Circular sequence comparison: algorithms and applications

This item was submitted to Loughborough University's Institutional Repository by the/an author.


Additional Information:

- This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. The Creative Commons Public Domain Dedication waiver (http://creativecommons.org/publicdomain/zero/1.0/) applies to the data made available in this article, unless otherwise stated.

Metadata Record: https://dspace.lboro.ac.uk/2134/22155

Version: Published

Publisher: BioMed Central (© Grossi et al)

Rights: This work is made available according to the conditions of the Creative Commons Attribution 4.0 International (CC BY 4.0) licence. Full details of this licence are available at: http://creativecommons.org/licenses/by/4.0/

Please cite the published version.
Circular sequence comparison: algorithms and applications

Roberto Grossi¹², Costas S. Iliopoulos³, Robert Mercas³⁴, Nadia Pisanti¹², Solon P. Pissis³*, Ahmad Retha³ and Fatima Vayani³

Abstract

Background: Sequence comparison is a fundamental step in many important tasks in bioinformatics; from phylogenetic reconstruction to the reconstruction of genomes. Traditional algorithms for measuring approximation in sequence comparison are based on the notions of distance or similarity, and are generally computed through sequence alignment techniques. As circular molecular structure is a common phenomenon in nature, a caveat of the adaptation of alignment techniques for circular sequence comparison is that they are computationally expensive, requiring from super-quadratic to cubic time in the length of the sequences.

Results: In this paper, we introduce a new distance measure based on q-grams, and show how it can be applied effectively and computed efficiently for circular sequence comparison. Experimental results, using real DNA, RNA, and protein sequences as well as synthetic data, demonstrate orders-of-magnitude superiority of our approach in terms of efficiency, while maintaining an accuracy very competitive to the state of the art.

© 2016 Grossi et al. This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made. The Creative Commons Public Domain Dedication waiver (http://creativecommons.org/publicdomain/zero/1.0/) applies to the data made available in this article, unless otherwise stated.
that comprise very small, single-stranded, circular RNA. Their multiple sequence alignment could prove useful in the analysis of their secondary structures and, therefore, the mechanisms by which they infect host plant cells [11].

Naturally-occuring circular proteins are found in both prokaryotes and eukaryotes [1]. Bacteriocins are very small toxins produced by bacteria in order to compete with closely-related bacterial strains. Many of these are circular, including gassericin A, found in Lactobacillus gasseri LA39 [12], and circulin A, found in Clostridium beijerinckii [13]. An interesting phenomenon known to occur naturally in linear protein structures is circular permutation [14]. This can be exemplified by swaposins: proteins highly-similar to saposins, resulting from circularly permuted linear peptide sequences [15]. The ability to align linear sequences from circular proteins can significantly speed up and enhance their analyses, and could also lead to the discovery of novel pairs of circularly permuted proteins.

Our problem
Conventional tools, designed for linear sequences, could yield an incorrectly high genetic distance between closely related circular sequences. Indeed, when sequencing molecules, the position where a circular sequence starts can be totally arbitrary. Due to this arbitrariness, a suitable rotation of one sequence would give much better results for a pairwise alignment, and hence highlight a similarity that any linear alignment would miss. A practical example of the benefit this can bring to sequence analysis is the following. Linearized human (NC_001807) and chimpanzee (NC_001643) MtDNA sequences, obtained from GenBank [16], do not start in the same region. Their pairwise sequence alignment using EMBOSS Needle [17] (default parameters) gives a similarity of 85.1 % and consists of 1195 gaps. However, taking different rotations of these sequences into account yields a much more significant alignment with a similarity of 91 % and only 77 gaps. This example motivates the design of efficient algorithms that are specifically devoted to the comparison of circular sequences [18–20].

In this paper, we consider the pairwise circular sequence comparison problem. Under the edit distance model, it consists in finding an optimal linear alignment of two circular strings. This problem, for two strings $x$ and $y$ of length $m$ and $n \geq m$, respectively, can be solved under the edit distance model in time $O(nm \log m)$ [21]. Several other super-quadratic [22] and approximate quadratic-time [23] algorithms exist. Trivially, for molecular biology applications, the same problem can be solved in time $O(nm^2)$, if extending the problem with scoring matrices and affine gap penalty scores. A direct application of pairwise circular sequence comparison is progressive multiple circular sequence alignment [11, 24, 25]. Multiple circular sequence alignment has also been considered in [26] under the Hamming distance model.

To the best of our knowledge, there is no fast (that is, with sub-quadratic time complexity) and exact (or at least very accurate) algorithm for circular sequence comparison under some realistic model (that is, allowing indels). Taking into account edit distance rather than Hamming distance is computationally challenging as the search space for seeking similarity is wider. Algorithms that speed up the process of string matching, by filtering out candidate positions in which a particular string can never occur, are known as filters. Filters that work for Hamming distance do not work in general for edit distance [27] as well. An exception to this are the $q$-gram filtering techniques [28] that have successfully been used for string matching under the edit distance model (e.g. [29–31]), as well as for multiple local alignments, both under the Hamming [32] and edit [31] distance models.

Our contribution
We present new efficient $q$-gram-based methods for pairwise circular sequence comparison. Specifically, our contribution is threefold.

1. We introduce the $\beta$-blockwise $q$-gram distance between two strings $x$ and $y$, that is, a more powerful generalization of the $q$-gram distance introduced as a string distance measure in [28]. Intuitively, and similarly to [29–31], this generalization comprises partitioning $x$ and $y$ in $\beta$ blocks each, as evenly as possible, computing the $q$-gram distance between the corresponding block pairs, and then summing up the distances computed blockwise.

2. We present an algorithm based on the suffix array [33] that finds the rotation of $x$ such that the $\beta$-blockwise $q$-gram distance between the rotated $x$ and $y$ is minimal, in time and space $O(\beta m + n)$, where $m = |x|$ and $n = |y|$, thereby solving exactly the circular sequence comparison problem under the $\beta$-blockwise $q$-gram distance measure. We also present a simple heuristic algorithm to solve an approximate version of the problem.

3. We present an experimental study, using real and synthetic data, which demonstrates orders-of-magnitude superiority of our approach, in terms of efficiency, while maintaining an accuracy very competitive to the optimal obtained after considering all rotations of $x$ against $y$ using EMBOSS Needle.

The paper is organized as follows. "Definitions and properties" section gives some preliminary definitions, notation, and properties. "Algorithms" section describes two algorithms, one is a heuristic approach and the other
is an exact algorithm for circular sequence comparison under the \( \beta \)-blockwise \( q \)-gram distance measure. "Implementation" section provides details of the implementation of the algorithms. "Experimental results" section presents the experimental results of the performance and accuracy of the algorithms. Finally, "Conclusions" section gives some concluding remarks and future proposals. A preliminary version describing a subset of the results in this paper appeared in [34].

