Journal of Heuristics (2025) 31:11
https://doi.org/10.1007/s10732-025-09548-3

Heuristics for the run-length encoded Burrows–Wheeler transform alphabet ordering problem

Lily Major[1] · Amanda Clare[1] · Jacqueline W. Daykin[1,2,3] · Benjamin Mora[4] · Christine Zarges[1]

Received: 18 July 2023 / Revised: 16 September 2024 / Accepted: 8 January 2025

Abstract

The Burrows–Wheeler Transform (BWT) is a string transformation technique widely used in areas such as bioinformatics and file compression. Many applications combine a run-length encoding (RLE) with the BWT in a way which preserves the ability to query the compressed data efficiently. However, these methods may not take full advantage of the compressibility of the BWT as they do not modify the alphabet ordering for the sorting step embedded in computing the BWT. Indeed, any such alteration of the alphabet ordering can have a considerable impact on the output of the BWT, in particular on the number of runs. For an alphabet Σ containing σ characters, the space of all alphabet orderings is of size $\sigma!$. While for small alphabets an exhaustive investigation is possible, finding the optimal ordering for larger alphabets is not fea-

Amanda Clare, Jacqueline W. Daykin, Benjamin Mora and Christine Zarges contributed equally to this work.

✉ Lily Major
jam86@aber.ac.uk

Amanda Clare
afc@aber.ac.uk

Jacqueline W. Daykin
jwd6@aber.ac.uk

Benjamin Mora
b.mora@swansea.ac.uk

Christine Zarges
chz8@aber.ac.uk

1 Department of Computer Science, Aberystwyth University, Penglais, Aberystwyth SY23 3DB, UK

2 Department of Information Science, Stellenbosch University, Merriman Avenue, Stellenbosch 7602, South Africa

3 Univ Rouen Normandie, INSA Rouen Normandie, Université Le Havre Normandie, Normandie Univ, LITIS UR 4108, F-76000 Rouen, France

4 Computer Science Department, Swansea University, Bay Campus, Fabian Way, Swansea SA1 8EN, UK

Published online: 28 January 2025

sible. Therefore, there is a need for a more informed search strategy than brute-force sampling the entire space, which motivates a new heuristic approach. In this paper, we explore the non-trivial cases for the problem of minimizing the size of a run-length encoded BWT (RLBWT) via selecting a new ordering for the alphabet. We show that random sampling of the space of alphabet orderings usually gives sub-optimal orderings for compression and that a local search strategy can provide a large improvement in relatively few steps. We also inspect a selection of initial alphabet orderings, including ASCII, letter appearance, and letter frequency. While this alphabet ordering problem is computationally hard we demonstrate gain in compressibility.

Keywords Alphabet ordering · Burrows–Wheeler transform · Compression · Local search · Random sampling · Run-length encoding

1 Introduction

The Burrows–Wheeler Transform (BWT), originally known as block-sorting compression, is a text transformation scheme which computes a permutation of an input string of data (Burrows and Wheeler 1994). The transformation is achieved by sorting the matrix of all circular shifts (rotations) of a string into lexicographic order and extracting the last column from the matrix. The algorithm can be implemented using a suffix array data structure with overall linear time complexity. Furthermore, using the index of the original string in the sorted matrix, the transform can be inverted in linear time, enabling efficient recovery of the input data. Hence the BWT is applicable to lossless compression activities, notably as a pre-processor preparing the data for compression. An important property of the transform is that it groups together characters with similar context, which are often identical characters, that is, it has a tendency to rearrange a string of characters into runs of the same character.

The computational efficiency of this remarkably simple innovation has enabled wide-ranging applications related to the indexing, searching, and compression of text (Adjeroh et al. 2008). The BWT is implemented in the popular open-source file compressor Bzip2 (Seward 1996), as well as in bioinformatic sequence alignment utilities including Bowtie2 (Langmead and Salzberg 2012), BWA (Li and Durbin 2009), and SOAP2 (Li et al. 2009), and additionally in image compression (Syahrul et al. 2008).

A large amount of research has been conducted to improve the space complexity of the BWT and common data structures used to query the BWT such as the FM-index. Such improvements include applying run-length encoding (RLE) to the transformed text or other structures.

This was first investigated by Mäkinen and Navarro as the run-length encoded FM-index (RLFM-index) (Mäkinen and Navarro 2005) and improved on (Sirén et al. 2008), including much more recently by Gagie, Navarro, and Prezza (Gagie et al. 2020), using a run-length encoded BWT (RLBWT) and suffix array samples (r-index). The r-index provides a full-text searchable index in $O(r)$ space, where r is the number of runs in the BWT, and has been of interest for the finding maximal exact matches step in bioinformatics read alignment (Rossi et al. 2022).

The order of the characters in the compressed BWT text heavily relies on the alphabet ordering used to sort the suffixes. By varying the alphabet ordering used for the BWT, the output can be influenced to further group characters, improving over the extended ASCII alphabet ordering which is typically implemented in software utilities.

Alphabet orderings for the BWT have been considered by Chapin and Tate (Chapin and Tate 1998), where both a hand-picked ordering and orderings created from a heuristic algorithm were used and tested on both text and imaging data. It is suggested that placing similar characters together in the alphabet (vowels, consonants, and punctuation) yield greater compression over ASCII in their pipeline involving the BWT.

Related BWT research has been conducted which, for a given ordered alphabet Σ, investigate a variety of non-lexicographic orderings of Σ^*, providing tailored methods to order the set of strings rather than reordering the alphabet. The ABWT is based on alternating lexicographical order (Giancarlo et al. 2018, 2020), which flips the order relation between $<$ and $>$ at each subsequent position during the scan of two strings being compared. The context adaptive BWT specifies an ordering for each longest common prefix in the Burrows–Wheeler Matrix (Giancarlo et al. 2023). The V-BWT is based on V-order which repeatedly deletes a V-type letter and at the penultimate stage of equality applies co-lexorder (Daykin and Smyth 2014). The D-BWT, which is applied to degenerate (also known as indeterminate) strings where each string position consists of a nonempty subset of letters over Σ, determines a lex-extension to sort the conjugates (Daykin and Watson 2017). The binary B-BWT applies binary block order and yields not one but twin transforms (Daykin et al. 2016).

Ordering texts within a collection has also been examined (Cox et al. 2012; Cazaux and Rivals 2019; Bentley et al. 2020), which involves adding a unique separator to each text in the collection. A notable result is that finding a minimal number of runs r by ordering of the texts can be done in linear time (Cazaux and Rivals 2019; Bentley et al. 2020), though this approach does not perform reordering of the alphabet within each text. Pibiri (2023) has implemented a graph-based algorithm for ordering the biological sequences within a collection such that the k-mers can be efficiently stored in a hash table. The ordering is chosen to ensure that the corresponding k-mer counts can be compressed using RLE while still enabling queries of the k-mer counts.

Reordering the alphabet to minimize r has been shown to be APX-hard (Gibney 2021; Bentley et al. 2020). In addition, Bentley, Gibney, and Thankachan showed that finding an alphabet ordering to ensure $r < t$ for a given threshold t is NP-complete (Bentley et al. 2020). We also know that r is limited to no more than twice the number of runs in the original text (Mantaci et al. 2017), providing a bound to the quality of any worst case alphabet ordering.

While the theoretical hardness of choosing the best alphabet ordering is clear, we still know little about the potential to efficiently and substantially improve on ASCII orderings. In this paper, we first randomly sample widely from the space and then use heuristic search to understand more about potential improvements to the alphabet ordering which could be made cheaply and quickly, reducing the size of RLBWT compressed texts.

We show that most randomly sampled alphabet orders achieved little improvement in size reduction. However, we show that a First-Improvement local search can quickly improve on randomly sampled alphabet orderings, even when using a limited number of steps. Additionally we consider initializing the search with promising initial orderings to seed the search, taking into account character frequency and appearance, and showing how they affect the speed of improvement of the search over time. We also evaluate a variety of operators for directing the search and find that a combined search strategy of two standard operators SWAP and INSERT (Eiben and Smith 2015) may improve the local minima depending on the order of their use. We further consider varied orderings of the neighbors of an alphabet ordering, searching them lexicographically, reverse-lexicographically, and randomly (Sect. 4.2).

Section 2 introduces and formalizes the problem and terminology. We discuss the special cases for small alphabets in Sect. 3, and First-Improvement local search methods for larger alphabets in Sect. 4. Our experimental setup is discussed in Sect. 5.1. A detailed discussion of our results can be found in Sect. 5. We summarize our contribution in Sect. 6 and provide an overview of future research directions.

2 Notation, problem definition, and modeling

In this section, we provide formal definitions for BWT and RLE with examples given for the key definitions. We also formally state the considered optimization problem. For each of the following definitions it is assumed that we have the following:

An *alphabet* Σ, is an ordered non-empty set of unique characters $\{x_0, x_1, \ldots, x_{\sigma-1}\}$, where $x_0 < x_1 < \cdots < x_{\sigma-1}$ and $x_i < x_j$ implies x_i precedes x_j. The standard 'Roman' alphabet ordering as implemented in the ASCII table is denoted as Σ_R.

