Evolutionary Search Techniques for the Lyndon Factorization of Biosequences

Amanda Clare
Aberystwyth University
Aberystwyth, UK
afc@aber.ac.uk

Jacqueline W. Daykin*†
Aberystwyth University
Aberystwyth, UK
jwd6@aber.ac.uk

Thomas Mills
Aberystwyth University
Aberystwyth, UK
Thomasandrewmills@outlook.com

Christine Zarges
Aberystwyth University
Aberystwyth, UK
c.zarges@aber.ac.uk

ABSTRACT

A non-empty string $\boldsymbol{x}$ over an ordered alphabet is said to be a Lyndon word if it is alphabetically smaller than all of its cyclic rotations. Any string can be uniquely factored into Lyndon words and efficient algorithms exist to perform the factorization process in linear time and constant space. Lyndon words find wide-ranging applications including string matching and pattern inference in bioinformatics. Here we investigate the impact of permuting the alphabet ordering on the resulting factorization and demonstrate significant variations in the numbers of factors obtained. We also propose an evolutionary algorithm to find optimal orderings of the alphabet to enhance this factorization process and illustrate the impact of different operators. The flexibility of such an approach is illustrated by our use of five fitness functions which produce different factorizations suitable for different downstream tasks.

CCS CONCEPTS

• **Mathematics of computing → Discrete mathematics**; • **Theory of computation → Theory and algorithms for application domains**;

KEYWORDS

algorithm, alphabet, artificial intelligence, Burrows-Wheeler transform, factorization, evolutionary search, genome, Lyndon word, pattern matching, string, word

*Also with King's College London.
†Also with Stellenbosch University.

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326872

ACM Reference Format:
Amanda Clare, Jacqueline W. Daykin, Thomas Mills, and Christine Zarges. 2019. Evolutionary Search Techniques for the Lyndon Factorization of Biosequences. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3319619.3326872

1 INTRODUCTION

A *string* or *word* is a finite sequence of symbols over an alphabet which is typically ordered. This paper addresses structures in strings and permutations of the string alphabet with application to factoring genomes for sequence alignment. Useful structures in strings include borders (such as in the English word *eraser*), repetitions (for instance in the DNA strand *GGAATGGAATGGAAT*) and palindromes (English examples include *kayak*, *racecar*). The structure of interest here is known as Lyndon - a word is said to have the Lyndon property if it is least alphabetically amongst all cyclic rotations of the letters, common English examples: *amazing*, *chicken* and *moon*.

Novel techniques are presented for manipulating or permuting the order of an alphabet so as to enhance the Lyndon factorization of a string, that is its decomposition into Lyndon words. Wide ranging applications of Lyndon words arise in digital geometry [4], musicology [5], string matching [9], and bioinformatics. For instance, Lyndon words have been applied in STAR [10], an algorithm to search for tandem approximate repeats, to exclude motifs (significant nucleotide or amino-acid patterns) that form necklace rotations. Motifs of tandem repeats are known to account for large portions of eukaryotic genomes and other life kingdoms.

Bioinformatics applications have very specific finite alphabets, namely cardinality 4 for DNA & RNA and cardinality 20 for protein. Furthermore, bioinformatics often involves huge volumes of data – the human genome contains around 3 billion base pairs residing in 23 pairs of chromosomes (structures of nucleic acids and protein) within the nucleus of cells. Sequence factorization facilitates useful approaches such as parallelism and block compression. Hence, we are interested in considering factorization techniques for specific application domains pertinent to biology while noting that the results are not restricted to this domain.

The focus of this paper is on how the manipulation of an alphabet ordering can have a considerable impact on the resulting Lyndon

factorization of a string, specifically the number of factors. Towards the goal of computing optimal alphabet orderings, we propose an evolutionary search technique to search for such alphabet orderings to enhance this factorization process. We explore different fitness functions to maximize, minimize, parameterize, and balance the length of the factors.

Experimental results given in Section 4 indicate that this is a promising line of enquiry and approach, while future research directions are proposed in Section 5.

1.1 Related Work

A *permutation* of length n is an ordering of the set $[1, n] = \{1, 2, \ldots, n\}$ of integers, a *permutation pattern* is a sub-permutation of a longer permutation, and a *permutation class* is a set C of permutations such that every pattern within a permutation in C is also in C. For clarification, given the permutation $\pi = 642351$ then 312 is a pattern as there are subsequences of π, such as 423, 625, 635, with the same relative ordering. An *interval* of a permutation is a consecutive substring consisting of consecutive symbols - for instance, 8628 is an interval in the permutation 886283418. Such intervals have real significance in genomic sequencing problems namely the matching of DNA sequences, reads, produced by high throughput sequencing to a reference sequence with the goal of variation calling. Also, genomes can be modelled as a permutation of genes, a common interval is then a set of orthologous genes that appear consecutively, possibly in different orders, in two genomes. Common intervals thus provide a measure of genes associated by function [8].

A notable permutation is the Burrows-Wheeler transform (BWT) which rearranges an input string in such a way that tends to cluster the data while also being invertible allowing recovery of the data - both the transformation and inverse can be computed efficiently in linear time. Formally, the BWT of $\boldsymbol{x}$ is defined as the pair (L, h) where L is the last column of the matrix $M_{\boldsymbol{x}}$ formed by all the lexicographically sorted cyclic rotations of $\boldsymbol{x}$ and h is the index of $\boldsymbol{x}$ in this matrix. The lexicographic nature and data clustering properties render the transform applicable to indexing techniques and numerous compression scenarios: lossless, test data, suffix trees & arrays, and image compression [1]. Indeed, the BWT is at the core of bzip2, a standard tool for lossless compression.

