boldsymbol{x}$ and $\boldsymbol{y}$. For such strings, scanning them and computing $\mathcal{L}_{\boldsymbol{x}}$ ($\mathcal{L}_{\boldsymbol{y}}$) and *occx* (*occy*) might result in an overhead. Therefore, such strings are possibly one of the worst case input strings for which the time required by `COMPARE-Sensitive` is the maximum. Therefore, we refer to them as "*bad strings*".

Lemma 3 *The probability of choosing a pair of bad strings $\boldsymbol{x}$ and $\boldsymbol{y}$ of length n from an alphabet of size σ is*

$$\frac{\sum_{\mathcal{L}_{\boldsymbol{x}}=1}^{\sigma}\sum_{k=1}^{n}\left(\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}\right)^2}{\sigma^{2n}}, \tag{1}$$

where $\mathcal{L}_{\boldsymbol{x}}$ is the maximum letter in $\boldsymbol{x}$, and $k = occx$ is the number of times $\mathcal{L}_{\boldsymbol{x}}$ occurs in $\boldsymbol{x}$.

Proof.

For a given $\mathcal{L}_{\boldsymbol{x}}$ and k the number of strings of length n is computed as follows:

1. the k positions where $\mathcal{L}_{\boldsymbol{x}}$ occurs can be chosen in $\binom{n}{k}$ ways;
2. the remaining $n-k$ positions can be chosen in $(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}$ ways.

Then for fixed values of $\mathcal{L}_{\boldsymbol{x}}$ and k the total number of ways to choose $\boldsymbol{x}$ is

$$\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}.$$

Since we choose $\boldsymbol{y}$ independently of $\boldsymbol{x}$, and $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, the number of ways in which we can choose $\boldsymbol{y}$ for fixed values of $\mathcal{L}_{\boldsymbol{x}}$ and k is also $\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}$.
Therefore the total number of ways to choose a pair of bad strings $\boldsymbol{x}$ and $\boldsymbol{y}$ for fixed values of $\mathcal{L}_{\boldsymbol{x}}$ and k is

$$\left(\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}\right)^2.$$

Since $\mathcal{L}_{\boldsymbol{x}} \in [1..\sigma]$ and $k \in [1..n]$, the total number of ways to choose a pair of bad strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is

$$\sum_{\mathcal{L}_{\boldsymbol{x}}=1}^{\sigma}\sum_{k=1}^{n}\left(\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}\right)^2.$$

The total number of ways in which we can choose two strings over an alphabet of size σ is σ^{2n}. Therefore the probability of choosing a bad pair of strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is

[2] For reasons of efficiency, the implementation of `SCAN` does not necessarily conform to the pseudocode given in (Figure 3).

$$\frac{\sum_{\mathcal{L}_{\boldsymbol{x}}=1}^{\sigma}\sum_{k=1}^{n}\left(\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}\right)^2}{\sigma^{2n}}.$$

□

To simplify the computation in Lemma 3, we assume that $|x| = |y| = n$; that is, the strings to be compared are of the same lengths. However, Lemma 3 can easily be extended where $|x| \neq |y|$.

We computed the probabilities for the length of strings (n) ranging from 1 to 100 over alphabets of size $\sigma = 2, 4, 20$, and plotted graphs (see Appendix A) for the same. From the graphs we see that the probability of choosing a pair of bad strings $\boldsymbol{x}$ and $\boldsymbol{y}$ reaches a maximum value, and then drops significantly and stabilities to a constant value as n approaches 100. Since for all practical purposes the strings of interest are very large ($\gg 100$), we conclude that `COMPARE-Sensitive` will run faster than `COMPARE` in all cases of interest.

3.2 Experimental results

In this section, we show the results of the experiments conducted to compare the performance of `COMPARE` and `COMPARE-Sensitive` on different classes of bad strings. As expected from the Lemma 3, `COMPARE-Sensitive` runs much faster than `COMPARE`. In fact from the experimental results we see that `COMPARE-Sensitive` is twice as fast as `COMPARE`.