A *string* $\boldsymbol{s} = c_0 c_1 \ldots c_{n-1} = \boldsymbol{s}[0..n-1]$ over an *alphabet* Σ, is a finite sequence of characters of length $n = |\boldsymbol{s}|$, such that $c_i \in \Sigma$, $c_i = \boldsymbol{s}[i]$. The individual characters c_i will be denoted as $\boldsymbol{s}[i]$ subsequently.

2.1 Key definitions

A BWT (Definition 2) is computed by first forming a Burrows–Wheeler Matrix (BWM, Definition 1) – the sorted list of strings formed by all *cyclic rotations* of a string – and taking the last column of the matrix. This has a tendency to group identical characters together and is useful for compressing with an RLE.

Alphabet reordering allows us to impact the ordering of the rows within a BWM. For any *string* $\boldsymbol{s}$ of length n with an *alphabet* Σ of size σ, there is a total of $\sigma!$ possible alphabet orderings.

Definition 1 (Burrows–Wheeler Matrix / BWM) Let $\boldsymbol{s}$ and $\boldsymbol{s'}$ be two strings over the same alphabet Σ. The string $\boldsymbol{s'}$ is said to be a *cyclic rotation* of $\boldsymbol{s}$ if and only if there exists two strings $\boldsymbol{u}$ and $\boldsymbol{v}$ with $|\boldsymbol{v}| = 1$ such that $\boldsymbol{s} = \boldsymbol{uv}$ and $\boldsymbol{s'} = \boldsymbol{vu}$.

The lexicographically ordered, row-arranged set of all *cyclic rotations* of $\boldsymbol{s}\$$ sorted according to the order of Σ, with \$ least in Σ, and denoted as $BWM(\boldsymbol{s}, \Sigma)$ is the BWM of $\boldsymbol{s}$. The BWM of $\boldsymbol{s}$ without adding an implicit \$ is denoted as $BWM_*(\boldsymbol{s}, \Sigma)$.

It should be noted that we do not consider the end marker (normally \$) to be movable within the alphabet ordering, instead always appearing first as an implicit least character. We illustrate Definition 1 in the following example. For the sake of simplicity we use the standard 'Roman' alphabet ordering Σ_R.

Example 1 Let a string $\boldsymbol{s} = cacatcg$ and using Σ_R, the BWM is as follows, where F and L denote the first and last columns of the matrix respectively:

$$BWM(\boldsymbol{s}, \Sigma_R) = \left|\begin{array}{cccccccc} F & & & & & & & L \\ \$ & c & a & c & a & t & c & g \\ a & c & a & t & c & g & \$ & c \\ a & t & c & g & \$ & c & a & c \\ c & a & c & a & t & c & g & \$ \\ c & a & t & c & g & \$ & c & a \\ c & g & \$ & c & a & c & a & t \\ g & \$ & c & a & c & a & t & c \\ t & c & g & \$ & c & a & c & a \end{array}\right|$$

Using Definition 1 we can now formally define a BWT as the last column L of the BWM.

Definition 2 (Burrows–Wheeler Transform / BWT) For an input string $\boldsymbol{s}$ of length n and an ordered alphabet Σ, we define the BWT as $\text{BWT}(\boldsymbol{s}, \Sigma) = \text{BWM}(\boldsymbol{s}, \Sigma)[i, n-1]$, $\forall i$ such that $0 < i < n-1$. The BWT of $\boldsymbol{s}$ without implicitly adding a \$ is denoted as $\text{BWT}_*(\boldsymbol{s}, \Sigma)$.

For example, the BWT for the string in Example 1 with the ordering Σ_R is therefore obtained from the last column L of the BWM as $gcc\$atca$.

In general, for any alphabet ordering Σ, the BWT can be computed from the suffix array (SA) of $\boldsymbol{s}$ as follows:

$$L[i] = \begin{cases} \$, & \text{if SA[i]=0} \\ \boldsymbol{s}\text{[SA[i]-1]}, & \text{otherwise.} \end{cases}$$

The suffix array may be computed in linear time (Ko and Aluru 2005; Kim et al. 2005) and enables BWT construction with the same time complexity. Inversion of the BWT may be achieved through the last-first mapping (Burrows and Wheeler 1994). For a string $\boldsymbol{s}$, the $\text{BWT}(\boldsymbol{s}, \Sigma)$ may be updated for a modified string $\boldsymbol{s'}$, $\text{BWT}(\boldsymbol{s'}, \Sigma)$ in sub-linear time (Kempa and Kociumaka 2022). Recent developments have led to an efficient algorithm for directly constructing the RLBWT (Nishimoto et al. 2022) in $O(n + r \log r)$ time but $O(n)$ time for strings where $r = O(n/\log n)$, and working space $O(r \log n)$ of bits.

Definition 3 (Substring) For a string $\boldsymbol{s}$, $\boldsymbol{u}$ is a *substring* of $\boldsymbol{s}$ if there are possibly empty strings $\boldsymbol{a}$ and $\boldsymbol{b}$ such that $\boldsymbol{s} = \boldsymbol{aub}$, and $\boldsymbol{u}$ is bounded by the indexes (i, j) where $0 \leq i \leq j \leq n-1$.

Definition 4 (Run / Maximal Unary Substring) For a string $\boldsymbol{s}$, $\boldsymbol{r}$ is a *run* of $\boldsymbol{s}$ if it is a *substring* of $\boldsymbol{s}$ and:

- Consisting of identical characters; $\boldsymbol{r}[j] = \boldsymbol{r}[k]\ \forall j, k < |\boldsymbol{r}|$.
- Of maximal length in $\boldsymbol{s}$.

Definition 5 (Run-Length Encoding / RLE) Let $\boldsymbol{p_0 p_1 p_2 \ldots p_{n-1}}$ be the sequence of *runs* of a string $\boldsymbol{s}$ such that $\boldsymbol{s} = \boldsymbol{p_0 p_1 p_2 \ldots p_{n-1}}$. RLE($\boldsymbol{s}$) is the string where each run of identical characters $\boldsymbol{p_i}$ in $\boldsymbol{s}$ is replaced by a single copy of that character and a count of the characters in the run. We denote the length of each run of characters in superscript, for instance: RLE($\boldsymbol{s}$) = $\boldsymbol{p_0}[0]^{|\boldsymbol{p_0}|}\boldsymbol{p_1}[0]^{|\boldsymbol{p_1}|}\ldots\boldsymbol{p_{n-1}}[0]^{|\boldsymbol{p_{n-1}}|}$.

We illustrate Definition 5 with an example using Σ_R.

Example 2 Let a string $\boldsymbol{s} = cacatcg$ using Σ_R, The output L column of $BWM(\boldsymbol{s}, \Sigma_R)$ is $gcc\$atca$. This is encoded as $\text{RLE}(BWT(\boldsymbol{s}, \Sigma_R)) = g^1c^2\$^1a^1t^1c^1a^1$, $|\text{RLE}(BWT(\boldsymbol{s}, \Sigma_R))| = 14$.

We are interested in overall memory usage for the RLE, so each run is encoded as the byte (character) and then the length of the run up to 255. Any runs over 255 in length are encoded as additional bytes (Sect. 2.2). This is represented as $|RLE(\boldsymbol{s})|$.

To demonstrate the influence an alphabet ordering can have on the BWM, BWT and RLE, we consider the same example string using an alternative alphabet ordering.

Example 3 Let a string $\boldsymbol{s} = cacatcg$, $\Sigma = \$ < a < g < c < t$.

$$BWM(\boldsymbol{s}, \Sigma) = \left|\begin{array}{cccccccc} F & & & & & & & L \\ \$ & c & a & c & a & t & c & g \\ a & c & a & t & c & g & \$ & c \\ a & t & c & g & \$ & c & a & c \\ g & \$ & c & a & c & a & t & c \\ c & a & c & a & t & c & g & \$ \\ c & a & t & c & g & \$ & c & a \\ c & g & \$ & c & a & c & a & t \\ t & c & g & \$ & c & a & c & a \end{array}\right|$$

With this new ordering of Σ, $\text{RLE}(BWT(\boldsymbol{s}, \Sigma)) = g^1c^3\$^1a^1t^1a^1$, $|\text{RLE}(BWT(\boldsymbol{s}, \Sigma))| = 12$.

Definition 6 (r value) The number of runs in the BWT.

Let $\boldsymbol{p_0 p_1 p_2 \ldots p_{n-1}}$ be the sequence of *runs* of a string $\boldsymbol{s}$, then $r(\boldsymbol{s}, \Sigma) = n$.

We clarify Definition 6 with an example.

Example 4 For example, let $\boldsymbol{s} = acacacbbacbac$ and $\Sigma = a < b < c$. $\text{BWT}_*(\boldsymbol{s}, \Sigma) = bccbccbcaaaaa$. As there are 7 runs in the BWT output, $r = 7$.

The r value for any BWT of a string is not necessarily less than the r value of a string itself. A classic example is the string $mississippi\$$ where the BWT is $ipssm\$pissii$ which also shows that computing the BWT permutation of the input data does not necessarily make the data more compressible.

2.2 Problem statement

We wish to investigate the sample space of alphabet orderings. Since reordering the alphabet to minimize the number of runs has been shown to be APX-hard, we will use and investigate different heuristics.

We evaluate the fitness of any new alphabet ordering Σ by the total length of its RLE. As we use the input data bytewise, the 'characters' in our alphabet are these bytes and not another encoding such as UTF. We do not use the total number of identical character runs in the BWT(r) as the fitness since we are interested in overall memory consumption of the representation. Each input file is read bytewise, so multi-byte (non-ASCII) characters are represented as more than one 'character' in the alphabet.