**Definitions and properties**

We begin with a few definitions, following [35]. We think of a string \( x \) of length \( m \) as an array \( x[0..m-1] \), where every \( x[i] \), \( 0 \leq i < m \), is a letter drawn from some fixed alphabet \( \Sigma \) of size \( |\Sigma| = O(1) \). We refer to any string \( x \in \Sigma^q \) as a \( q \)-gram. The empty string of length 0 is denoted by \( \epsilon \). A string \( x \) is a factor of a string \( y \) if there exist two strings \( u \) and \( v \), such that \( y = uvx \). Let \( x \) be a non-empty string and \( y \) be a string. We say that there is an occurrence of \( x \) in \( y \), or, simply, that \( x \) occurs in \( y \), when \( x \) is a factor of \( y \). The Parikh vector associated with a string \( w \in \Sigma^* \) is denoted by \( \mathcal{P}(w) \) and represents a vector of size \( |\Sigma| \), where each component denotes the number of occurrences in \( w \) of the corresponding letter from \( \Sigma \).

Consider the strings \( x, y, u, \) and \( v \), such that \( y = uvx \). If \( u = \epsilon \), then \( x \) is a prefix of \( y \). If \( v = \epsilon \), then \( x \) is a suffix of \( y \). We denote by \( \text{SA} \) the suffix array of \( y \) of length \( n \), that is, an integer array of size \( n \) storing the starting positions of all lexicographically sorted suffixes of \( y \), i.e. for all \( 1 \leq r < n \), we have \( y[\text{SA}[r-1]..n-1] < y[\text{SA}[r]..n-1] \) [33].

Let \( lcp(r,s) \) denote the length of the longest common prefix between \( y[\text{SA}[r]..n-1] \) and \( y[\text{SA}[s]..n-1] \), for all positions \( r, s \) on \( y \), and 0 if they do not have a common prefix. We denote by \( \text{LCP} \) the longest common prefix array of \( y \) defined by \( \text{LCP}[r] = \text{LCP}(r-1,r) \), for all \( 1 \leq r < n \), and \( \text{LCP}[0] = 0 \). The inverse \( \text{iSA} \) of the array \( \text{SA} \) is defined by \( \text{iSA}[\text{SA}[r]] = r \), for all \( 0 \leq r < n \). \( \text{SA} \), \( \text{iSA} \), and \( \text{LCP} \) of \( y \) can be computed in \( O(n) \) time and space [36].

A circular string of length \( m \) can be viewed as a traditional linear string which has the left- and right-most letters wrapped around and glued together in some way. Under this notion, the circular string can be seen as \( m \) different linear strings, which would all be considered equivalent. Given a string \( x \) of length \( m \), we denote by \( x^i = x[i..m-1][0..i-1] \), \( 0 < i < m \), the \( i \)th rotation of \( x \) and \( x^0 = x \). For instance, the string \( x = abababc \) has the following rotations: \( x^1 = babababc, x^2 = abababc \), and so on.

We give some further definitions following [28]. The \( q \)-gram profile of a string \( x \) is the vector \( G_q(x) \), where \( q > 0 \) and \( G_q(x)[v] \) denotes the total number of occurrences of \( q \)-gram \( v \in \Sigma^q \) in \( x \). The \( q \)-gram distance between two strings \( x \) and \( y \) is defined as

\[
D_q(x,y) = \sum_{v \in \Sigma^q} |G_q(x)[v] - G_q(y)[v]|.
\]

Note that \( D_q \) is a pseudo-metric as \( D_q(x,y) \) can be 0 even if \( x \neq y \). \( D_q \) has the following properties [28] for all \( x, y, z \in \Sigma^* \) of length at least \( q \).

1. Positivity: \( D_q(x,y) \geq 0 \)
2. Symmetry: \( D_q(x,y) = D_q(y,x) \)
3. Triangular inequality: \( D_q(x,y) \leq D_q(x,z) + D_q(z,y) \)
4. \(||(x-|y|)|| \leq D_q(x,y) \leq |x| + |y| - 2q - 2 \)
5. \( D_q(x_1,x_2,y_1,y_2) \leq D_q(x_1,y_1) + D_q(x_2,y_2) + 2(q - 1) \), for \( x_1,x_2,y_1,y_2 \in \Sigma^* \)
6. \( D_q(h(x),h(y)) \leq D_q(x,y) \), for a non-length-increasing morphism \( h \) on \( \Sigma^* \).

**Example 1** Let \( x = \text{GGAGTCTA} \), \( y = \text{TTCTAGCG} \), and \( q = 3 \). Table 1 shows the \( q \)-gram profiles of strings \( x \) and \( y \) and the \( q \)-gram distance between them. Each row represents the frequency of a \( q \)-gram in the given string. For succinctness of presentation, only those rows with frequency greater than zero (in either string) are shown, as well as rows representing AAA, CCC, GGG, and TTT as points of reference.

For a given integer parameter \( \beta \geq 1 \), we define a generalization of the \( q \)-gram distance in (1) by partitioning \( x \) and \( y \) in \( \beta \) blocks as evenly as possible, and computing the \( q \)-gram distance between each pair of blocks, one from \( x \) and one from \( y \). The rationale is to enforce locality in the resulting overall distance. For the sake of presentation in the rest of the paper, we assume that the lengths \( |x| = m \) and \( |y| = n \) are both multiples of \( \beta \), so that \( x \) and \( y \) are conceptually partitioned into \( \beta \) blocks, each of size \( m/\beta \) for \( x \) and \( n/\beta \) for \( y \).

**Definition 1** Given strings \( x \) of length \( m \) and \( y \) of length \( n \geq m \) and integers \( \beta \geq 1 \) and \( q > 0 \), the \( \beta \)-blockwise \( q \)-gram distance \( D_{\beta,q}(x,y) \) is defined as

\[
D_{\beta,q}(x,y) = \sum_{j=0}^{\beta-1} D_q\left(x \left\lfloor \frac{jm}{\beta} \right\rfloor \left( \frac{j+1)}{\beta} - 1 \right) \right), \quad y \left\lfloor \frac{jm}{\beta} \right\rfloor \left( \frac{j+1)}{\beta} - 1 \right) \right).
\]

**Example 2** Following Example 1, let \( x = \text{GGAGTCTA} \) and \( y = \text{TTCTAGCG} \), \( q = 3 \), and \( \beta = 2 \). Further let \( x_1 = \text{GGAG} \), \( x_2 = \text{TCTA} \) and \( y_1 = \text{TTCT} \), \( y_2 = \text{AGCG} \) be
the two blocks of \( x \) and \( y \), respectively. Table 2 shows the \( q \)-gram profiles of strings \( x_1 \), \( x_2 \), \( y_1 \), and \( y_2 \) and the \( q \)-gram distance between \( x_1 \) and \( y_1 \) and the \( q \)-gram distance between \( x_2 \) and \( y_2 \).
In this paper, we consider the following problem, where we search for the \( i \)th rotation of \( x \) that minimizes its blockwise distance from \( y \) as defined in (2). Ties are broken arbitrarily.