The experiments were conducted on a Windows 10 64-bit Operating System, with Intel(R) Xeon(R) CPU E31245 v3 @ 3.40GHz x64-based processor, having an installed memory (RAM) of 32.0 GB. The code was implemented in the `C++` language using Visual Studio 2017.

Strings $\boldsymbol{x}$ in the sample pair of strings $(\boldsymbol{x}, \boldsymbol{y})$ were chosen from the sample strings found at:

`http://www.cas.mcmaster.ca/ bill/strings/`,

and the string $\boldsymbol{y}$ was generated as a permutation of $\boldsymbol{x}$. Since $\boldsymbol{y}$ is a permutation of $\boldsymbol{x}$, the strings $\boldsymbol{x}$ and $\boldsymbol{y}$ fail conditions (C1) and (C2) of Lemma 3. Therefore each pair $(\boldsymbol{x}, \boldsymbol{y})$ is a pair of bad strings. In particular, strings $\boldsymbol{x}$ were chosen from the sample DNA, protein, random ($\sigma = 2$, $\sigma = 21$) and highly periodic strings available at the above URL. The lengths of the strings in a pair range from $n = 10K$ to $50K$. To minimize the effects of external factors (for example delays caused due to interrupts etc.) on the experiments, we executed the algorithms `COMPARE` and `COMPARE-Sensitive` on the same pair of strings ten times, and used the *minimum* time taken by each of them. In addition to this, we take the average of the time taken for 100 different pairs, to get the final data point for comparison in the graphs.

Let T_c and T_{cs} be the slopes of the `COMPARE` and `COMPARE Sensitive` lines seen in the graphs. Let

$$\alpha = \frac{T_c}{T_{cs}}.$$

Then the α values computed from the graphs for DNA strings (see Figure 5), random strings over alphabets 2 and twenty one (see Figures 6, 7), protein strings

(see Figure 8), and highly periodic strings (see Figure 9) are: $\alpha_{DNA} = 2.023$, $\alpha_{rand2} = 2.031$, $\alpha_{rand21} = 1.984$, $\alpha_{protein} = 1.954$, and $\alpha_{hp} = 2.009$, respectively. These α values suggest that `COMPARE-Sensitive` is twice as fast as `COMPARE`. Extrapolating for a value of $n = 1,000,000,000$ (1 billion) for a DNA string, the time taken by `COMPARE` is 24.44 10^{-3} secs while that for `COMPARE-Sensitive` is 12.06 10^{-3} secs. Moreover, it is seen that the size σ of the alphabet does not have any discernible affect on α.

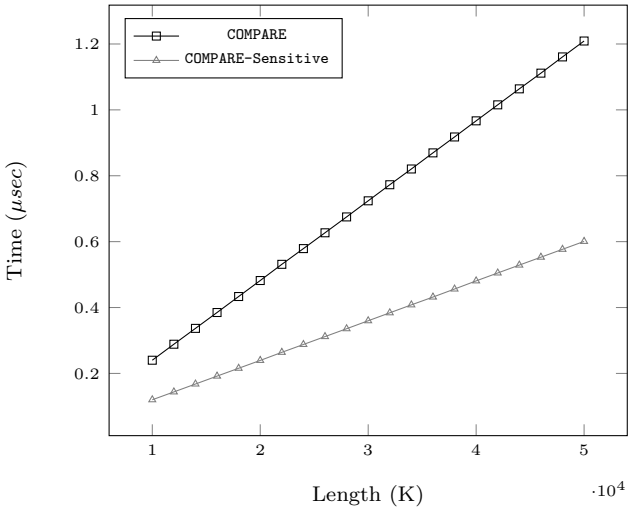

Figure 5. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ is randomly selected from DNA strings of length $2K$ to $50K$ and $\sigma = 4$, and y is a permutation of $\boldsymbol{x}$.