We encode the RLE as a sequence of byte pairs, with the first byte of each pair representing the run's character, and the second byte representing the length of the run. As the maximum value that may be represented in a byte is 255, any larger runs will be represented with multiple pairs of bytes.

We therefore seek to minimize the size of RLE for our tested texts.

RLBWT Alphabet Ordering Problem
Input: A string $\boldsymbol{s}$ of length n over an alphabet Σ of size σ
Output: A reordering of Σ for $RLE(BWT(\boldsymbol{s}, \Sigma))$ that minimizes $|RLE(BWT(\boldsymbol{s}, \Sigma))|$.

3 Methods for small alphabets

3.1 Binary alphabet orderings

We consider the simplest non-trivial case where there are only 2 characters and show that when using the reverse of an alphabet ordering that r remains the same for both orderings for primitive strings.

Lemma 1 *Let $\boldsymbol{s}$ be a binary string over $\Sigma = \{a, b\}$ of length n. Let $\Sigma_1 = a < b$, $\Sigma_2 = b < a$. Then $r(\boldsymbol{s}, \Sigma_1) = r(\boldsymbol{s}, \Sigma_2)$.*

Proof Suppose $\boldsymbol{s}$ is primitive, then all of its conjugates are distinct (Petersen 1996). Let $\boldsymbol{p} = p_0 \dots p_{n-1}$ and $\boldsymbol{q} = q_0 \dots q_{n-1}$ be two adjacent rows in $\mathrm{BWM}_*(\boldsymbol{s}, \Sigma_1)$ such that $\boldsymbol{p}$ is lexicographically less than $\boldsymbol{q}$. Assume that $\boldsymbol{s}$ contains two distinct characters (otherwise the claim is trivial), then let t be the minimal index such that $p_t \neq q_t$, thus $p_t = a < b = q_t$. On the other hand in $\mathrm{BWM}_*(\boldsymbol{s}, \Sigma_2)$, $\boldsymbol{q}$ is lexicographically less than $\boldsymbol{p}$. The argument holds for all pairs of adjacent rows showing that the two matrices are flipped row-wise, hence have the same r value.

In the case that $\boldsymbol{s}$ is not primitive (thus having some cyclic rotations which are not distinct) and has the form $\boldsymbol{u}^k$, $k > 1$, then all groups of k identical and adjacent rows in $\mathrm{BWM}_*(\boldsymbol{s}, \Sigma_1)$ will likewise be adjacent in $\mathrm{BWM}_*(\boldsymbol{s}, \Sigma_2)$ after the flipping. □

Observe that using transitivity the argument on reversing an alphabet extends to an arbitrary finite alphabet which motivates our search for effective orderings. We illustrate concepts with the following ternary example:

Example 5 (Ternary primitive)
Let $\boldsymbol{s} = aabbcc$, and $\Sigma_1 = a < b < c$:

$$BWM_*(\boldsymbol{s}, \Sigma_1) = \begin{vmatrix} F & & & & & L \\ a & a & b & b & c & c \\ a & b & b & c & c & a \\ b & b & c & c & a & a \\ b & c & c & a & a & b \\ c & a & a & b & b & c \\ c & c & a & a & b & b \end{vmatrix}$$

$\mathrm{BWT}_*(\boldsymbol{s}, \Sigma_1) = caabcb$, $r(\boldsymbol{s}, \Sigma_1) = 5$.
In the case with $\Sigma_2 = c < b < a$:

$$BWM_*(\boldsymbol{s}, \Sigma_2) = \begin{vmatrix} F & & & & & L \\ c & c & a & a & b & b \\ c & a & a & b & b & c \\ b & c & c & a & a & b \\ b & b & c & c & a & a \\ a & b & b & c & c & a \\ a & a & b & b & c & c \end{vmatrix}$$

$\mathrm{BWT}_*(\boldsymbol{s}, \Sigma_2) = bcbaac$, $r(\boldsymbol{s}, \Sigma_1) = 5$.
Thus both orderings have the same r value.

Example 6 (Ternary non-primitive) Let $\boldsymbol{s}$ = abcabc, and $\Sigma_1 = a < b < c$:

$$BWM_*(\boldsymbol{s}, \Sigma_1) = \begin{vmatrix} F & & & & & L \\ a & b & c & a & b & c \\ a & b & c & a & b & c \\ b & c & a & b & c & a \\ b & c & a & b & c & a \\ c & a & b & c & a & b \\ c & a & b & c & a & b \end{vmatrix}$$

Likewise for $\Sigma_2 = c < b < a$ it is trivial to see that the r value of both orderings is the same.

$$BWM_*(\boldsymbol{s}, \Sigma_2) = \begin{vmatrix} F & & & & & L \\ c & a & b & c & a & b \\ c & a & b & c & a & b \\ b & c & a & b & c & a \\ b & c & a & b & c & a \\ a & b & c & a & b & c \\ a & b & c & a & b & c \end{vmatrix}$$

Table 1 Percentage difference in file size using each of the 24 alphabet orderings for the alphabet {a,c,g,t} for the 150 concatenated *E.coli* data files when using the RLBWT

Min % change	Max % change	Mean	Std
− 91.576	− 91.627	− 91.597	0.02

3.2 Exhaustive search on biological data

For small alphabets it may be feasible to search through all possible alphabet orderings to find the one(s) that provide(s) the best RLBWT compression of the data.

For example, in the case of genomic data, such as a collection of the genome sequences of many *E. coli* bacteria, we could expect a limited 4-letter alphabet $\{a,c,g,t\}$ representing nucleotides giving 24 possible alphabet orderings. For such a collection, the genomes would share much in common, and the RLBWT of a concatenation of these sequences should capture the commonalities effectively. However, almost all bioinformatics algorithms choose to use the ASCII ordering $a < c < g < t$, even though this may not provide the best results.

We took a collection of 150 diverse *E. coli* genomes from NCBI and compared the compression obtained using each possible alphabet ordering on the concatenated 150 genomes—this totaled 724,548,306 bytes. The ordering $t < c < a < g$ provided the best compression (-91.627% change), and the worst ordering was $c < g < t < a$ (-91.576% change)—for further details see Table 1. Although an exhaustive search through all possible alphabet orderings is often prohibitively expensive, this example demonstrated that better choices can make improvements, and motivated our investigation to find such orderings and examine the search space of orderings for its properties.

4 Methods for larger alphabets

Since the permutation space of the alphabet ordering for an alphabet of size σ is $\sigma!$, in most practical cases, exhaustive enumeration of all alphabet orderings is not a feasible approach. We therefore consider different variants of a First-Improvement local search for larger alphabets. We use different types of texts, considering a variety of different text lengths and alphabet sizes. The main goal of our experimental analysis is to provide insights into the working principles of the considered methods for the given problem and to provide guidelines for their use. The considered algorithms are introduced in Sects. 4.1 and 4.2.

4.1 Baseline: random sampling

We use uniform random sampling as a baseline approach to inspect the statistical distribution of potential compression gains that could be made by reordering the alphabet. The results are compared with the results of First-Improvement local search as described in the next section.

An interesting property of random sampling is that the mean number of improvements to be expected is actually bounded by the logarithm of number of trials. Indeed, as every new sample (i.e., alphabet order) is independent of the previous samples, the chance of obtaining a better compression after T sampling steps with a new sample would be $P_{improving(T)} = \frac{1}{T+1}$, should the compression value obtained be unique for every sampling event. Therefore, the expected number of successive improvements in compression using T random samples would be simply given by $\sum_{i=0}^{T} \frac{1}{i+1} = O(\log T)$. In practice, the compression value will belong to a limited set of integer values, and each new sample result may just be equal to the best compression value obtained so far. As such, the actual chance $P_{improving}(T)$ after T samples to improve compression is less than $\frac{1}{T+1}$, and $O(\log T)$ becomes an upper bound for the total number of improvements expected from T random samples.

4.2 First-improvement local search

We consider a variant of First-Improvement Local Search as our main optimisation approach. Pseudocode for this approach is given in Algorithm 1. The algorithm takes as input a text and its alphabet. It starts from some initial alphabet ordering π (line 3) and tries to improve the ordering until a provable local optimum is reached (line 15). In each loop it considers all neighbors of the current ordering π in a given order (line 7). If an improvement is found (line 8), the algorithm moves to the better ordering (line 10) and repeats the process (updating the neighborhood as needed, line 11). The fittest ordering is returned.

In our experiments, we consider 9 different initialization methods in line 3 of Algorithm 1 as discussed in Sect. 4.2.1. We also consider 12 different neighborhoods as discussed in Sect. 4.2.2, leading to a total number of 108 algorithm configurations.

Algorithm 1 First-improvement local search

```
1: Input: A string w over an alphabet Σ of size σ
2: Output: An ordering π of Σ
3: π ← INITIALISEORDER(Σ)                    ▷ Initialisation (Section 4.2.1)
4: N ← INITIALISENEIGHBORHOOD(π)             ▷ Neighbors of π (Section 4.2.2)
5: do
6:     improvementFound = false
7:     for all π′ ∈ N do
8:         if f(π′) < f(π) then              ▷ Neighbor is improvement
9:             improvementFound = true
10:            π ← π′
11:            N ← UPDATENEIGHBORHOOD(π)
12:            break
13:        end if
14:    end for
15: while (improvementFound)
16: return π
```

4.2.1 Initialization

We consider the following 9 initialization methods in line 3 of Algorithm 1. With the exception of random initialization, all methods are deterministic and based on some heuristic or standard ordering from the literature.