### Circular Sequence Comparison (CSC)

**Input:** strings \( x \) and \( y \) of lengths \( m \) and \( n \), respectively, and integers \( \beta \geq 1 \) and \( q < m 

**Output:** \( i \) such that \( D_{\beta,q}(x^i, y) \) is minimal

### Algorithms

We use the following result to first give a naïve solution to the CSC problem.

**Lemma 1** [28] If we have space \( \mathcal{O}(|\Sigma|^q) \) available, then the \( q \)-gram distance \( D_q(x, y) \) can be computed in time \( \mathcal{O}(m + n) \) and extra space \( \mathcal{O}(m + n) \) where \( m = |x| \) and \( n = |y| \).

We then apply Lemma 1 to each pair of blocks of \( x \) and \( y \) separately.

**Lemma 2** If we have space \( \mathcal{O}(|\Sigma|^q) \) available, then the \( \beta \)-blockwise \( q \)-gram distance \( D_{\beta,q}(x, y) \) can be computed in time \( \mathcal{O}(m + n) \) and extra space \( \mathcal{O}\left(\frac{m+n}{\beta}\right) \) where \( m = |x| \) and \( n = |y| \).

The naïve algorithm, denoted by \( \text{nCSC} \), computes for \( x' = xx \) the values:

\[
\delta_i = D_{\beta,q}(x'[i \ldots i + m - 1], y),
\]

for all \( 0 \leq i < m \); we report position \( i \) such that \( \delta_i \) is minimal. This requires the application of Lemma 2, \( m \) times. Therefore, we obtain the following.

**Lemma 3** If we have space \( \mathcal{O}(|\Sigma|^q) \) available, then algorithm \( \text{nCSC} \) solves the CSC problem in time \( \mathcal{O}(m(m + n)) \) and extra space \( \mathcal{O}\left(\frac{m+n}{\beta}\right) \).

**Algorithm hCSC: a Heuristic algorithm**

Here we give a simple heuristic algorithm, denoted by \( \text{hCSC} \), to solve the CSC problem faster than \( \text{nCSC} \), and return an approximation of the best rotation.

**Step 1:** We split \( x' = xx \) in \( 2\beta \) non-overlapping string blocks of length \( m/\beta \). We obtain strings \( x_0, x_1, \ldots, x_{2\beta-1} \), such that \( x_i = x'[i \times \frac{m}{\beta}, \ldots, (i+1) \times \frac{m}{\beta} - 1] \), for all \( 0 \leq i < 2\beta \). We split \( y \) in \( \beta \) non-overlapping string blocks of length \( n/\beta \). We obtain strings \( y_0, y_1, \ldots, y_{\beta-1} \), such that \( y_i = y[i \times \frac{n}{\beta}, \ldots, (i+1) \times \frac{n}{\beta} - 1] \), for all \( 0 \leq i < \beta \).

**Step 2:** For a given sequence \( x_j, \ldots, x_{j+\beta-1} \) of strings and \( y \), we compute the \( \beta \)-blockwise \( q \)-gram distance as follows:

\[
\delta_j = D_{\beta,q}\left(x'[\beta \cdot \frac{j m}{\beta}, \ldots, \beta \cdot \frac{j m}{\beta} + m - 1], y\right) = \sum_{i=0}^{\beta-1} D_q(x_{j+i}, y_i).
\]

We compute \( \delta_j \), for all \( 0 \leq j \leq \beta \). We choose \( j_{\text{best}} = j \) such that \( \delta_j \) is minimal, for all \( 0 \leq j \leq \beta \). In other words, we have found a window of length \( m \) starting at position \( j_{\text{best}} \) such that \( (j_{\text{best}} + 1) \mod (m/\beta) = 0 \), consisting of \( \beta \) blocks of length \( m/\beta \) each, that minimizes its \( \beta \)-blockwise \( q \)-gram distance from \( y \).

**Step 3:** To perform a refinement on the position of the window, we consider all starting positions included in
the two blocks starting at positions $i_{\text{best}}$ and $j_{\text{best}} - m/β$. This includes $2m/β - 1$ starting positions in total—we do not need to consider position $j_{\text{best}} - m/β$ as this was already considered by another window in Step 2. Similarly to Step 2, we obtain the $β$-blockwise q-gram distance $δ_i$ between the window starting at position $i$ and $y$, for all $j_{\text{best}} - m/β < i ≤ j_{\text{best}} + m/β - 1$. We report position $i_{\text{best}} = i$ such that $δ_i$ is minimal, for all $j_{\text{best}} - m/β < i ≤ j_{\text{best}} + m/β - 1$.

**Analysis**  Step 1 can be done trivially in time $O(m + n)$. If we have space $O(|\Sigma|^q)$ available, then, by Lemma 1, $D_q(x_{j + i}, y_i)$ can be computed in time $O\left(\frac{m + n}{β}\right)$. By Lemma 2, $δ_i$ can be computed in time $O\left(β\left(\frac{m + n}{β}\right)\right) = O(m + n)$. Hence, Step 2 can be done in time $O(β(m + n))$. In Step 3, the blockwise q-gram distance $δ_i$ between a single window and $y$ can be computed in time $O\left(β\left(\frac{m + n}{β}\right)\right) = O(m + n)$. There exist $2m/β - 1$ such windows. Hence, Step 3 can be done in time $O\left(\frac{m(m + n)}{β}\right)$. Overall, the algorithm requires time $O\left(β(m + n) + \frac{m(m + n)}{β}\right)$ and space $O(|\Sigma|^q + m + n)$.

For practical purposes, setting $β = O(\sqrt{m})$ and $q = O(\log_{|\Sigma|}m)$ gives an algorithm with time complexity $O(\sqrt{m}(m + n))$ and space complexity $O(m + n)$.

**Algorithm saCSC: an exact suffix-array-based algorithm**

The above heuristic hCSC does not guarantee to find the exact value $i$, for which $δ_i = D_{β,q}(x', y)$ is minimal. In particular, when we identify $i_{\text{best}}$ in Step 2, that is, the $j$ for which $δ_j$ is minimal, we take into account only the values of $j$ such that $(j + 1) \mod (m/β) = 0$. Thus, Step 3 cannot guarantee that $i_{\text{best}}$, the local minimum obtained by shifting the window $m/β$ positions to the right and left of $i_{\text{best}}$, is minimal for all $0 ≤ i < m$. In this section, we give a fast and exact algorithm, denoted by saCSC, to find $i$ such that $δ_i = D_{β,q}(x', y)$ is minimal, based on the suffix array (see "Definitions and properties" section).

We partially follow the idea from [37]. This work investigates the string matching problem in the setting of $k$-abelian equivalences: two strings are considered $k$-abelian equivalent for some positive integer $k$, if they have the same length and share the same factors of length at most $k$, including multiplicities. Note that if $k$ is greater than or equal to the string’s length, then the strings must be equal. A version of this result, called extended $k$-abelian equivalence, focuses only on the factors of length $k$. By setting $k = q$, it is quite straightforward to notice the equivalence with $q$-grams. Therefore, in order to avoid confusion we will refer to the former notion from now on as $q$-abelian equivalence.