Furthermore, the BWT string permutation has been applied in bioinformatics applications including the highly successful (in terms of both implementations and citations) Bowtie sequence alignment program [14]. Exact pattern matching is insufficient for short read alignment because alignments may contain mismatches – these mismatches may be due to errors arising during the molecular sequencing process, genuine differences between the reference genome and query organisms, or both. Bowtie addresses this issue by conducting a backtracking search to find candidate alignments that satisfy a quality-based value, which for multiple candidates follows a greedy approach. In order to limit excessive backtracking Bowtie introduces the technique of double indexing of the reference genome – the BWT (forward index) and co-BWT (mirror index). Alignment attempts are restricted to the left and right half of the query along with the appropriate index.

For many string-based applications, it is useful to capture properties of a string to facilitate an algorithm. One example used in this work is the *Parikh vector* (also known as a permuted string or permuted pattern), $p(\boldsymbol{v})$, of a finite word $\boldsymbol{v}$ which enumerates the occurrences of each letter of the alphabet in $\boldsymbol{v}$. Two strings are considered Abelian equivalent if one can be turned into the other by permuting its letters; in other words, if the two strings have the same Parikh vector.

Applications of permutations are far reaching and include jumbled pattern matching which is a special case of approximate pattern matching: searching for an occurrence of a jumbled version of a query string $\boldsymbol{p}$ in a text $\boldsymbol{t}$, i.e. searching for a substring $\boldsymbol{p_0}$ which has the same Parikh vector as $\boldsymbol{p}$. Our application of Parikh vectors is in order-based string factoring methods.

2 PRELIMINARIES

2.1 Notation

Given an integer $n \geq 1$ and a nonempty set of symbols Σ (bounded or unbounded), a ***string of length n***, equivalently ***word***, over Σ takes the form $\boldsymbol{x} = x_1 \ldots x_n$ with each $x_i \in \Sigma$. For brevity, we write $\boldsymbol{x} = \boldsymbol{x}[1..n]$ with $\boldsymbol{x}[i] = x_i$. The length n of a string $\boldsymbol{x}$ is denoted by $|\boldsymbol{x}|$. The set Σ is called an ***alphabet*** whose members are ***letters*** or ***characters***, and Σ^+ denotes the set of all nonempty finite strings over Σ. The ***empty string*** of length zero is denoted $\boldsymbol{\varepsilon}$; we write $\Sigma^* = \Sigma^+ \cup \{\boldsymbol{\varepsilon}\}$ and let $|\Sigma| = \sigma$. We use exponents to denote repetition, for instance if $\alpha \in \Sigma$ then α^3 means $\alpha\alpha\alpha$. If $\boldsymbol{x} = \boldsymbol{uwv}$ for strings $\boldsymbol{u, w, v} \in \Sigma^*$, then $\boldsymbol{u}$ is a ***prefix***, $\boldsymbol{w}$ is a ***substring*** or ***factor***, and $\boldsymbol{v}$ is a ***suffix*** of $\boldsymbol{x}$; we say $\boldsymbol{u} \neq \boldsymbol{x}$ is a ***proper prefix*** and similarly for the other terms. If $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is said to be a ***rotation*** (***cyclic shift*** or ***conjugate***) of $\boldsymbol{x}$; for example, the anti-clockwise single letter rotations of the string *abca* are *abca*, *aabc*, *caab*, *bcaa*. A string $\boldsymbol{x}$ is said to be a ***repetition*** if and only if it has a factorization $\boldsymbol{x} = \boldsymbol{u}^k$ for some integer $k > 1$; otherwise, $\boldsymbol{x}$ is said to be ***primitive***. Observe that every rotation of a repetition is also a repetition. For a string $\boldsymbol{x}$, the reversed string $\overline{\boldsymbol{x}}$ is defined as $\overline{\boldsymbol{x}} = \boldsymbol{x}[n]\boldsymbol{x}[n-1]\cdots\boldsymbol{x}[1]$. A string which is both a proper prefix and a proper suffix of a string $\boldsymbol{x} \neq \boldsymbol{\varepsilon}$ is called a ***border*** of $\boldsymbol{x}$; a string is ***border-free*** if the only border it has is the empty string $\boldsymbol{\varepsilon}$. We write strings in mathbold such as $\boldsymbol{w}$.

The ***Parikh vector*** $p(\boldsymbol{x})$ of a string $\boldsymbol{x}$ over a finite ordered alphabet $\Sigma = \{\lambda_1, \ldots, \lambda_\sigma\}$ is defined as the vector of multiplicities of the characters, $p(\boldsymbol{x}) = (p_1, \ldots, p_\sigma)$, where $p_i = |\{j \mid x_j = \lambda_i\}|$.

If Σ is a totally ordered alphabet then ***lexicographic ordering*** (***lexorder***) $\boldsymbol{u} < \boldsymbol{v}$ with $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^+$ means that either $\boldsymbol{u}$ is a proper prefix of $\boldsymbol{v}$, or $\boldsymbol{u} = \boldsymbol{ras}$, $\boldsymbol{v} = \boldsymbol{rbt}$ for some $a, b \in \Sigma$ such that $a < b$ and for some $\boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t} \in \Sigma^*$. Using the Roman alphabet: *ant* < *bee* < *horse* < *horses* < *zoo* < *zoology* - and we see it is the alphabetical order used in the English dictionary.