- **Random:** We determine 20 random orderings using Fisher-Yates shuffle (Knuth 1998a). The same 20 orderings are used for all experiments using random initialization.
- **ASCII:** The (extended) ASCII ordering of the alphabet.
- **First appearance:** Characters are ordered by their order of appearance in the text.
- **Least frequent:** Characters are ordered by the number of occurrences in the text, least frequent first.
- **Most frequent:** Characters are ordered by the number of occurrences in the text, most frequent first.
- **Chapin tate:** A hand-tuned ordering used by Chapin and Tate (Chapin and Tate 1998)[1] for a similar compression problem based around the BWT. The ASCII alphabet ordering, however '@' replaces '!', '+,-.' is rearranged to '+-,.', AEIOU and aeiou are both brought to the front of each block of both upper and lower case. The consonants are reorganised as $B < C < D < G < F < H < R < L < S < M < N < P < Q < J < K < T < W < V < X < Y < Z$.
- **Inverse permutation Chapin Tate:** The inverse permutation (Knuth 1998b) of the Chapin Tate ordering, spacing out the vowels and reordering the consonants. Since grouping vowels is important for the Chapin Tate ordering we investigate the effects of not doing this.
  The vowels are ordered as $A < I < O < U < E$, however they are interspersed through the ordering instead of being together at the start of the ordering. This results in an ordering of $A < F < G < H < B < J < I < K < C < S < T < M < O < P < D < Q < R < L < N < U < E < W < V < X < Y < Z$. Other changes to the ordering such as '!' and '@' being swapped, and the '+-,.' rearrangement are also present in this ordering.
- **Vowels**: Vowels (aeiouAEIOU) are placed at the beginning of the ordering. Similar to the Chapin Tate ordering, the main aim is to explore if grouped punctuation and the consonant reordering really helps the problem, or if moving the vowels alone will yield a better initial ordering.
- **FDA:** The FDA algorithm was introduced to determine an alphabet ordering for a variant of the Lyndon factorization problem (Major et al. 2020). The main motivation to include this method is to explore if these orderings might be more generally useful for alphabet ordering problems on strings.
  It should be noted that FDA determines a partial alphabet ordering. A total ordering is produced from the partial ordering by topological sort.

For each of these orderings, we place the selected end marker character (usually $) as least in the ordering when performing the BWT.

[1] And personal communications with B. Chapin.

4.2.2 Local search neighborhoods

For the neighborhoods in lines 4 and 11 of Algorithm 1 we consider combinations of two standard operators for permutation sample spaces, namely SWAP (aka EXCHANGE) and INSERT (aka JUMP) (Eiben and Smith 2015). SWAP picks two integers $0 \leq i < j \leq \sigma - 1$ and swaps the characters at positions i and j. INSERT picks two integers $0 \leq i, j \leq \sigma - 1$ with $i \neq j$. It moves the character at position i to position j, shifting all subsequent characters to the right.

Looking at the two operators in isolation we observe that SWAP yields a neighborhood of size $\sigma(\sigma - 1)/2$ while INSERT yields a neighborhood of size $\sigma(\sigma - 1)$. Both neighborhood sizes are quadratic in σ, the size of the alphabet.

We first consider both operators in isolation and investigate three different orderings of the neighbors in the neighborhoods:

- **Random order:** Using a random order of the neighbors is the most common approach.
- **Lexicographic order (LEX):** We hypothesize that it maybe be beneficial to first fix characters at the start of the ordering. We therefore consider the fixed lexicographic order of neighbors. For example, for SWAP we consider i-j-pairs in the following order: $(0, 1), (0, 2), ..., (0, \sigma - 1), (1, 2), ..., (\sigma - 2, \sigma - 1)$. Any unspecified neighborhood ordering should be assumed to be LEX.
- **Reverse lexicographic order (REVLEX):** We consider the opposite case by reversing the lexicographic order given above.

Finally, we consider a combination of SWAP and INSERT. More precisely, we first try all possible SWAPs followed by all possible INSERTs and vice versa. For each of the two lists of neighbors we consider all three orders defined above, ensuring that all SWAPs are sorted before all INSERTs (and vice versa) as appropriate. If a combination of operators is used, the second operator will only be used until an improvement is found. The algorithm then returns to using the first operator.

5 Results and discussion

5.1 Experimental setup

All our main experiments are run on Super Computing Wales[2] on a single node (2x Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz with 20 cores each) and Aberystwyth DCS cluster on a single node (2x Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz with 48 cores each). As usual, we report the number of objective function evaluations rather than wall-clock time.

We remark that we use super computing resources to extensively explore the search space, but on the other hand for a targeted application, for instance on a specified dataset, a commodity machine has been found to be time-efficient in practice. To demonstrate this, we used a single core of an AMD Ryzen 7 PRO 3700U @ 2.30GHz, and the `alice29.txt` file (See Table 2). As we show in Table 5, the number of

[2] https://portal.supercomputing.wales/index.php/about-sunbird/.

local search steps to beat the best ordering that was randomly sampled is relatively small. Thus we report the wall-clock time taken to run for only 1000 steps (fitness evaluations) using SWAP, LEX only (See Sects. 4.2.1 and 4.2.2). We found that this took an average of 47.5 s for each of the 28 initializations. Though it should be noted that this time increases linearly with the number of cores and in practice it is likely that a single initialization will be used.

For each stochastic variant of First-Improvement local search, we report statistics on the results of 20 independent runs. However, to avoid the distortion of our results due to different random starting points, all variants with random initialization use the same 20 starting points which were randomly determined prior to running our experiments. This way we can analyze the effect of different neighborhoods on the same starting points without introducing additional variables. For each run we report the number of function evaluations as 'steps'.

We define C as the percentage change in file size relative to the uncompressed size (measured as a percentage in bytes):

$$C = \left(\frac{\text{Compressed Size} - \text{Uncompressed Size}}{\text{Uncompressed Size}} \right) \cdot 100 \tag{1}$$

We present C as raincloud plots (Allen et al. 2021) (a combination of a distribution, boxplot, and jittered point cloud) to give an indication of the density and shape of the sample space. A negative value for C demonstrates a reduction in size while a positive value demonstrates an increase in size. The smaller the value for C the better the compression. The number of steps presented is hitting time and not exhaustive checking of the neighbors.

5.2 Benchmarking

We use a standard benchmark for data compression for our analysis, namely the Canterbury corpus (Arnold and Bell 1997). Table 2 lists the different files contained in the corpus, including the size of each file and the corresponding alphabet size. The file `kennedy.xls` is excluded from our experiments for technical reasons: our BWT implementation relies on a unique end marker character and the alphabet size of 256 leaves no available character if the file is used bytewise. We reorder the alphabet by mapping the input text characters to new ones based on the order of characters in the alphabet. We then use the SAIS suffix array implementation by Yuta Mori[3] to compute the suffix array (Nong et al. 2011; Ko and Aluru 2005). Another implementation is also provided in our repository[4] but was not used due to speed.

We do not run to completion for some files in the corpus (for example: ptt5, sum, xargs.1) for some of the SWAP THEN INSERT and INSERT THEN SWAP methods due to prohibitive runtimes. Instead we run to a limit of 10,000 steps as this is more than

[3] https://web.archive.org/web/20230309123010/https://sites.google.com/site/yuta256/sais.

[4] https://github.com/jam86/Heuristics-for-the-Run-length-Encoded-Burrows--Wheeler-Transform-Alphabet-Ordering-Problem.

Table 2 Files in the Canterbury corpus with their size in bytes and the number of unique bytes in their alphabet

File	Description	Bytes	Alphabet
`alice29.txt`	The text of Alice's adventures in Wonderland	152,089	74
`asyoulik.txt`	Text from Shakespeare's play as you like it	125,179	68
`cp.html`	HTML with a large number of links	24,603	86
`fields.c`	C source code	11,150	90
`grammar.lsp`	LISP source code	3721	76
`kennedy.xls`	Microsoft excel document	1,029,744	256
`lcet10.txt`	Conference proceedings	426,754	84
`plrabn12.txt`	Text from John Milton's Paradise lost	481,861	81
`ptt5`	Fax data	513,216	159
`sum`	Sun SPARC executable	38,240	255
`xargs.1`	GNU man page for xargs	4227	74

the maximum number of steps to outperform a random sample (Table 5). A full list of the files and methods run until 10,000 steps only is available in our repository.

5.3 Randomly sampled alphabet orderings

We inspected the landscape of percentage change in compression that can be achieved using the RLBWT by changing the alphabet order, and sampled 240,000 alphabet orders uniformly at random (by Fisher-Yates shuffle) for each of the texts. We do this to learn more about the shape of the sample space and to understand whether there are many best orderings to be found or few.

The number of samples was chosen since it was large but remained tractable to compute on a local machine (Intel(R) i7-8700K CPU @ 3.70GHz with 12 cores). To exemplify our findings, the results for files `alice29.txt`, `sum` and `fields.c` are shown in Fig. 1. The number of samples may be few in comparison with the very large $\sigma!$ space, however due to the smooth overall distributions without outliers we can see that the sampling already gives a clear picture of the shape of the space from which further random samples would be obtained. The distributions are bell-shaped but not normally distributed (scistats.normaltest Virtanen et al. (2020)), having a long thin tail downwards where better solutions can be found.