In [37], the authors propose a linear-time algorithm to solve the string matching problem when looking at $q$-abelian equivalent strings: given a string $x$ of length $m$, a string $y$ of length $n ≥ m$, and a positive integer $q < m$, all factors of $y$ that are $q$-abelian equivalent to $x$ can be found in time and space $O(m + n)$. The idea of the algorithm in [37] consists in constructing the suffix array of the string $xy$, and ranking sets of identical $q$-length prefixes of suffixes in the suffix array in the order of their appearance. Then it constructs new strings based on this ranking, and solves the problem as in the jumbled matching case [38], i.e. identifying all factors of $y$ that have the same Parikh vector as $x$.

We first describe our algorithm for a single block ($β = 1$) and then address the general case ($β ≥ 1$).

**Basic algorithm for $β = 1$.** We construct the suffix array of the string $xy$ and assign a rank $i_q$ to the prefix with length $q$ of each suffix with length at least $q$, based on its order in the suffix array. That is, the first $i_q$ suffixes, of length at least $q$, in the suffix array, all sharing the same prefix of length $q$, will get rank 0; the next $i_q$ suffixes, of length at least $q$, sharing the same prefix of length $q$, different from the previous one, will get rank 1, and so on. Next, based on this ranking, we construct two new strings $x'$ of length $2m - q + 1$ and $y'$ of length $n - q + 1$, such that $x'[i] = j$, if $j$ is the rank of the $q$-length prefix of the $(i + 1)$th suffix of $xx$ in the suffix array of $xyx$ (the same goes for $y$). It is not difficult to see that the ranks go up at most to value $m + n - q + 1$. However, we can reduce this value to $m + 2$ by introducing two new ranks $α_x$ and $α_y$: we can conceptually replace by $α_x$ every letter of $x'$ that does not occur in $y'$, and by $α_y$ every letter of $y'$ that does not occur in $x'$. Hence we can consider that the new strings $x'$ and $y'$ are defined over an integer alphabet of size at most $\min(n - q + 1, m) + 2 ≤ m + 2$.

**Example 3** Let $x = \text{GAGTCTA}$, $y = \text{TCTAGCG}$, and $q = 3$. We denote $xy$ by $z$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z[i]$</td>
<td>G</td>
<td>A</td>
<td>G</td>
<td>T</td>
<td>C</td>
<td>T</td>
<td>A</td>
<td>G</td>
<td>A</td>
<td>G</td>
<td>T</td>
<td>C</td>
<td>T</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>T</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>G</td>
</tr>
<tr>
<td>SA[i]</td>
<td>6</td>
<td>17</td>
<td>1</td>
<td>8</td>
<td>13</td>
<td>19</td>
<td>4</td>
<td>15</td>
<td>11</td>
<td>20</td>
<td>0</td>
<td>7</td>
<td>18</td>
<td>2</td>
<td>9</td>
<td>5</td>
<td>16</td>
<td>12</td>
<td>3</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>LGP[i]</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>$x'[i]$</td>
<td>$a_x$</td>
<td>$a_x$</td>
<td>$a_x$</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>$a_x$</td>
<td>$a_x$</td>
<td>$a_x$</td>
<td>$a_x$</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y'[i]$</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>$a_y$</td>
<td>$a_y$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Here, \( x'[3] = y'[0] = 2 \) denotes that \( x[3..5] = y[0..2] = TCT \) and \( x'[0] = ax \) denotes that \( x[0..2] = GAG \) does not occur in \( y \).

We observe that when identifying the \( q \)-gram distance between two blocks, we can apply the idea in [37], with the only difference that we should also maintain a Parikh vector that stores the differences between the number of occurrences of \( q \)-grams (in fact the new letters given by the ranks) in the current block of \( xx \) and \( y \). Moreover, at the time of the construction of \( y' \), we also construct a Parikh vector \( P(y') \), storing for each letter of \( y' \), the number of its occurrences in \( y' \). Notice that \( |P(y)| \leq m + 2 \).

Later on, when computing the \( q \)-gram distances, we can construct another vector \( \text{diff} \) to store the letter differences between \( P(y') \) and the Parikh vector covering the \( m - q + 1 \) letters of \( x' \) associated with a window of length \( m \) on the string \( xx \). This gives us the current Parikh difference and, in fact, represents the \( q \)-gram distance between the two analyzed blocks, where \( |\text{diff}| \leq m + 2 \). Apart from these, we only need another vector \( \delta \) of size \( m \), which stores at each position \( i \) the actual \( q \)-gram distance \( \delta_i \) between \( y \) and the window starting at position \( i \) in \( xx \), which is the \( i \)th rotation \( x' \) of \( x \).

We use a sliding window of length \( m \) to maintain the above information. When the window is shifted one position to the right, we have to add to the difference-vector \( \text{diff} \) the previous first element of the window, and deduct from it the current last element of it. The distance \( \delta_i \) between \( y' \) and the factor of \( x' \) starting at position \( i \) is thus updated using, in addition, the value of the \( q \)-gram distance \( \delta_{i-1} \) as follows. If, after adding the previous first element to the vector, we have a non-positive value at this position, we update the distance by decreasing the previous value by 1; otherwise, we increase it by 1. If, after deducting the current last element to the vector, we have a non-negative value at this position, we update the distance by decreasing the previous value by 1; otherwise, we increase it by 1. The distance will never be less than the number of occurrences of \( a_x \). Furthermore, if the previous first element was \( a_y \), the new distance decreases by 1, and for every newly added \( a_x \), it increases by 1. As these operations require constant time, after going once through \( x' \) with \( y' \), we obtain the list of distances \( \delta_i \) from \( y \) to each rotation \( x' \) in linear time.

We are now able to give a more formal description of the steps to solve the CSC problem for \( \beta = 1 \), which follow a dynamic programming scheme.

**Step 1:** Construct the \( SA, iSA \), and \( \text{LCP} \) of \( xx \). Rank the \( q \)-length prefixes of suffixes using \( \text{LCP} \)-array queries. Construct \( x' \) and \( y' \), as well as \( P(y') \), the Parikh vector storing, for each letter of \( y' \), the number of its occurrences in \( y' \); making proper use of letters \( a_x \) and \( a_y \), the ranks that do not occur in either \( y' \) or \( x' \), respectively. Further, create \( \text{diff} = P(y') \) and \( \delta_0 = \sum_{i=0}^{\mid P(y') \mid -1} P(y')[i] \).

**Example 4** Following Example 3, let \( x = \text{GAGTCTA} \), \( y = \text{TCTAGCG} \), \( q = 3 \), and \( z = \text{xxx} \).

| \( y' \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| \( \text{diff} \) | G | A | G | T | C | T | C | T | A | G | C | G | x'[1] | 0 | 2 | 0 | 1 | 0 | 2 | 2 | 2 | 1 | 1 |
| \( \delta \) | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The table below represents vector \( \text{diff} \), right after the execution of Step 1, which implies that \( \delta_0 = 5 \).