A fundamental application of lexorder in the fields of algebra, combinatorics on words, and stringology yields the concept of Lyndon words: a string/word is ***Lyndon*** if and only if it is the strictly least in lexorder amongst all its cyclic rotations, that is conjugacy class. For example, consider the lexordered conjugacy class of the string *abac*: *abac* < *acab* < *baca* < *caba*, then *abac* is a Lyndon word as it is strictly least while no other rotation can be a

Lyndon word. Lyndon words are necessarily primitive and border-free. Remarkably, any string $\boldsymbol{x}$ exhibits a Lyndon factorization $\mathrm{LF}_{\boldsymbol{x}}$:

Theorem 2.1. *[6] Any word $\boldsymbol{x}$ can be written uniquely as a non-increasing product $LF_{\boldsymbol{x}} = \boldsymbol{x} = \boldsymbol{u}_1\boldsymbol{u}_2 \cdots \boldsymbol{u}_k$ of Lyndon words.*

Theorem 2.1 shows that there is a unique decomposition of any word into non-increasing Lyndon words ($\boldsymbol{u}_1 \geq \boldsymbol{u}_2 \geq \cdots \geq \boldsymbol{u}_k$) which allows for divide & conquer type applications. Thus we feel that the resulting number of factors in the factorization is significant. Since a string factorization over a specified alphabet Σ is unique and maximal, in the sense that no factor $\boldsymbol{f}$ can be extended to the left or right without losing the Lyndon criteria of $\boldsymbol{f}$, it follows that the number of factors in the factorization will be fixed – this point is relevant to the termination of search in Algorithm 1 – accordingly we seek enhancement of the number of factors via permutation of Σ. Note therefore that our solutions will be depicted as permuted vectors over the Roman alphabet.

2.2 Alphabet ordering

To motivate our interest in alphabet ordering (in the case of an unordered alphabet) or permuting an (ordered) alphabet, consider the following example of a Lyndon factorization of a string $\boldsymbol{x}$ of length n on a binary alphabet $\{0, 1\}$. Let $\boldsymbol{x} = 01^j0^21^{j-1} \cdots 0^j1, j > 1$. Then the number of factors varies from $O(1)$ to $O(j)$; specifically, for $\{1 < 0\}$ the number of factors in the factorization is 3, whereas if $\{0 < 1\}$ the number of factors is j.

In recent research, Clare and Daykin [7] proposed a greedy alphabet ordering algorithm for enhancing a Lyndon factorization. They mainly considered minimizing the number of factors while observing that the algorithm could be modified to maximize. For the heuristic they introduced a variant of the Parikh vector, the Exponent Parikh vector which, for each distinct letter in a string records its individual RLE (run length encoding) exponent pattern (left-to-right sequence of exponents for a letter) - so the sum of these exponents is the Parikh entry for that letter. The Exponent Parikh vector was then used to start the alphabet ordering with a selection of a least letter; subsequently, if a cycle order was generated the algorithm would backtrack using the Exponent Parikh vector to start again. Experimentation using prokaryotic reference genomes over the biological DNA alphabet $\{A, C, G, T\}$ demonstrated encouraging, though not optimal, performance of the algorithm.

In a quest to improve the factorization process, methods from evolutionary computation were introduced.

3 EVOLUTIONARY ALGORITHM

With different sized alphabets and no general pattern of characters running in a sequence, the problem of reordering (permuting) the alphabet to enhance a Lyndon factorization can be hard to solve. While for DNA and RNA alphabets it is computationally feasible to perform a brute force search to find the optimal ordering, this is not possible for larger alphabets such as the 20 amino acids that form protein strings or the 26 letters of the English alphabet – compare $4! = 24$ to $26! = 4.0329146e + 26$.

Evolutionary algorithms have been successfully used in bioinformatics [13] and for permutation-based search problems [17]. They also bring an advantage in that they can be stopped at any time, therefore can be adapted to the complexities of the different ways we can optimize the alphabet orderings. We therefore propose an evolutionary algorithm to find optimal orderings of the alphabet to further enhance a Lyndon factorization. We explore different fitness functions to

- maximize the number of factors,
- minimize the number of factors,
- find a specific number of factors, and
- balance the length of the factors.

- Maximal Factorization ($a < b < c < d$):
  $(b)(acdbd)(abbcdbbddbdbd)(abbacb)(abacbc)$
- Minimal Factorization ($a < c < d < b$):
  $(b)(acdbdabbcdbbddbdbdabbacbabacbc)$
- Balanced Factorization ($b < a < c < d$):
  $(bacdbda)(bbcdbbddbdbda)(bbacbabacbc)$

Figure 1: Example bacdbdabbcdbbddbdbdabbacbabacbc: Lyndon factorizations for different alphabet orderings with $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$

Figure 1 shows the effect of maximization, minimization and balancing the Lyndon factorization of an example string using the alphabet $\Sigma = \{a, b, c, d\}$.

Our algorithm (Algorithm 1) starts by initializing a population from random permutations of the original alphabet ordering. In addition, one individual in the initial population is created using a simple heuristic: We order the alphabet by first appearance in the input string.

The algorithm then enters the main loop for a specified number of generations unless an additional exit criterion is met. In the case of minimization the exit criterion is met if the input string forms a Lyndon word (a single factor). In the event of optimizing to a specific number of factors, the exit criterion is met when one of the solutions in the population has the required number of factors. For maximizing the number of factors and balancing the lengths of the factors there is no exit criterion and the algorithm escapes once the main loop has been executed for the specified number of generations.