These distributions demonstrate a spread of percentage compression for different alphabet orders, with the majority being sub-optimal choices and the sample space having only a thin tail of better choices. These figures also highlight the surprisingly good compression achieved by the ASCII ordering, lying far below most random choices of alphabet order, even for executable files such as `sum`. The full set of figures for all corpus files can be seen in our repository.

However, most texts in the corpus can be compressed to be smaller than the original by using the RLBWT, despite the fact that most randomly chosen alphabets are poor choices (See Table 3). This is true even with the worst choice of alphabet order. The

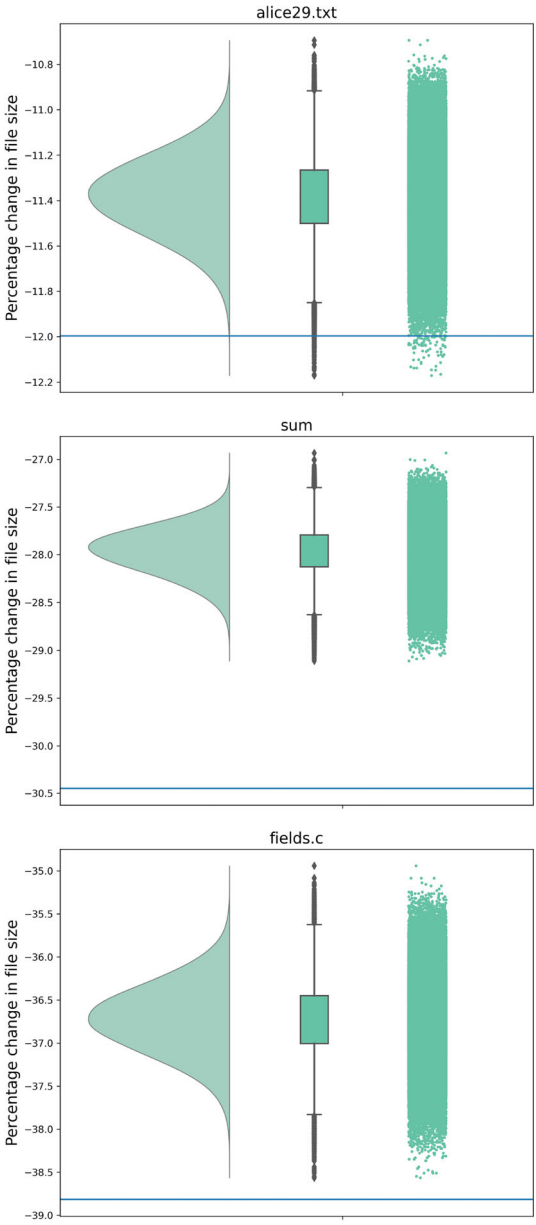

Fig. 1 A raincloud plot of the percentage change in compression for 240,000 random samples of alphabet orderings used with the RLBWT for three of the corpus texts. Similar shaped distributions can be seen for the novel `alice29.txt`, SPARC executable file `sum` and C source file `fields.c`. Horizontal blue line represents the ASCII ordering, which outperforms most of the randomly sampled orderings

best alphabet order found when randomly sampling for `ptt5` reduced the file size by 74.207%. The file `plrabn12.txt` does not compress well and the best alphabet order sampled for this file increased the size by 1.019%, and ASCII performs worse than the best randomly sampled alphabet order.

While for most files, randomly sampled alphabet orderings may compress the size of the file somewhat, few sampled orderings improve on the ASCII alphabet ordering. In

Table 3 Percentage difference compared to the original file size for each file when using the 240,000 orderings found using random sampling

File	Min % change	Max % change	Mean	Std
`alice29.txt`	− 12.171	− 10.694	− 11.385	0.172
`asyoulik.txt`	− 0.533	1.028	0.324	0.177
`cp.html`	− 25.375	− 23.156	− 24.163	0.255
`fields.c`	− 38.565	− 34.942	− 36.725	0.407
`grammar.lsp`	− 28.353	− 21.903	− 24.955	0.711
`lcet10.txt`	− 22.116	− 21.140	− 21.554	0.113
`plrabn12.txt`	1.019	2.073	1.633	0.109
`ptt5`	− 74.207	− 73.664	− 73.914	0.080
`sum`	− 29.111	− 26.935	− 27.967	0.251
`xargs.1`	− 6.269	0.781	− 2.484	0.735

The minimum, maximum, and mean percentage changes are shown for each text

fact, several of the initialization ordering methods (Sect. 4.2.1) also already outperform even the best of the randomly sampled orderings for many of the files. The heatmap in Fig. 2 shows the rankings of the initialization orderings for the different files. The benefits of ASCII and Chapin Tate orderings can be seen clearly in this figure. It can also be seen from this figure that even after randomly sampling 240,000 orderings, the best of these is not good enough. Random sampling is therefore a too costly search strategy and necessitates another solution.

5.4 Improved fitness with local search

In contrast to the compression obtained via random sampling of alphabet orders (Sect. 5.3), even the most simple of our local search methods—Swap, Lex—can achieve a better fitness value for our tested data. Figure 3 shows the contrast between the best solutions found during a local search (using the Swap operator) and the results of the random sampling. This figure shows `alice29.txt`, `sum` and `fields.c`. Other corpus files have similar plots, which can be seen in our repository. The large gap between their distributions indicates that there are excellent alphabet orderings that have not previously been sampled, even when sampling 240,000 orderings.

5.5 The impact of initialization on local search

When performing a local search, the initial alphabet order makes a difference to the best solution that can be found by the local search using the Swap operator.

Table 4 shows which initialization ordering achieves the best compression when the search terminates at the local minimum or after 1000 steps. While randomly chosen orders are competitive if the search is terminated early after 1000 steps, ASCII, Chapin Tate, and Inverse Permutation Chapin Tate perform best if the search is allowed to complete.

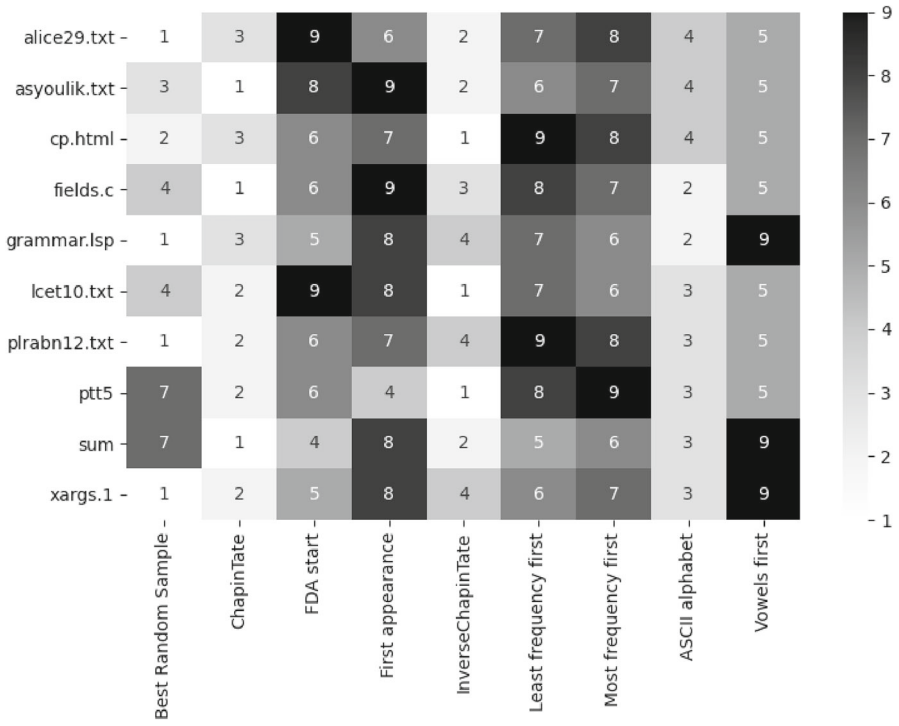

Fig. 2 Ranking of each initialization method and the best randomly sampled ordering for each file before local search is applied

The plots in Fig. 4 exemplify how the results improve over the time taken by the search. A steep drop in the size of the file is observed followed by a large number of steps until the local optima is reached. The initialization order that leads to the best local minimum at the end of the search is not obvious at the start of the search, nor is there consistently an initialization order that would produce a good local minimum across all files. The range of random sample fitness may completely cover the fixed start positions as in `alice29.txt` or lay above many of the best fitness starts as in `fields.c`. Overall the range of random sample starts covers a large amount of solutions that are found using the fixed start positions in our tested files.

The steep improvement early in the search suggests that a limited search may still be beneficial. If the search is terminated early, ASCII and Chapin Tate are the best orderings from which to start the search (Fig. 5).

While using a random initialization still yields an improvement in fitness over time, the overall change in fitness is not as good as choosing a fixed initialization for the explored texts. It may be possible for multiple orderings to achieve around the same fitness for 1000 neighbor evaluations.