<table>
<thead>
<tr>
<th>( a_x )</th>
<th>( a_y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**Step 2:** Read the first \( m - q + 1 \) letters of \( x' \), which constitute our sliding window of length \( m \) on the string \( xx \). When reading letter \( x'[i] \), update \( \text{diff} \) by decreasing by 1 the value of the newly read letter, and update \( \delta_0 \) by either increasing the current value of the distance when there were read too many of the current letters, or decreasing it, when more of these letters still occur in \( y' \).

<table>
<thead>
<tr>
<th>( \text{diff} )</th>
<th>x'[1]</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \delta )</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example 5** Following Example 4, the table below represents vector \( \text{diff} \), right after the execution of Step 2, which implies that \( \delta_0 = 6 \).

<table>
<thead>
<tr>
<th>( a_x )</th>
<th>( a_y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**Step 3:** Let \( i \) be the current position in \( x' \) and repeat this step, one position at a time. Shift the window to the right, update the information for \( \text{diff} \):

\[
\text{diff}[x'[i]] = \text{diff}[x'[i]] + 1 \quad \text{and} \quad \text{diff}[x'[i + m]] = \text{diff}[x'[i + m]] - 1,
\]

and calculate \( \delta_{i+1} \), based on this information, sequentially applying the two following rules.
\[ \delta_{i+1} = \begin{cases} \delta_i - 1, & \text{if } \text{diff}[x'[i]] \leq 0 \\ \delta_i + 1, & \text{if } \text{diff}[x'[i]] > 0 \end{cases} \]

\[ \delta_{i+1} = \begin{cases} \delta_{i+1} - 1, & \text{if } \text{diff}[x'[i + m]] \geq 0 \\ \delta_{i+1} + 1, & \text{if } \text{diff}[x'[i + m]] < 0 \end{cases} \]

**Example 6** Following Example 5, the table below represents vector \( \text{diff} \) at iteration \( i' = 3 \) of Step 3, which implies that \( \delta_0 = 4 \). This is in fact the best rotation of \( x \), that is, \( x^3 = \text{TCTAGAG} \).

<table>
<thead>
<tr>
<th>( \alpha_i )</th>
<th>( \beta_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**Correctness** Steps 1 and 2 are trivially correct as at the end of them we have that \( \text{diff} \) is the difference between \( P(y') \) and the vector corresponding to the window. These operations follow directly from the definitions of SA and LCP, and are followed by a simple traversal of the suffix array in order to obtain the ranks and create the \( P(y') \) and \( \text{diff} \) vectors. Also, \( \delta_0 \), which was initially the number of letters in \( y' \), is decreasing as long as the difference between the vectors for a specific letter is non-negative (thus, we still have more occurrences of that letter in \( y' \) compared to the window), and increasing otherwise. In Step 3, we update the difference vector by increasing the value at position \( x'[i] \) and decreasing that of the new letter \( x'[i + m] \) added to the difference. The \( q \)-gram distance at that position is based on the values of the newly obtained difference vector, as well as the \( q \)-gram distance at the previous position: if \( \text{diff}[x'[i]] \leq 0 \), then obviously there were more letters \( x'[i] \) in \( y' \) than in the window, thus we need to decrease, while, if \( \text{diff}[x'[i]] > 0 \), then there were at least as many letters \( x'[i] \) in the window as in \( y' \), and taking one out increases the distance. The complementary reasoning applies to the newly added letter \( x'[i + m] \). The value of \( \delta_i \) never goes below the number of occurrences of \( \alpha_p \) in \( y' \) (it is equal to that, when all other elements of \( \text{diff} \) are 0) and represents the \( q \)-gram distance between \( y \) and \( x' \), the corresponding window of length \( m \) starting at position \( i \) in \( xx \).

**Analysis** In Step 1, constructing \( \text{SA} \), \( \text{iSA} \), and \( \text{LCP} \) of \( xx \) can be done in time and extra space \( O(m + n) \) ("Definitions and properties" section). Furthermore, the construction of \( x', y', P(y'), \text{diff} \), and \( \delta_0 \) is done with the same time and space cost. In Step 2, updating \( \text{diff} \) and \( \delta_0 \) after reading each letter takes constant time, as we execute two operations, thus \( O(m) \) in total. Constant time is required for each iteration in Step 3 to compute the value of \( \delta_i, \quad 1 \leq i < m \), and update \( \text{diff} \), since a constant number of operations are executed, thus \( O(m) \) in total. Hence, we can solve the CSC problem for \( \beta = 1 \) in time and space \( O(m + n) \).

**General algorithm for** \( \beta \geq 1 \). We can now generalize this algorithm to solve the CSC problem for any \( \beta \geq 1 \), which gives algorithm \( \text{saCSC} \). We maintain a Parikh vector for each block, and apply the above basic algorithm for the \( j \)-th block in each string, computing their \( q \)-gram distance. If we denote by \( P_j(y') \) and \( \text{diff}_j \) for all \( 0 \leq j < \beta \), the \( \beta \) Parikh vectors of \( y' \) and of the \( q \)-gram distances, respectively, as well as by \( \delta_{ij} \) the \( q \)-gram distance between the \( j \)-th block of \( y \) and \( x' \), then the updates will be given by the formulae below. Hence, at each position \( i < m \), we can update all of the \( \beta \) Parikh vectors corresponding to the blocks, as previously described, in time \( O(\beta) \). As an example, see here the modification of the previous Step 3, with the other two steps being easily adapted in a similar fashion.

**Step 3**: When shifting the window one position to the right from position \( i \), update the information for every \( \text{diff}_j \), where \( 0 \leq j < \beta \), as follows

\[
\text{diff}_j \left[ x'[\lfloor i + jm/\beta \rfloor] \right] = \text{diff}_j \left[ x'[\lfloor i + jm/\beta \rfloor] \right] + 1
\]

\[
\text{diff}_j \left[ x'[\lfloor i + (j+1)m/\beta \rfloor] \right] = \text{diff}_j \left[ x'[\lfloor i + (j+1)m/\beta \rfloor] \right] - 1,
\]

and calculate \( \delta_{i+1,j} \) based on this information, sequentially applying the two following rules

\[
\delta_{i+1,j} = \begin{cases} \delta_{ij} - 1, & \text{if } \text{diff}_j \left[ x'[\lfloor i + jm/\beta \rfloor] \right] \leq 0 \\ \delta_{ij} + 1, & \text{if } \text{diff}_j \left[ x'[\lfloor i + jm/\beta \rfloor] \right] > 0 \end{cases}
\]

\[
\delta_{i+1,j} = \begin{cases} \delta_{i+1,j} - 1, & \text{if } \text{diff}_j \left[ x'[\lfloor i + (j+1)m/\beta \rfloor] \right] \geq 0 \\ \delta_{i+1,j} + 1, & \text{if } \text{diff}_j \left[ x'[\lfloor i + (j+1)m/\beta \rfloor] \right] < 0 \end{cases}
\]

Therefore, we obtain the following result.