The main loop consists of firstly evaluating every solution in the population using one of the proposed fitness functions (Section 3.1). Half of the population is then discarded during the semi-proportional selection method (Section 3.2). Finally, the population is repopulated by means of crossover (Section 3.3.2) and mutation (Section 3.3.1).

Algorithm 1: Evolutionary Search

```
1. Initialization of the Population;
while Exit Criteria Not Met do
    2. Evaluate Alphabet Orderings;
    3. Semi-Proportional Selection;
    4. Create offspring using crossover and mutation;
end
```

3.1 Fitness Functions

An alphabet $\Sigma = \{\lambda_1, \ldots, \lambda_\sigma\}$ is given. The objective is to find a permutation π^* of the alphabet that optimizes the given fitness function.

We consider four types of optimization problems and five different fitness functions $f : S_\sigma \to \mathbb{R}$ where S_σ denotes the permutation space over the alphabet Σ. All fitness functions use Duval's [11] linear time and constant space algorithm to compute the number of factors. However, an extension has been made in which the lexicographical ordering of strings over a given ordered alphabet can be modified by reordering the alphabet.

Let $i_{w,\pi}$ denote the number of Lyndon factors for a word $w \in \Sigma^*$ given an alphabet ordering π.

(1) For minimizing the number of factors the fitness score of the fitness function is simply minimizing the number of factors, i. e., we minimize $f(\pi) = i_{w,\pi}$.
(2) The fitness function is 'reversed' for maximization, where a higher number of factors is more favourable, i. e., we maximize $f(\pi) = i_{w,\pi}$.
(3) To calculate the fitness value when balancing the lengths of factors, Duval's algorithm is computed on the input string and the length of every factor is recorded. Let ℓ_j denote the length of the j-th factor. We then define two different fitness functions:
   (a) The fitness value is the standard deviation of the factor lengths, i. e.,
   $$f(\pi) = \sqrt{\frac{\sum_{j=1}^{i_{w,\pi}} (\mu - \ell_j)^2}{i_{w,\pi}}},$$
   where μ denotes the mean length.
   (b) The fitness value is the difference between the maximum and the minimum length, i. e.,
   $$f(\pi) = \max_{1 \le j \le i_{w,\pi}} \ell_j - \min_{1 \le j \le i_{w,\pi}} \ell_j$$
(4) When searching for a specific number of factors k, that is a parameterized form of the problem, the fitness function simply minimizes the absolute difference between the actual number of factors and the desired number of factors, i. e., we minimize $f(\pi) = |k - i_{w,\pi}|$. Note that it might not be possible to achieve the number of factors specified. For instance, given the input string $\boldsymbol{x} = {\lambda_i}^n$, then the desired parameter $n/2$ is generally unobtainable as the factorization of $\boldsymbol{x}$ has n factors over any alphabet ordering. Thus, we define an optimal solution as the closest number of factors possible.

3.2 Semi-Proportional Selection

At every generation, solutions need to be selected as parents in order to create the next generation of offspring. Accordingly, to allow for exploration and exploitation, it is important to both balance the selection pressure and preserve a sufficient degree of diversity in the population: Too high selection pressure (i. e., selection of only the best solutions) will likely result in loss of diversity while too low selection pressure (i. e., not biasing the selection towards better solutions) will lead to stagnation of the search as we are more likely to lose the current best solutions.

Preliminary tests with a ranking-based fitness-proportional selection method also led to a loss of diversity. We therefore decided to select parents uniformly at random, but only from the top-ranked half of the current population. Moreover, the lower ranked half of the population is discarded and replaced by the offspring created in the next step.

3.3 Creation of Offspring

As discussed above in Section 3.2, two parents are selected at random from the top 50% of the population and are then used to create an offspring. The selected parents are assured to be unique to allow more variation into the population. Once a child has been created using crossover (see Section 3.3.2), it is passed to the mutation operator. Depending on the mutation rate, the generated child may then be mutated by the mutation operator explained in Section 3.3.1. The child is added to the population and the process is repeated until the population is back to its original size. The algorithm will repeat the evaluation process until either the maximum number of generations is met, or an optimal solution is found (in case of minimization and specific).

3.3.1 Mutation Operators. Our evolutionary approach for the Lyndon factorization problem extends two generic permutation-based mutation operators: Swap Mutation and Insert Mutation [12]. Both mutation operators select two random elements. Swap Mutation then swaps the two elements. Similarly, Insert Mutation inserts one of the selected elements in the position next to the other and then shifts the other elements accordingly.

We observe that changes in some positions have more impact than others. Consider the alphabet $\Sigma = \{a, b, c, d, e\}$ with the ordering $a < b < c < d < e$. The lowest ordered characters are the ones that are situated at the beginning of the permutation, for example the characters a and b are considered low ordered characters. Similarly, the highest ordered characters are the ones that are situated at the end of the permutation. After analyzing the output from computing the factorization on a series of strings, we observed that changes to the first ordered character made a dramatic impact on minimizing or maximizing the number of factors.

To apply these generic operators to the Lyndon factorization problem, both of these generic operators have been extended as discussed below. Pseudocode for the complete mutation operator is shown in Algorithm 2.

As discussed above, frequently selecting the lower ordered characters allows the algorithm to jump to different areas of the search space and escape a local optima. Thus, we bias the selection of the two elements to increase the probability that lower ordered elements are selected. To be more precise, with probability of at least 0.3 one of the two elements will be selected from the three lowest ordered elements in the alphabet.