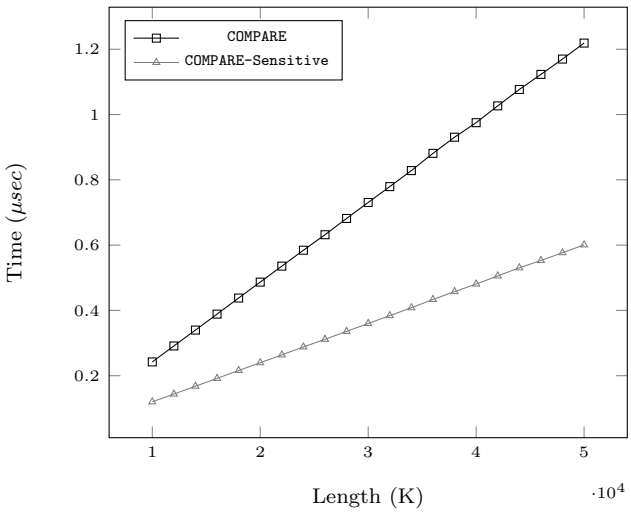

Figure 6. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ is randomly selected from random strings of length $2K$ to $50K$ and $\sigma = 2$, and y is a permutation of $\boldsymbol{x}$.

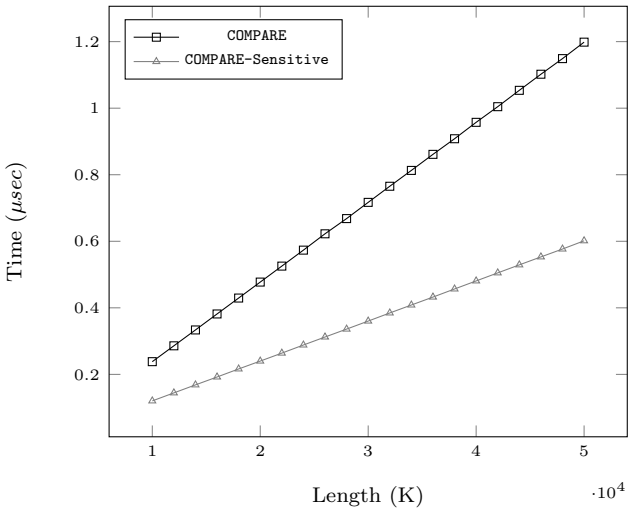

Figure 7. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ is randomly selected from random strings of length $2K$ to $50K$ and $\sigma = 21$, and y is a permutation of $\boldsymbol{x}$.

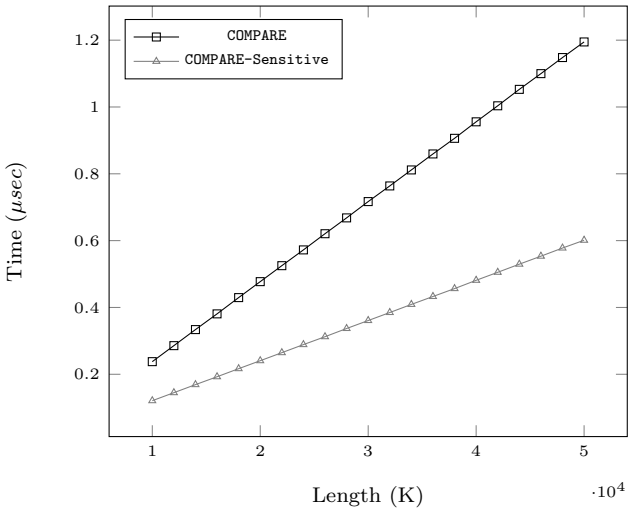

Figure 8. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ is randomly selected from protein strings of length $2K$ to $50K$ and $\sigma = 20$, and y is a permutation of $\boldsymbol{x}$.