**Theorem 1** Algorithm \( \text{saCSC} \) solves the CSC problem in \( O(\beta m + n) \) time and space.

**Implementation** We implemented algorithms \( \text{nCSC}, \text{hCSC}, \) and \( \text{saCSC} \) as the program \( \text{CSC} \). Given one of the three methods, two sequences \( x \) and \( y \) in (Multi)FASTA format, the number \( \beta \) of blocks, and the length \( q \) of the \( q \)-grams, \( \text{CSC} \) finds the rotation of \( x \) (or an approximation of it) that minimizes its \( \beta \)-blockwise \( q \)-gram distance from \( y \). The implementation is distributed under the GNU General Public License (GPL), and it is available freely at http://www.github.com/solonas13/csc.

Example 7 Consider the following pair of strings obtained from saCSC and the following pair of strings formed for $L = 25$.

The Needleman-Wunsch algorithm for all rotations of $x''$ and string $y''$ gives the following optimal alignment

which tells us that a refined rotation is in fact $x'$, where $i' = i - 13$

We take $p$ block(s) of the prefix of $x'$, concatenate it with a string of equal length $L$ comprised only of letter $\$, where $\notin \Sigma$, and concatenate that with $p$ block(s) of the suffix of $x'$ to form a new string $x''$ of length $3L$. We do the same with $y$ to form a new string $y''$.

The refinement algorithm works by taking all rotations of $x''$ and comparing their similarity to $y''$. Each rotation of $x''$ is compared to $y''$ excluding when a $\$ letter is found at index 0 of the rotation of $x''$. We measure the similarity between the strings for which equality between letters are positively valued; inequalities, insertions, and deletions are negatively valued; and comparisons involving $\$ are neither positively nor negatively valued. The goal of rotating $x''$ serves to find the rotation that maximizes the similarity to $y''$ and, to this end, we make use of the Needleman-Wunsch algorithm.

The rotation of $x''$ which results in the maximum score is chosen as the best rotation, and hence, the final rotation $x'$ of $x$ is computed based on this rotation of $x''$. Ties are broken arbitrarily. We denote this new algorithm, consisting of saCSC and the refinement stage, by saCScr.

The application of the Needleman-Wunsch algorithm on strings of length $3L$ has a time complexity of $O(L^2)$. Considering all rotations of $x''$ results in a time complexity of $O(L^3)$ for the refinement step. Overall, saCScr takes time $O(\beta m + n + L^3)$. 

Refining algorithm saCSC

The application of the $\beta$-blockwise $q$-gram distance via algorithm saCSC suggests that an optimal or a close-to-optimal rotation of $x$ can be found when compared to cNW. Due to the locality property offered by the newly introduced distance notion, it is reasonable to assume that the close-to-optimal rotation returned by saCSC may be refined via some quick heuristics that take into consideration the blocks at both ends.

Let $x'$ be the close-to-optimal rotation of $x$ returned by saCSC. We introduce a new input parameter $0 < p \leq \beta$, which defines the length $L$ of the prefixes and suffixes of $x'$ and $y$ to be considered in the refinement as follows:

$$L = \left\lfloor p \times \frac{m}{\beta} \right\rfloor$$

We take $p$ block(s) of the prefix of $x'$, concatenate it with a string of equal length $L$ comprised only of letter $\$, where $\notin \Sigma$, and concatenate that with $p$ block(s) of the suffix of $x'$ to form a new string $x''$ of length $3L$. We do the same with $y$ to form a new string $y''$.

For comparison purposes, we implemented a naïve algorithm that compares all rotations of $x$ against $y$ using the Needleman-Wunsch algorithm [39] with substitution matrices and affine gap penalty scores [40]; we denote this implementation by cNW. We also implemented the following heuristics. We first use the Smith-Waterman local alignment algorithm [41] to search for the best local alignment of $x$ and $y$ and then use a central match from this local alignment to anchor the global alignment (see also [11]); we denote this implementation by hSW.
Experimental results

The following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600 CPU at 3.4GHz and 12GB of RAM under 64-bit GNU/Linux. All programs were compiled with gcc version 4.7.3. We used both synthetic data and real data. All input datasets referred to in this section are publicly maintained at the same web-site. First, in "Accuracy"—"Time performance" sections, we establish the quality (accuracy and performance) of our methods. Then, in "Application to synthetic data"—"Application to real data" sections, we show applications of our methods.

Accuracy

We began with simulating three DNA sequence datasets using INDELible [42], with each dataset consisting of 12 sequences (denoted by α), each of length approximately 2500 bp (denoted by γ). INDELible produces linear sequences with substitutions, insertions, and deletions at rates defined by the user. Three unique substitution rates (denoted by θ) were set, per dataset, using the substitution model JC69 (Jukes-Cantor, 69): 5, 20, and 35 %. The insertion and deletion rates were set, respectively, to 4 and 6 % (denoted by κ and α), relative to substitution rate of one, similar to those observed in MtDNA in primates and mammals [25]. We refer to these datasets as Original.

To allow for comparison of the performance of the algorithms in realigning randomly rotated sequences, which should be similar to those obtained from sequencing circular DNA structures, such as MtDNA, one random rotation was generated in each sequence in all datasets, creating new datasets which will be referred to as Random. Using the three Random datasets allowed us to test the accuracy of hCSC and saCSC; notice that nCSC and saCSC always return the same rotation. For each Random dataset, an all-against-all sequence comparison was performed. That is, all 66 possible pairs of sequences in each dataset were given as input to both hCSC and saCSC. β was set to ⌈√m⌉ = 50 and q was set to ⌈log₂(Σ) m⌉ = 6. The resultant re-rotated sequences were aligned using EMBOSS Needle (default parameters) and the similarity scores were compared to those of the Original and Random datasets, which were input directly to EMBOSS Needle (default parameters). The results can be found in Fig. 1.

The results show that: (a) hCSC and saCSC yield significantly improved similarity scores compared to those obtained from giving Random datasets as input directly to EMBOSS Needle; and (b) hCSC and saCSC yield similarity scores that are identical or almost identical—notice that the black (Original), green (hCSC), and blue (nCSC/saCSC) points coincide—to those obtained from giving Original datasets as input directly to EMBOSS Needle. This implies that algorithms hCSC, nCSC, and saCSC return the rotation maximizing the similarity score for all pairwise comparisons.

Hence, we establish here that the introduced distance measure coupled with the respective algorithms consistently yield a very high accuracy, compared to the standard measure [17, 39, 40], for both low and high substitution rates.

Time performance

We then compared the time performance of the algorithms. Each algorithm was given a pair of randomly generated sequences starting from m = n = 50 bp and doubling 8 times to a length of m = n = 12,800 bp. It was expected that the slowest algorithm would be cNW which runs in time O(mn²). Then it would be algorithm nCSC which runs in time O(m(m + n)), then algorithm hCSC, which runs in time O(β(m + n) + m(m+n) β), and lastly algorithm saCSC, which runs in time O(βm + n).