Moreover, we randomly select one of the two mutation operators for each offspring. Using the heuristic that mutating the order of two elements (Swap Mutation) in the alphabet ordering is more likely to make a dramatic change on the resulting factors compared to only changing the order of a single element (Insert Mutation), the selection is biased towards the Insert Mutation operator, which is selected with probability 0.9. This allows us to mutate the orderings

enough to move out of local optima, but not so much that it takes us away from a good solution.

Algorithm 2: Mutation Operator

```
Select positions i, j ∈ [1, n] uniformly at random;
With probability 0.3 do
    Choose i from {1, 2, 3} uniformly at random
With probability 0.1 do
    swap elements at positions i and j
else
    move element at position j to position i + 1
    and shift accordingly
```

3.3.2 Crossover Operator. As the problem is permutation-based, it limits us to generic crossover operators for this kind of search space [12]. Furthermore, the observation made about dramatic changes due to lower ordered characters also prevents us from using crossover methods that do not aim at preserving large parts of the order of the elements in the two parents.

Partially Mapped Crossover (PMX) has proven to work well on permutation-based problems such as the Traveling Salesperson Problem [18] and has been selected as the crossover operator. The main idea is to select two random elements as split points and transfer the contents between the split points from the first parent directly to the child. Remaining characters are then inserted in the best order with respect to the second parent.

To be more precise: The elements between the two split points are called the '*swath*'. All elements within the swath are directly copied from the first parent to the child. A look-up is performed in the second parent to see which elements within the swath were not copied to the child – these characters are denoted as i, and the elements in the child that took the position of i are denoted as j. An attempt is then carried out to place each element i in the position occupied by j in the second parent. If there is an element present in the child at the same index of j in the second parent, we denote this element as k and attempt to place i in the position occupied by k in the second parent. The remaining elements are then entered in the same order of appearance in the second parent. By reversing the roles of the parents, we can create another child.

4 EXPERIMENTS

The algorithm was implemented using Java[1] and tested on two different types of sequences: random sequences (Section 4.1) and protein sequences from a bacterial genome (Section 4.2). Based on preliminary experiments, the parameters shown in Table 1 are used.

Parameter	Value
Generations	1000
Population Size	16
Mutation Rate	100%
Crossover	PMX

Table 1: Parameters

4.1 Results for Random Sequences

We created 10 random sequences of length 300 over an alphabet of size 20. We executed 100 independent runs of our algorithm on all sequences using the five fitness functions described in Section 3.1. Due to space restrictions we report the average fitness (over the 100 runs) of the best individual in the population (plus the standard deviation) against the number of generations for three representative sequences.

[1]Code can be found at: https://github.com/thomasamills/LyndonEvolve

Recall that the best possible fitness value for minimization is 1 and for specific is 0. However, in both cases these values are not necessarily possible for a given sequence and in general the optimal fitness value is unknown and may be different for different sequences.

4.1.1 Minimization. Figure 2 plots the fitness against the number of generations for the minimization fitness function. We see that in all three cases the best fitness value in the initial population is already quite good. We conjecture that this is due to our approach to create an initial ordering which seems to be a good heuristic for the minimization problem. It is worth noting that the fitness converged to 2 for all 10 random sequences and that at most 300 generations were needed to find a solution with two factors.

4.1.2 Maximization. Figure 2 plots the fitness against the number of generations for the maximization fitness function. We see that the maximization problems appears to be more difficult than the minimization problem as more generations are needed to yield considerable improvements. It is interesting to see that the maximum fitness reached during our 1000 generations is very similar across different sequences.[2]

4.1.3 Balanced. Figures 4 and 5 plot the fitness against the number of generations for the two balance fitness functions. Similarly to the maximization fitness function more generations are needed to yield considerable improvements. While the differences after 1000 generations still seem relatively large, it is unclear if better solutions could be obtained.

4.1.4 Specific=12. Finally, Figure 6 plots the fitness against the number of generations for the fitness functions trying to reach a specific number of factors. We see that this problem seems again easy: Less than 25 generations are needed to reach an optimal solution for all 10 sequences considered. Investigating how the difficulty depends on the chosen target number is subject to future research.

4.2 Results for Biosequences

The algorithm has also been tested on the 573 proteins from a bacterial genome (*Buchnera aphidicola*[3]), from the RefSeq Prokaryote Genomes reference collection [16].

We first factored each protein in the genome using lexicographical ordering, resulting in a total of 4,043 factors for the set of proteins, with a mean number of factors of 7.0 and standard deviation of 2.25. We then re-factored the data set using the evolutionary algorithm with each of the fitness functions discussed in Section 3.1.

[2]In fact it is very similar for all 10 random sequences.
[3]GCF_000009605.1_ASM960v1_protein.faa