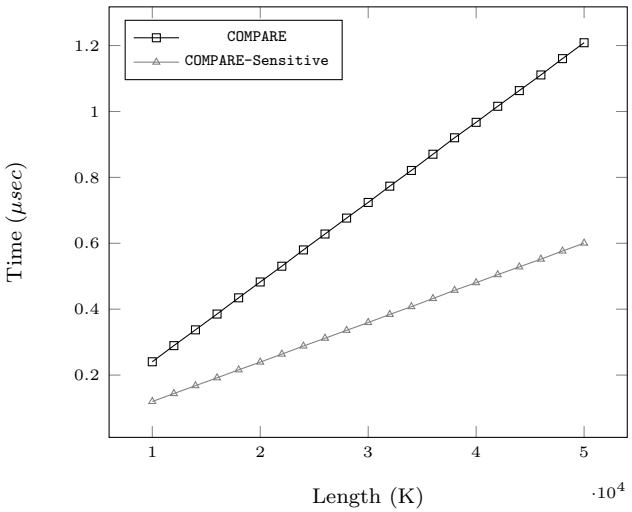

Figure 9. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$ where $\boldsymbol{x}$ is randomly selected from highly periodic strings of length $2K$ to $50K$ and $\sigma = 2$, and y is a permutation of $\boldsymbol{x}$.

Acknowledgments

The third and fifth authors were funded by the NSERC Grant Number: 10536797. The second author was part-funded by the European Regional Development Fund through the Welsh Government.

References

1. A. Alatabbi, J. W. Daykin, J. Kärkkäinen, M. S. Rahman, and W. F. Smyth: *V–order: New combinatorial properties & a simple comparison algorithm.* Discrete Appl. Math., 215 2016, pp. 41–46.
2. A. Alatabbi, J. W. Daykin, M. S. Rahman, and 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.
3. A. Alatabbi, J. W. Daykin, M. S. Rahman, and W. F. Smyth: *Simple linear comparison of strings in V–order.* Fundam. Inform., 139(2) 2015, pp. 115–126.
4. A. Alatabbi, J. W. Daykin, M. S. Rahman, and W. F. Smyth: *String comparison in V–order: New lexicographic properties & on-line applications.* arXiv:1507.07038, 2015.
5. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, New York, NY, USA, 2007.
6. 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), 1996, pp. 43–53.
7. D. E. Daykin and J. W. Daykin: *Lyndon–like and V–order factorizations of strings.* J. Discrete Algorithms, 1(3–4) 2003, pp. 357–365.
8. D. E. Daykin, J. W. Daykin, and 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 (eds.), 2009, pp. 295–309.
9. D. E. Daykin, J. W. Daykin, and 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.
10. D. E. Daykin, J. W. Daykin, and W. F. Smyth: *A linear partitioning algorithm for hybrid Lyndons using V–order.* Theoret. Comput. Sci., 483 2013, pp. 149–161.
11. J. W. Daykin and W. F. Smyth: *A bijective variant of the Burrows– Wheeler transform using V–order.* Theoret. Comput. Sci., 531 2014, pp. 77–89.
12. M. Lothaire: *Combinatorics on Words*, Reading, MA (1983); 2nd Edition, Cambridge University Press, Cambridge (1997)., Addison–Wesley, 1983.
13. B. Smyth: *Computing Patterns in Strings*, Pearson/Addison–Wesley, 2003.

A Figures for experiments conducted to compute probabilities of choosing a pair of bad strings of length n from an alphabet of size σ.