From this we conclude that the landscape has lots of local optima and that the path through the landscape is therefore important. We examine other methods such as

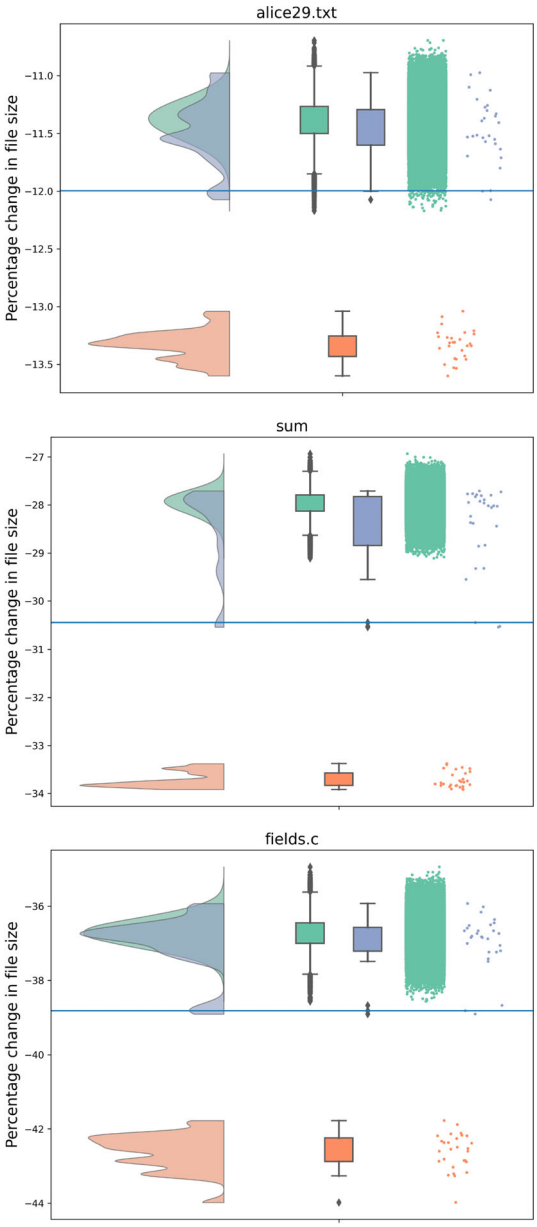

Fig. 3 Raincloud plot showing that the best alphabet orders found at the conclusion of local search (orange) give noticeably better compression than that achieved using either randomly sampled alphabet orders (green) or the ASCII alphabet order (horizontal line). The compression achieved by random samples of alphabet orderings are displayed in green. The best achieved via local search with SWAP only is shown in orange, (also see Sect. 5.4). The change in compression for the different local search initialization orderings at the start of the search are shown in blue, and these overlap with the random samples. The compression when using the ASCII alphabet is plotted as a horizontal blue line and can be seen at the lower end of the random samples

reverse and random neighbor orderings, and using INSERT to make further jumps in the landscape.

Table 4 Determining the best initialization orderings for an early-terminated search

File	After 1000 steps Best initialization (1000)	C (1000)	At local minimum Best initialization (all)	C (all)
alice29.txt	IPCT	− 12.368	Random	− 13.601
asyoulik.txt	CT	− 1.108	CT	− 2.07
cp.html	IPCT	− 25.920	Random	− 27.993
fields.c	CT	− 40.359	ASCII	− 43.982
grammar.lsp	Random	− 29.589	Random	− 33.996
lcet10.txt	IPCT	− 22.503	Random	− 23.04
plrabn12.txt	ASCII	0.948	Random	0.228
ptt5	IPCT	− 74.472	IPCT	− 74.748
sum	ASCII	− 30.737	FDA	− 33.917
xargs.1	CT	− 7.783	CT	− 12.042

Local search was performed with the SWAP, LEX neighborhood for each corpus file, terminating after 1000 steps. IPCT = Inverse Permutation Chapin Tate, CT = Chapin Tate, Random = Random Initialization

5.6 Local search operators: SWAP and INSERT

The path that the local search explores through the search space is important in determining where in the space the local optima is found. Evaluating more local search neighbors may therefore lead to finding better neighbors and a better overall optima. Since SWAP moves two elements into new locations at once. We therefore consider the INSERT operator, either in combination with SWAP (as first or second operator) or alone.

Table 5 shows the number of steps taken using each of the different search operators to find an ordering that performs better than the best of the 240,000 randomly sampled orderings. We note that local search only requires a few steps before outperforming random sampling, independently of any search operators, and typically takes only a few seconds to compute. Indeed, Table 6 shows that methods based on local search create a large number of updates when compared to a naive random sampling of the search space. While the average number of updates obtained with the latter is close to the theoretical one (12.4 vs. 12.38), we see that local search algorithms provide substantially more updates. For instance, the SWAP method using the lexicographic order generates on average 236 updates across 10 files. Such a number of updates with random sampling would on expectation require more than 10^{100} samples ($\approx e^{236.5}$), which is today not computable.

Methods involving randomized neighbor orderings perform well in the time taken to beat the best of the 240,000 random samples compared to all other methods. Of these, the SWAP THEN INSERT operator performs best.

However, to fully locate any local minimum may in some cases take a very long time (Fig. 6). Our experiments are limited to 10 million steps for any single run, which is reached for some configurations. When considering the initialization that locates the best fitness for any method, we find an initial steep drop in fitness for all methods, and that there are groupings of methods that perform similarly. Generally the methods

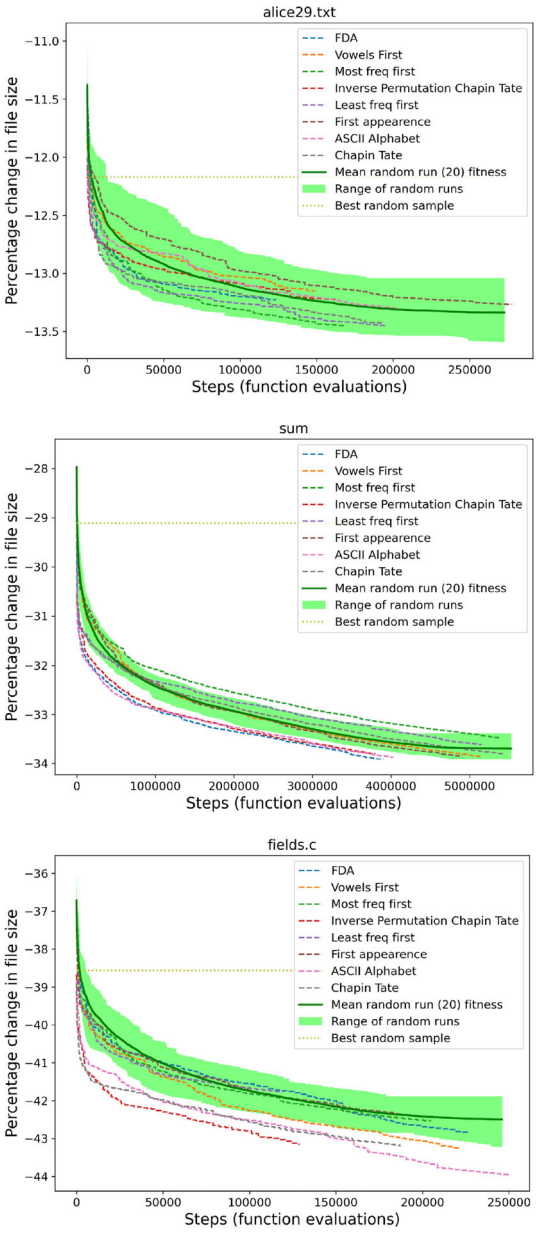

Fig. 4 Local search using SWAP, LEX from different alphabet order initialization methods over time until a local minimum is reached. It can be observed that there is no best initial order that consistently results in the best local minima for all texts

involving randomized neighbor orderings perform well in few steps and have a fitness which remains competitive with the LEX and REVLEX methods. We observe that even when the number of steps is limited to relatively few in comparison to the number needed to reach a local minimum, the percentage change in file size reached may still be good.

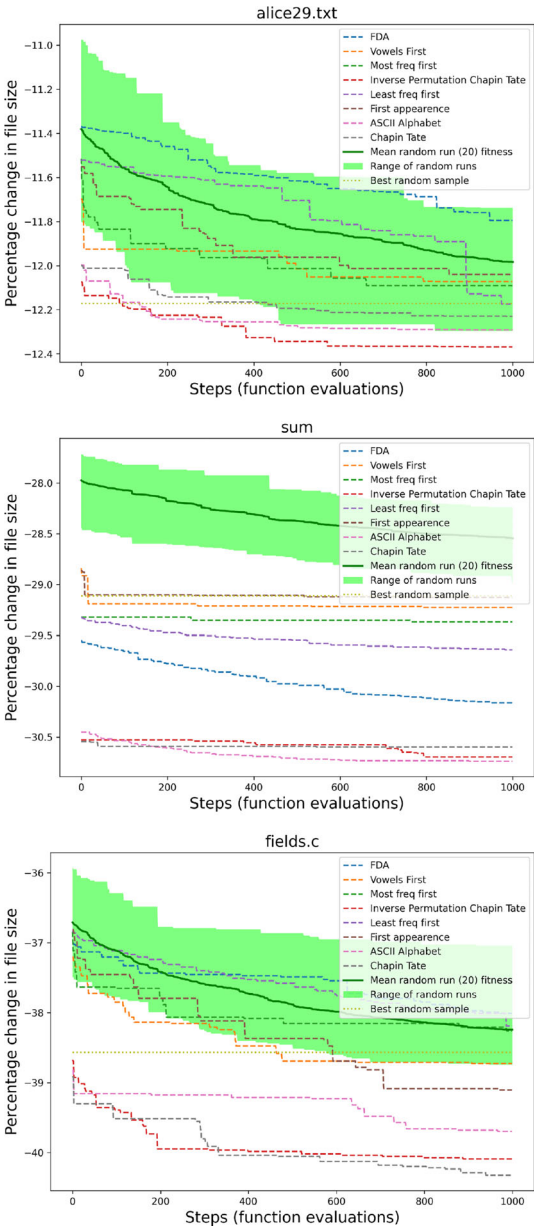

Fig. 5 Local search for different alphabet order initialization methods over time with SWAP, LEX, limited to 1000 neighbor evaluations. It can be observed that there is no best initial order that consistently results in the best local minima for all texts, and that the best initialization order is dependent on the number of steps performed