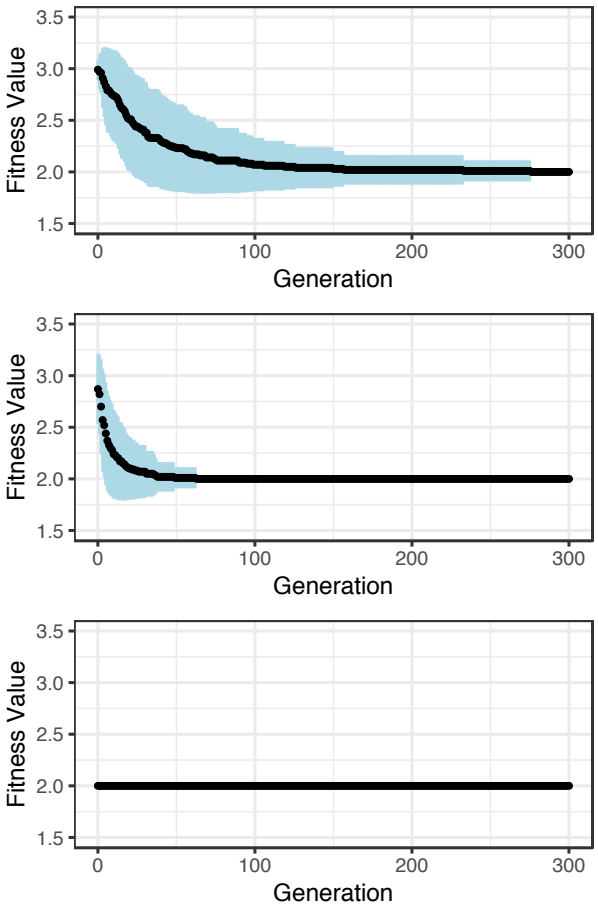

Figure 2: Minimization – Average fitness and standard deviation over 100 independent runs for three random sequences of length 300 over an alphabet of size 20.

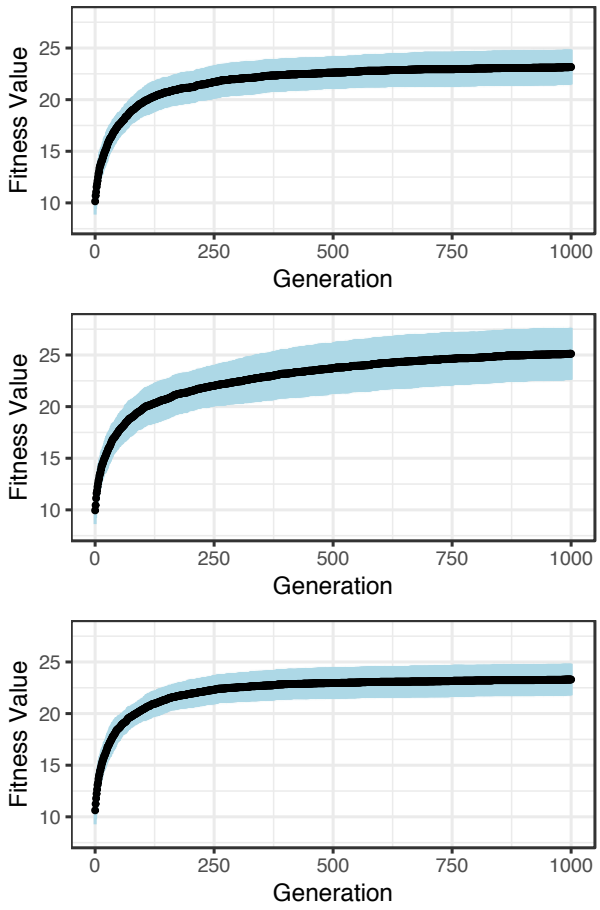

Figure 3: Maximization – Average fitness and standard deviation over 100 independent runs for three random sequences of length 300 over an alphabet of size 20.

The results are summarised in Figure 7. Interestingly, using the minimization operator, we can see that we can always find an alphabet reordering that produces at most two Lyndon factors, and in most cases, just one factor. This is potentially due to M (methionine) being the first letter of each sequence. M appears at the start of each sequence due to the AUG triplet of RNA bases used as the initiation site for translation of mRNA into each protein. However, methionine is also used throughout the proteins, not just at the start, so a minimal factoring must still take into account the rest of the sequence. The maximum number of factors appears to follow a normal distribution, with mean of 22.7. A balanced factorization gave a range of numbers of factors, from 2 to 31, depending on the protein. The specific factorization into 12 factors was achievable for every one of the 573 proteins.

5 CONCLUSION AND FUTURE WORK

We have presented an evolutionary approach for finding the optimal alphabet reordering prior to factoring a string into Lyndon words. This is a permutation problem where altering some positions has much more impact than others. As such we have discussed suitable operators to provide an effective algorithm, provided an implementation of the algorithm, and evaluated it on random sequences and proteins from a bacterial genome.

Proposed future research directions include the following:

- The output of the greedy alphabet ordering algorithm [7] may be a useful starting point from which to create one or more of the populations at the start of the search, even though this could be a local optimum.
- Investigate further mutation and crossover operators for this permutation-based problem (such as the best-order crossover and other permutation crossover operators [2, 3]). After observing that changes to the first few elements of the permutation can cause drastic changes to the fitness we would like to learn more about the landscape of solutions and the operators that can be effective in this landscape.
- The relationship between Lyndon factors and the suffix array data structure used in the BWT (and many other algorithms) has previously been highlighted [15]. The authors of that