Initially, β was set to ⌈√m⌉ and q was set to ⌈log₂(Σ) m⌉. The results in Fig. 2 demonstrate orders-of-magnitude superiority of saCSC compared to cNW and nCSC, confirming our theoretical findings. Algorithm hCSC is the second fastest. Although β was set to ⌈√m⌉, saCSC clearly outperforms hCSC, due to the use of a highly optimized implementation of the suffix-array construction [43], thus highlighting the importance of suitably implemented data structures such as suffix arrays.

Since the time complexities of hCSC and saCSC depend on β, we repeated the same experiment with these two algorithms setting β to ⌈m/25⌉ and q to ⌈log₂(Σ) m⌉—
notice that $q$ does not affect the time efficiency of the algorithms. The results in Fig. 2 show that hCSC and saCSC are still the fastest, even though $m = O(\beta)$, and that saCSC is clearly the fastest of all. As expected for $m = O(\beta)$, we observe that hCSC and saCSC become gradually slower as $m$ grows.

More algorithms could have been included in the comparison but their (at least) quadratic time complexity [22, 23] prevents them from competing with saCSC.

### Application to synthetic data

For evaluating the proposed methods for circular sequence comparison in some relevant application, we also implemented the following pipeline for distance-based phylogenetic reconstruction of a dataset with $N$ circular sequences.

1. For each pair $(x, y)$ of the $N$ sequences, we use one method for circular sequence comparison to compute the best rotation $x'$.
2. A similarity score for $(x', y)$ is then computed using EMBOSS Needle (default parameters) and stored in cell $[x, y]$ of an $N \times N$ similarity score matrix.
3. The similarity score matrix is transformed into a distance matrix by converting each score into a distance relative to the maximum score in the similarity score matrix.
4. Neighbour joining clustering is performed on the distance matrix, using NINJA [44], to produce a phylogenetic tree.

Phylogenetic trees were constructed by NINJA [44], for the aforementioned Random datasets, using output from the following algorithms: cNW (EMBOSS default parameters), hSW (see introduction of “Implementation” section, EMBOSS default parameters), and saCSCr ($\beta = 50$, $q = 5$, and $p = 1$). Notice that, output from cNW should be the same as from EMBOSS Needle (default parameters) with the Original datasets as input. In terms of accuracy, the Robinson-Foulds (RF) distance metric [45, 46] was used to compare the three resultant phylogenetic trees with the tree resulting from EMBOSS Needle (default parameters) on the Original and Random datasets, denoted by NW(o) and NW(r), respectively. RF distance can be defined as the number of operations required to transform one tree in to another. If two isomorphic trees share the same labelling then they have an RF distance of 0. The results displayed in Table 3 clearly show that saCSCr and cNW produce the most accurate results with these nine datasets. As also shown in [11], hSW followed by EMBOSS Needle (default parameters) can often result in sub-optimal global alignments.

In terms of time performance, the elapsed time required for each method to process each dataset was recorded and the results are displayed in Table 4. It is clear, from the results presented heretofore, that saCSCr outperforms all other algorithms by at least one order of magnitude.

### Application to real data

We have concluded thus far that using $\beta = \lceil \sqrt{m} \rceil$ and $q = \lceil \log_2 |\Sigma| \rceil$ results in a reasonable trade-off between running time and accuracy. In the following section, where necessary, we adopt these values and multiply or divide them by a constant factor (factor of two), depending on the length of the input sequences.

### DNA sequences

**Pairwise sequence comparison.** As the input data set, we used two real sequences from GenBank: human

![Fig. 2 Time performance. Elapsed-time comparison of algorithms cNW, nCSC, hCSC, and saCSC](image-url)

Table 3 RF distances between the tree obtained from the NW(o) and those obtained from NW(r), cNW, hSW, and saCSCr

<table>
<thead>
<tr>
<th>Dataset $&lt;x, y, \theta, \kappa, \omega&gt;$</th>
<th>NW(r)</th>
<th>cNW</th>
<th>hSW</th>
<th>saCSCr</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;12, 2500, 0.05, 0.06, 0.04&gt;</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;12, 2500, 0.20, 0.06, 0.04&gt;</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;12, 2500, 0.35, 0.06, 0.04&gt;</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;25, 2500, 0.05, 0.06, 0.04&gt;</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;25, 2500, 0.20, 0.06, 0.04&gt;</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;25, 2500, 0.35, 0.06, 0.04&gt;</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;50, 2500, 0.05, 0.06, 0.04&gt;</td>
<td>86</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>&lt;50, 2500, 0.20, 0.06, 0.04&gt;</td>
<td>84</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;50, 2500, 0.35, 0.06, 0.04&gt;</td>
<td>56</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The number of sequences in the dataset is denoted by $x$; $y$ denotes their lengths; $\theta$ denotes the substitution rate; $\kappa$ and $\omega$ denote the relative insertion and deletion rates, respectively.
(NC_001807) and chimpanzee (NC_001643) MtDNA sequences. The MtDNA genome size for human is 16,571 bp and for chimpanzee is 16,554 bp. Their pairwise sequence alignment using EMBOSS Needle (default parameters) gives a similarity of 85.1%. We used cNW (EMBOSS default parameters) to obtain the rotation of NC_001807 that maximizes its similarity score with NC_001643. This experiment took approximately 28 h and the resultant rotation 578 of NC_001807 improved the similarity score to 91%. This result was then compared to those obtained from saCSCr (equivalent to saCScr with \( p = 0 \)) and saCScr with varying parameters, displayed in Table 5.

The convergence of the results after the additional step of refinement (see Table 5 in italics) demonstrates the convenience and necessity of saCScr.

For clarity of presentation hereafter, instead of using \( \beta \), we denote by \( \ell \) the length of the block chosen in algorithm saCScr.

We repeated this experiment with the human and gorilla (NC_011120) MtDNA sequences. The MtDNA genome size for gorilla is 16,412 bp. Their pairwise sequence alignment using EMBOSS Needle (default parameters) gives a similarity of 83.5%. After using saCScr to rotate sequence NC_001807 (\( \ell = 50, q = 5 \), and \( p = 1 \)), EMBOSS Needle (default parameters) gave a significantly improved similarity of 88.4%.

Finally, note that the experiments which used saCSCr and saCScr each took a fraction of a second to run.

### Distance-based phylogenetic reconstruction

Three datasets of 16 primate, 12 mammalian and 19 mixed mammalian and primate MtDNA sequences, of average length 16,500 bp, were obtained from GenBank. We followed the same pipeline as described in "Application to synthetic data" section. The RF distance between the trees produced by cNW (EMBOSS default parameters), and the trees produced by saCScr (\( \ell = \lceil \sqrt{m} \rceil = 129, q = 5 \), and \( p = 1 \)) followed by EMBOSS Needle (default parameters), was 0.