This is further demonstrated when considering the final fitness achieved in all runs across all methods. Across our tested files, the number of required steps to locate a minimum splits the methods into three groups (in increasing order of steps): random neighbor ordering methods, SWAP and SWAP-first methods, and INSERT and INSERT-first methods (Fig. 7). Even when considering the final fitness achieved, a randomized neighbor ordering remains competitive (Fig. 8). The file `sum` is not shown as it has

Table 5 The minimum number of local search steps for the best initialization to find an ordering performing better than the best ordering out of the 240,000 randomly sampled orderings

Method	alice29.txt	grammar.lsp	plrabn12.txt	xargs.1	All others
I, LEX	1437	107	9254	553	0
I, RANDOM	10	17	40	22	0
I, REVLEX	57	1475	1009	1249	0
ITS, LEX	1437	107	9254	553	0
ITS, RANDOM	16	15	33	54	0
ITS, REVLEX	57	1475	1009	1249	0
S, LEX	96	87	302	178	0
S, RANDOM	4	34	16	41	0
S, REVLEX	105	421	235	428	0
STI, LEX	96	87	302	178	0
STI, RANDOM	16	20	37	41	0
STI, REVLEX	105	421	235	428	0

Files asyoulik.txt, cp.html, fields.c, ptt5, lcet10.txt, and sum are not shown because no local search steps were needed, as one of the initial orderings was already better than the randomly sampled orderings without the need for search. I=INSERT, S=SWAP, ItS=INSERT then SWAP, StI=SWAP then INSERT

data which was not run to completion due to prohibitive runtimes (Sect. 5.1). However, the trend described also holds with the completed data for sum.

When completing the search to a local minimum, INSERT and INSERT-first methods perform slightly better than methods involving SWAP, however the INSERT and INSERT-first methods may take prohibitively long compared to SWAP, as they search a wider neighborhood, for very little gain.

6 Conclusion and future work

The BWT is an important string transformation, enabling a concise, searchable representation of a string. It relies on ordering the characters in the string and this is usually assumed to be ASCII ordering, without exploring whether alternatives might be more effective.

We studied heuristics for the computationally hard RLBWT Alphabet Ordering Problem which takes a string s of length n over an alphabet Σ of size σ and seeks an ordering of Σ for $RLE(BWT(s, \Sigma))$ that minimizes $|RLE(BWT(s, \Sigma))|$. We have performed extensive benchmarking using files from the Canterbury corpus and implemented the experimentation on Super Computing Wales HPC. We started by inspecting a large sample space of 240,000 randomly sampled alphabet orderings and found only limited improvement over the ASCII ordering. This motivated searching local neighborhoods to improve fitness - this was achieved using a First-Improvement algorithm.

Table 6 Number of successive improvements obtained with two local search algorithms compared to random sampling for ASCII ordering

File	Random sampling	`Swap, Lex`	`Insert, Lex`
`alice29.txt`	14 (240 K)	231 (205.84 K)	319 (519.2 K)
`asyoulik.txt`	11 (240 K)	218 (125.32 K)	310 (399.06 K)
`cp.html`	17 (240 K)	176 (242.68 K)	241 (491.92 K)
`fields.c`	10 (240 K)	157 (249.31 K)	169 (496.97 K)
`grammar.lsp`	11 (240K)	67 (65.54 K)	86 (161.88 K)
`lcet10.txt`	11 (240 K)	259 (340.97 K)	380 (934.5 K)
`plrabn12.txt`	15 (240 K)	383 (435.92 K)	457 (929.79 K)
`ptt5`	12 (240 K)	352 (1.25 M)	549 (3.59 M)
`sum`	10 (240 K)	452 (4.03 M)	561 (10 M)
`xargs.1`	13 (240 K)	70 (63.98 K)	102 (192.87 K)
average number of updates	12.4	236.5	317.4

Total number of steps are within brackets. Random sampling has created on average 12.4 updates across the 10 files while the theoretical mean number of updates for random sampling is approximately 12.38 for 240K samples (See Sect. 4.1). In contrast, local search algorithms deliver significantly more updates to the compression when compared to random sampling

Various initializations have been applied to the test files to attempt to speed up the search which include: ASCII order, letter frequency and order of appearance, and a hand-tuned ordering given by Chapin and Tate (1998). Jumping around the complex landscape was implemented with neighborhood search using SWAP and INSERT operators as well as combinations of these operators. Additionally varying the neighbors of an alphabet ordering has been explored, searching them lexicographically, reverse-lexicographically, and randomly. Overall, we inspected a combination of 9 initializations, 4 operators, and 3 neighborhood search methods, giving a total of 108 algorithm configurations.

The number of search steps needed to outperform the best result of random sampling was found to be relatively few and could be computed in seconds, but quickly increased for achieving a local minimum. Indeed, we demonstrated that reaching a similar number of improvements with random sampling would require investigating a much larger number of random samples (e.g., $> 10^{100}$), which is not feasible to compute.

The chosen initial alphabet order was found to influence the best solution that can be found, and we demonstrated this using the SWAP operator. While we observed variation in the initial ordering which achieved the best fitness within a time limit, there is not necessarily one best initial ordering. However, all initializations exhibit a clear decrease in file size followed by a large number of steps until the local minimum is reached.

We observed that the random neighbor ordering methods perform well in the early stages of the search, and while not the best they remain competitive overall. INSERT and INSERT-first methods are slower than SWAP and SWAP-first methods to reach a local minimum but will usually achieve a better compression. Although our empirical

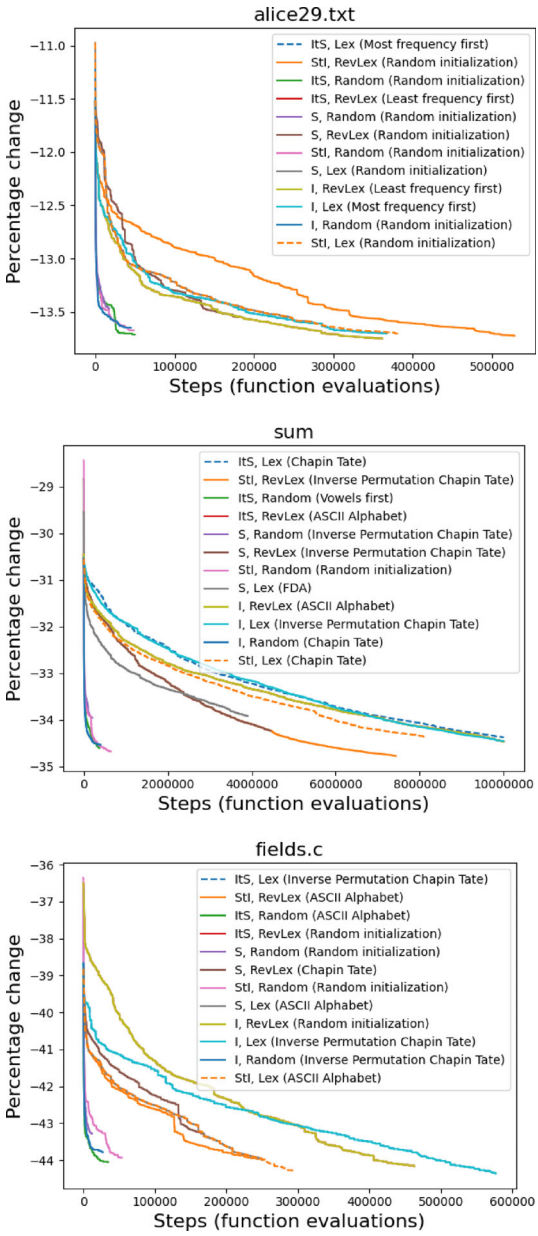

Fig. 6 The best initialization at the minimum for corpus texts `alice29.txt`, `sum`, and `fields.c` over time for different neighborhood search methods

evidence shows that local search is indeed effective for improving the RLBWT, and trade-offs occur, nonetheless we are still able to recommend a time-limited local search using SWAP with Random neighborhood exploration to improve rapidly upon the ASCII ordering. If more computational time is available to explore a better ordering