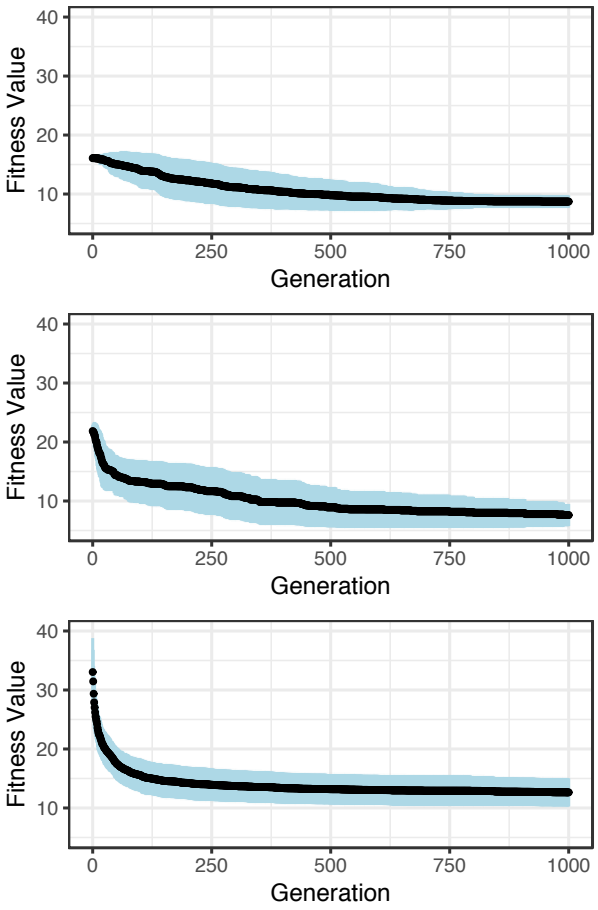

Figure 4: Balanced (standard deviation) – Average fitness and standard deviation over 100 independent runs for three random sequences of length 300 over an alphabet of size 20.

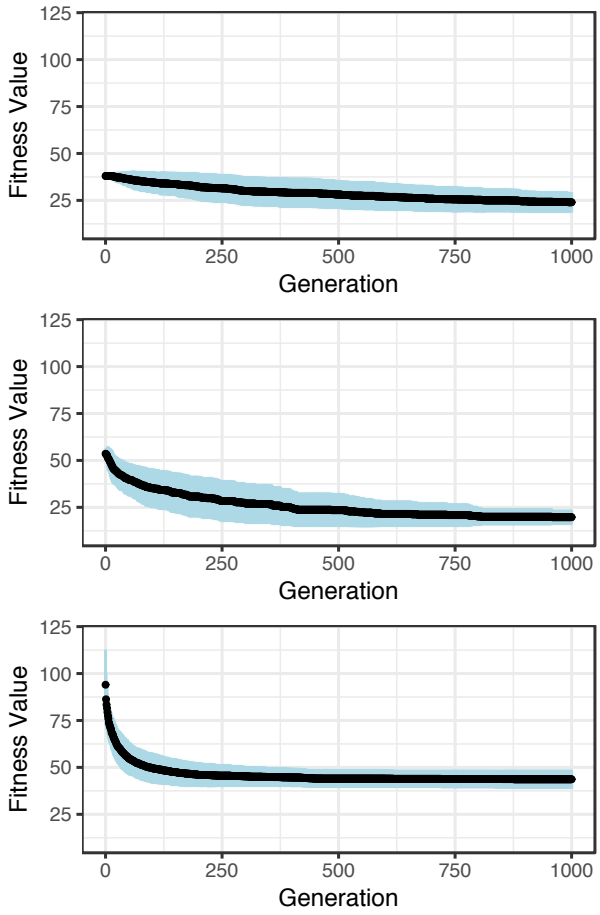

Figure 5: Balanced (difference) – Average fitness and standard deviation over 100 independent runs for three random sequences of length 300 over an alphabet of size 20.

work suggest that a suitable Lyndon factorization can help to partition the work of generation of the suffix array. We would therefore like to explore the optimal and near-optimal factorizations produced by this work for the construction of the suffix array and for partitioning a string prior to application of the BWT.

- Investigate further the alphabet orderings produced by a minimization of the number of factors, because they capture information about the protein sequences themselves. We would like to further understand whether the orderings themselves can be used as fingerprints for proteins, and whether there are orderings suitable for application to all proteins in a genome or even to all genomes in a genus.

REFERENCES

[1] D. Adjeroh, T. Bell, and A. Mukherjee. 2008. *The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching.* Springer Publishing Company. 352 pages.

[2] A. Andreica and C. Chira. 2015. Best-order crossover for permutation-based evolutionary algorithms. *Appl. Intell.* 42, 4 (2015), 751–776.

[3] Marco Baioletti, Alfredo Milani, and Valentino Santucci. 2018. Algebraic Crossover Operators for Permutations. In *Proceedings of the 2018 IEEE Congress on Evolutionary Computation*. IEEE, 1–8.

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

[5] M. Chemillier. 2004. Periodic musical sequences and Lyndon words. *Soft Comput.* 8(9) (2004), 611–616.

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

[7] A. Clare and J. W. Daykin. 2019. Enhanced string factoring from alphabet orderings. *Inf. Process. Lett.* 143 (2019), 4–7.

[8] S. Corteel, G. Louchard, and R. Pemantle. 2006. Common intervals in permutations. *Discrete Mathematics & Theoretical Computer Science* 8, 1 (2006), 189–214.

[9] M. Crochemore and D. Perrin. 1991. Two-way string matching. *J. ACM* 38(3) (1991), 651–675.

[10] O. Delgrange and E. Rivals. 2004. STAR: an algorithm to search for tandem approximate repeats. *Bioinformatics* 20, 16 (2004), 2812–2820.

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

[12] A. E. Eiben and J. E. Smith. 2003. *Introduction to Evolutionary Computing*. Springer.

[13] G. Fogel and D. Corne. 2002. *Evolutionary Computation in Bioinformatics*. Morgan Kaufmann.

[14] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.* 10, 3 (2009), R25.

[15] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. 2014. Suffix array and Lyndon factorization of a text. *J. Discrete Algorithms* 28 (2014), 2–8.