#### RNA sequences

Eighteen viroid sequences were obtained from RefSeq, a database of curated molecular biological sequences [47]. Their lengths and target hosts vary, ranging from 348 to 371 bp and infecting peppers and citrus fruits, respectively. We followed the same pipeline as described in "Application to synthetic data" section. The RF distance between the tree produced by cNW (EMBOSS default parameters), and the tree produced by saCScr (\( \ell = \lceil \sqrt{m} \rceil = 19, q = \lceil \log_{10} m \rceil = 5 \), and \( p = 1 \)) followed by EMBOSS Needle (default parameters), was 0.

#### Protein sequences

**Linear, circularly-permuted protein sequences**

Eight sequences of proteins, of average length 950 amino acids, belonging to \( \beta \)-glucosidase family [48] were obtained from the UniProt protein database [49]. We followed the same pipeline as described in "Application to synthetic data" section. The RF distance between the tree produced by cNW (EMBOSS default parameters), and the tree produced by saCScr (\( \ell = \lceil \sqrt{m} \rceil = 31, q = \lceil \log_{10} m \rceil = 5 \), and \( p = 1 \)) followed by EMBOSS Needle (default parameters), was 0.

**Naturally-occurring circular proteins**

Ten bacteriocin protein sequences, of average length 20 amino acids, were obtained from Cybase [50], a database of cyclical protein sequences. We followed the same pipeline as described in "Application to synthetic data" section. The RF distance between the tree produced by cNW (EMBOSS default parameters), and the tree produced by saCScr (\( \ell = 2\lceil \sqrt{m} \rceil = 10, q = 2\lceil \log_{10} m \rceil = 6 \), and \( p = 1 \)) followed by EMBOSS Needle (default parameters), was 0.

### Table 4 Elapsed-time comparison (in seconds) for algorithms cNW, hSW, and saCScr

<table>
<thead>
<tr>
<th>Dataset &lt; ( \alpha, \gamma, \theta, \kappa, \omega &gt; )</th>
<th>cNW</th>
<th>hSW</th>
<th>saCScr</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;12, 2500, 0.05, 0.06, 0.04&gt;</td>
<td>10,139.36</td>
<td>72.43</td>
<td>6.90</td>
</tr>
<tr>
<td>&lt;12, 2500, 0.20, 0.06, 0.04&gt;</td>
<td>9888.84</td>
<td>80.91</td>
<td>6.57</td>
</tr>
<tr>
<td>&lt;12, 2500, 0.35, 0.06, 0.04&gt;</td>
<td>10,052.33</td>
<td>80.16</td>
<td>6.28</td>
</tr>
<tr>
<td>&lt;25, 2500, 0.05, 0.06, 0.04&gt;</td>
<td>46,311.85</td>
<td>369.02</td>
<td>27.61</td>
</tr>
<tr>
<td>&lt;25, 2500, 0.20, 0.06, 0.04&gt;</td>
<td>46,230.07</td>
<td>375.41</td>
<td>28.92</td>
</tr>
<tr>
<td>&lt;25, 2500, 0.35, 0.06, 0.04&gt;</td>
<td>46,289.99</td>
<td>400.30</td>
<td>30.44</td>
</tr>
<tr>
<td>&lt;50, 2500, 0.05, 0.06, 0.04&gt;</td>
<td>122,165.95</td>
<td>1563.96</td>
<td>125.63</td>
</tr>
<tr>
<td>&lt;50, 2500, 0.20, 0.06, 0.04&gt;</td>
<td>121,810.69</td>
<td>1617.89</td>
<td>123.12</td>
</tr>
<tr>
<td>&lt;50, 2500, 0.35, 0.06, 0.04&gt;</td>
<td>120,679.32</td>
<td>1662.82</td>
<td>123.77</td>
</tr>
</tbody>
</table>

The number of sequences in the dataset is denoted by \( \alpha \); \( \gamma \) denotes their lengths; \( \theta \) denotes the substitution rate; \( \kappa \) and \( \omega \) denote the relative insertion and deletion rates, respectively.

### Table 5 Rotations of GenBank sequence NC_001807 obtained when compared to NC_001643 with varying parameters of saCScr

<table>
<thead>
<tr>
<th>( q )</th>
<th>( \beta )</th>
<th>( p )</th>
<th>Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>50</td>
<td>0</td>
<td>566</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>1</td>
<td>578</td>
</tr>
<tr>
<td>5</td>
<td>( \sqrt{m} )</td>
<td>0</td>
<td>567</td>
</tr>
<tr>
<td>5</td>
<td>( \sqrt{m} )</td>
<td>1</td>
<td>578</td>
</tr>
<tr>
<td>5</td>
<td>( 2\sqrt{m} )</td>
<td>0</td>
<td>583</td>
</tr>
<tr>
<td>5</td>
<td>( 2\sqrt{m} )</td>
<td>1</td>
<td>578</td>
</tr>
<tr>
<td>5</td>
<td>( \sqrt{m} )</td>
<td>0</td>
<td>566</td>
</tr>
<tr>
<td>5</td>
<td>( \sqrt{m} )</td>
<td>1</td>
<td>578</td>
</tr>
</tbody>
</table>
Conclusions
In this paper, we introduced a new distance measure for sequence comparison based on \(q\)-grams, and showed how it can be applied effectively and computed efficiently for circular sequence comparison. The most efficient algorithm presented here, saCSC\(_r\), solves our defined problem CSC, exactly. Extensive experimental results, using both real and synthetic data, show that it maintains an accuracy very competitive to the optimal obtained after considering all rotations of \(x\) against \(y\) naively using global alignments. We also showed that algorithm saCSC\(_r\) can bridge the gap between the optimal solution and our approximation via an additional refinement step. Finally, the presented experimental study demonstrates orders-of-magnitude superiority of our approach in terms of runtime efficiency. Our immediate target is to implement algorithm saCSC\(_r\) in BEAR [24], a state-of-the-art tool for improving multiple circular sequence alignment.

Authors' contributions
RG, CSL, RM, NP, and SPP devised the algorithms. SPP, AR, and FV implemented the algorithms. AR and FV conducted the experiments. All authors contributed equally in writing up the manuscript. All authors read and approved the final manuscript.

Author details
1 Department of Informatics, University of Pisa, Pisa, Italy. 2 ERABLE team, INRA, Paris, France. 3 Department of Informatics, King’s College London, London, UK. 4 Department of Computer Science, Kiel University, Kiel, Germany.

Acknowledgements
The publication costs for this article were funded by the Open Access funding scheme of King’s College London. RG and NP have been partially supported by the Italian Ministry of Education, Universities, and Research (MIUR) under PRIN 2012CiE3XT national research project AMANDA—Algorithms for Massive and Networked Data, and by the University of Pisa under FRA 2015 project Computational Methods for Personalized Medicine. RM is supported by the PRI.ME programme of DAAD co-funded by BMFB and the EU’s 7th Framework Programme (No65728), and the Newton International Fellowship co-funded by the Royal Society and the British Academy. FV is supported by an EPSRC Grant (Doctoral Training Grant #EP/M506357/1).

Competing interests
The authors declare that they have no competing interests.

Received: 8 December 2015 Accepted: 25 April 2016
Published online: 10 May 2016

References