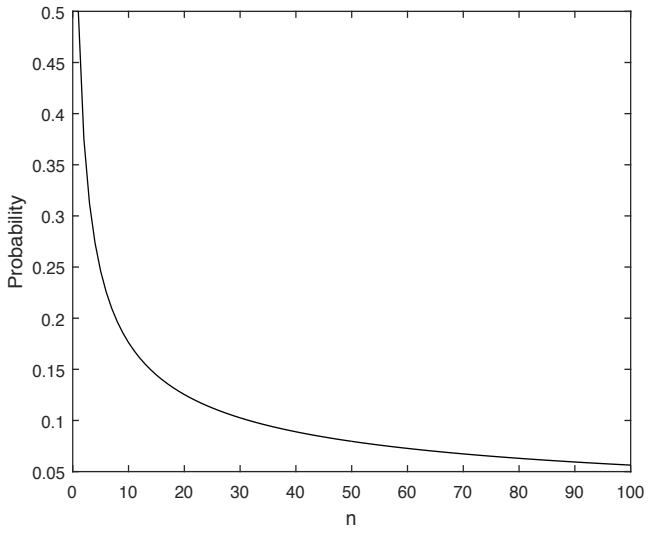

Figure 10. Probabilities computed for strings over alphabet of size $\sigma = \mathbf{2}$, and length $n \in [1..100]$ using Equation (1).

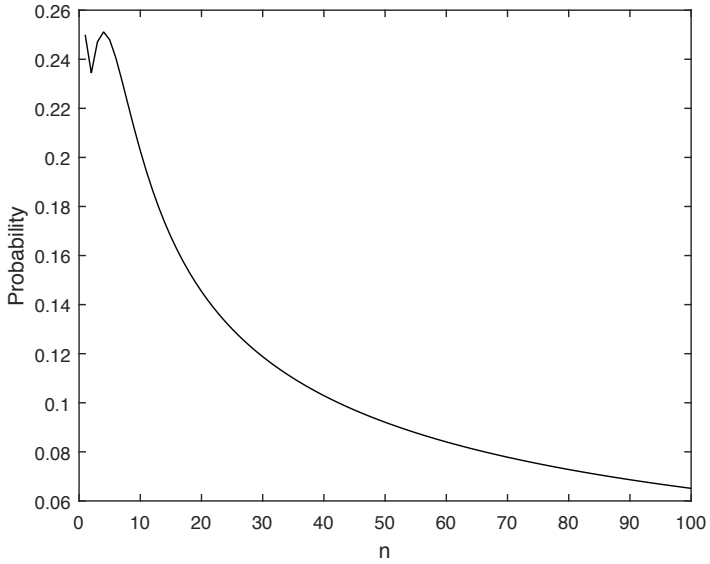

Figure 11. Probabilities computed for strings over alphabet of size $\sigma = \mathbf{4}$, and length $n \in [1..100]$ using Equation (1).

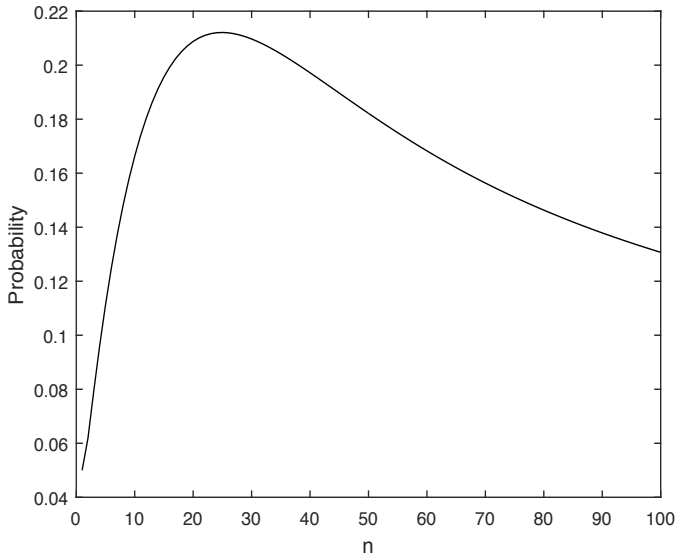

Figure 12. Probabilities computed for strings over alphabet of size $\sigma = \mathbf{20}$, and length $n \in [1..100]$ using Equation (1).