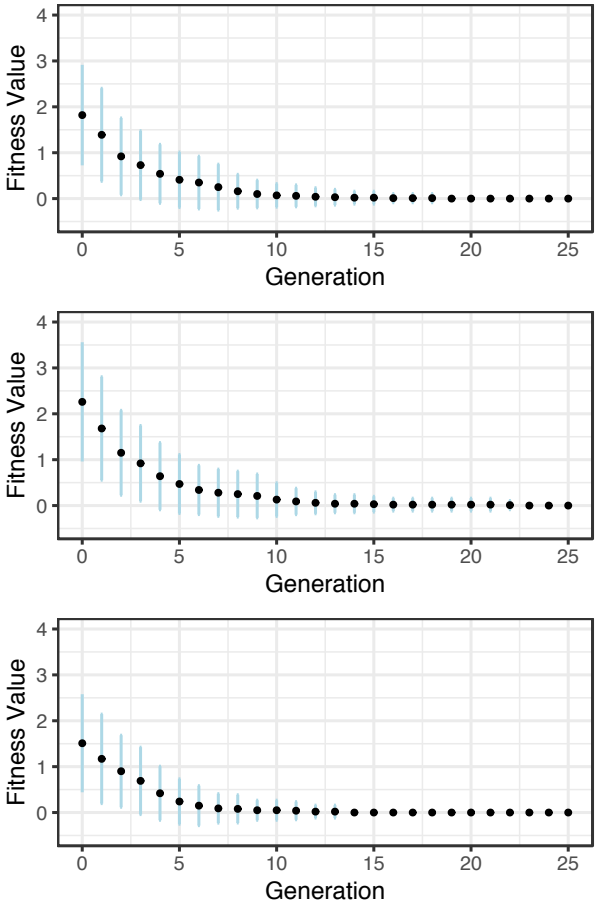

Figure 6: Specific = 12 – Average fitness and standard deviation over 100 independent runs for three random sequences of length 300 over an alphabet of size 20.

[16] N. A O'Leary et al. 2016. Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation. *Nucleic Acids Res.* 44(D1) (2016), D733–45.
[17] D. Whitley. 1997. Permutations. In *Handbook of evolutionary computation*, Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz (Eds.). IOP Publishing and Oxford University Press, C1.4.
[18] D. Whitley. 1997. Permutations. In *Handbook of evolutionary computation*, Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz (Eds.). IOP Publishing and Oxford University Press, C3.3.3.

ACKNOWLEDGMENTS

This research was part-funded by the European Regional Development Fund through the Welsh Government, grant 80761-AU-137 (West):

Cronfa Datblygu Rhanbarthol Ewrop
European Regional Development Fund

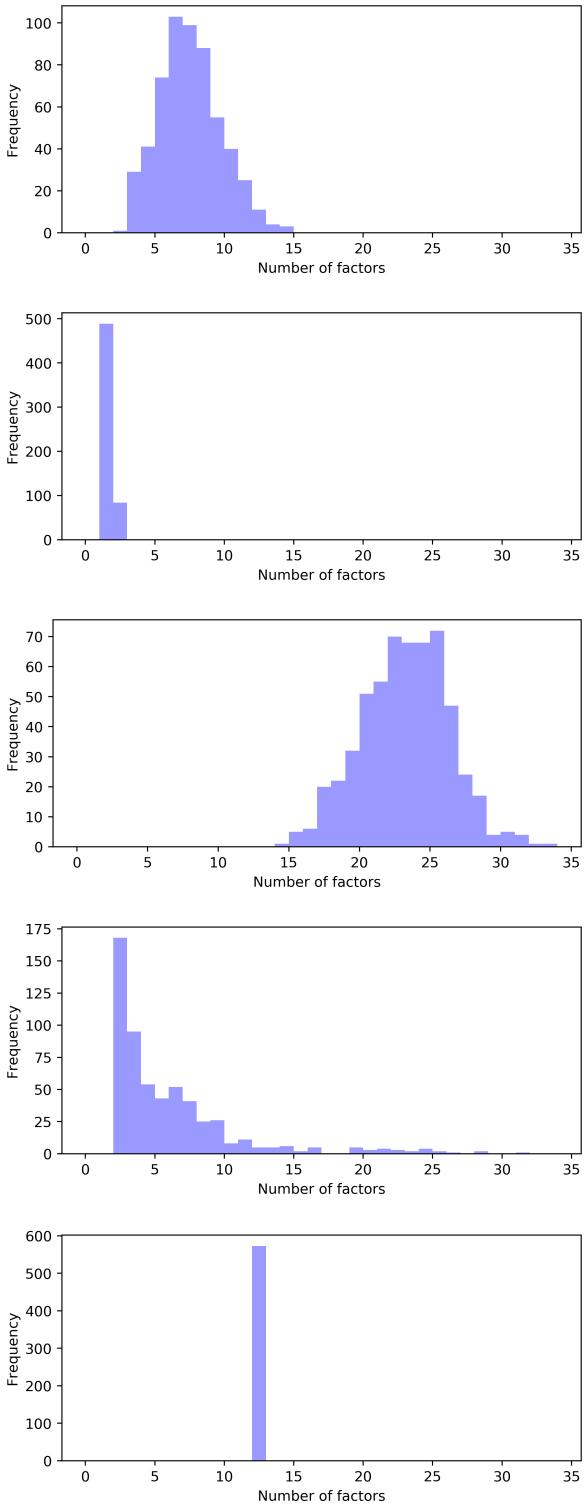

Figure 7: Distributions of the number of Lyndon factors per protein for lexicographic ordering and then using each of the fitness functions (from top to bottom: lexicographic, minimization, maximization, balanced, specific=12)