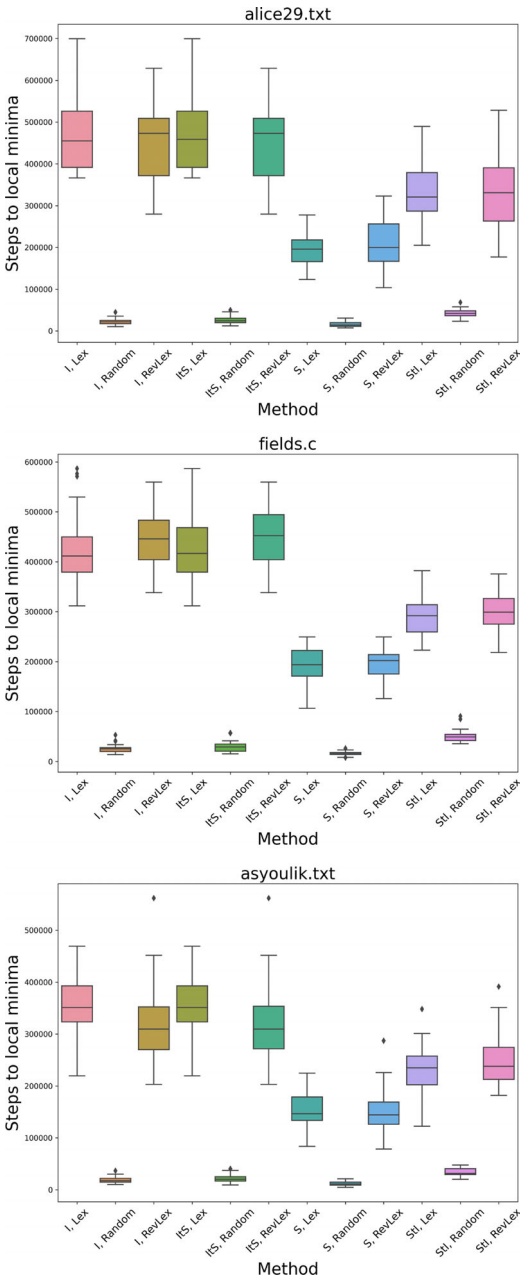

Fig. 7 Number of steps taken to find a local minimum for `alice29.txt`, `fields.c`, and `asyoulik.txt` for each neighborhood method in local search. The variation in each box plot shows the difference made by distinct initializations

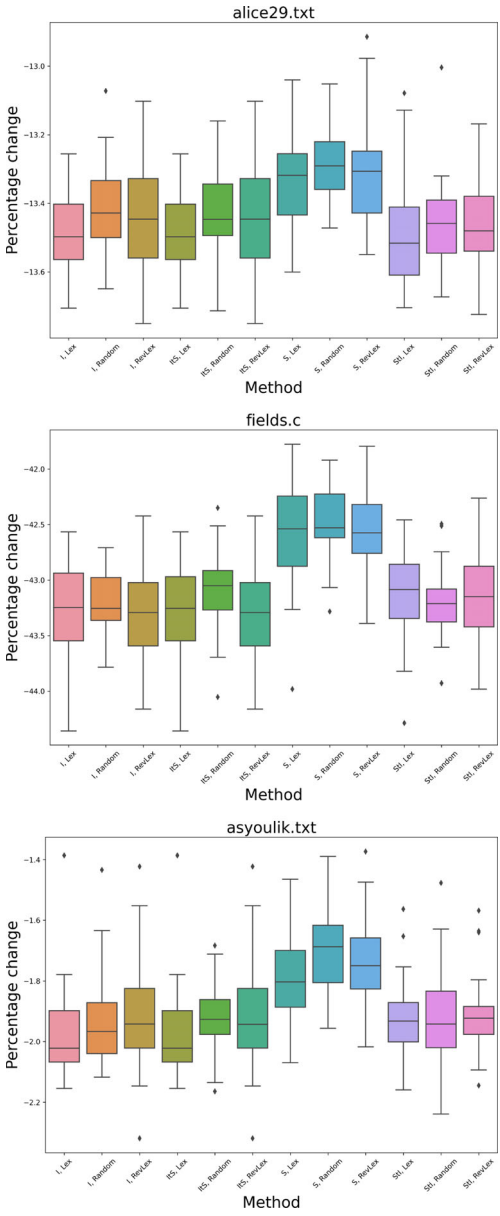

Fig. 8 Percentage change in compression for `alice29.txt`, `fields.c`, and `asyoulik.txt` for each neighborhood method in local search. The variation in each box plot shows the difference made by distinct initializations

then we recommend a local search using INSERT THEN SWAP with Lexicographic neighborhood exploration.

We found it interesting that the ASCII ordering performs very well when compared to a random order. Also, it works relatively well as an initialization for the local search. The reasons for ASCII performing well are hard to speculate, though we suspect that ASCII commonly appears in the lower half of the search space. We found that the fitness

of randomly sampled alphabet orderings is not normally distributed but instead the distribution has long tails towards the best and worst solutions, which suggests that only a few orderings are relatively good compared with other orderings in the search space. Further exploration of the search space, exhaustively, for large alphabet sizes would be required to fully understand the space. Our current research involves visualizing different alphabet orderings and their effect on the Burrows–Wheeler Matrix with the hope of better understanding such a search space.

If limited time is available we recommend the Chapin Tate ordering to start the search, or a random order to initialize a longer search. However, there is no clear best initialization suiting different files considered in the corpus.

This is a difficult problem in theory and we have now demonstrated that this is a challenging problem in practice for the range of files in the Canterbury Corpus, with no simple winning strategy. We have demonstrated that navigating trade-offs can be worthwhile for enhancing compressibility. Our local search performs much better (faster convergence, better fitness) than random sampling and is useful even when computational time is limited.

In future work we intend to investigate further what constitutes a good alphabet ordering, the effect that different changes to an ordering can have on the transformed string, what factors contribute to the quality of an ordering for a given string and why some orderings perform better than others, in relation to the type of data (for instance natural language versus other structured data).

We want to determine which characters can be moved to benefit the search and to use this knowledge to inspire new and more specific local search operators. This may include using operators which re-order multiple characters at a time or incorporation of various crossover operators. In addition different encoding methods for RLE may give better results and should be investigated.

Acknowledgements We acknowledge the support of the Supercomputing Wales project, which is part-funded by the European Regional Development Fund (ERDF) via Welsh Government.

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Lily Major. All authors wrote, read, and approved the final manuscript.

Funding This work is supported by the UKRI AIMLAC CDT, http://cdt-aimlac.org, grant no. EP/S023992/1, and was part-funded by the European Regional Development Fund through the Welsh Government, grant 80761-AU-137 (West).

Data Availability Code is available in our repository https://github.com/jam86/Heuristics-for-the-Run-length-Encoded-Burrows--Wheeler-Transform-Alphabet-Ordering-Problem, and https://doi.org/10.5281/zenodo.8139504. Data from our experiments is available at https://doi.org/10.5281/zenodo.8139367.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer, New York (2008). https://doi.org/10.1007/978-0-387-78909-5

Allen, M., Poggiali, D., Whitaker, K., Marshall, T., van Langen, J., Kievit, R.: Raincloud plots: a multi-platform tool for robust data visualization. Wellcome Open Res. **4**, 63 (2021). https://doi.org/10.12688/wellcomeopenres.15191.2

Arnold, R., Bell, T.: A corpus for the evaluation of lossless compression algorithms. In: Proceedings DCC '97. Data Compression Conference, (pp. 201–210) (1997). https://doi.org/10.1109/DCC.1997.582019

Bentley, J.W., Gibney, D., Thankachan, S.V.: On the complexity of BWT-runs minimization via alphabet reordering. In: Grandoni, F., Herman, G., Sanders, P. (Eds.) 28th Annual European Symposium on Algorithms (ESA 2020), (Vol. 173, pp. 15:1–15:13). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ESA.2020.15

Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. (Tech. Rep.). Palo Alto: Digital Systems Research Center (1994)

Cazaux, B., Rivals, E.: Linking BWT and XBW via Aho-Corasick automaton: applications to run-length encoding. In: Pisanti, N., Pissis, S.P. (Eds.) 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019), (Vol. 128, pp. 24:1–24:20). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2019). https://doi.org/10.4230/LIPIcs.CPM.2019.24

Chapin, B., Tate, S.R.: Higher compression from the Burrows–Wheeler transform by modified sorting. In: Data Compression Conference, DCC 1998, p. 532, (1998)

Cox, A.J., Bauer, M.J., Jakobi, T., Rosone, G.: Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. Bioinformatics **28**(11), 1415–1419 (2012). https://doi.org/10.1093/bioinformatics/bts173

Daykin, J.W., Groult, R., Guesnet, Y., Lecroq, T., Lefebvre, A., Léonard, M., Prieur-Gaston, Élise.: Binary block order Rouen transform. Theoret. Comput. Sci. **656**, 118–134 (2016). https://doi.org/10.1016/j.tcs.2016.05.028

Daykin, J.W., Smyth, W.F.: A bijective variant of the Burrows-Wheeler transform using V-order. Theoret. Comput. Sci. **531**, 77–89 (2014)

Daykin, J.W., Watson, B.: Indeterminate string factorizations and degenerate text transformations. Math. Comput. Sci. **11**(2), 209–218 (2017). https://doi.org/10.1007/s11786-016-0285-x

Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing, 2nd edn. Springer, Berlin (2015)

Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space. J. ACM (JACM) **67**(1), 1–54 (2020). https://doi.org/10.1145/3375890

Giancarlo, R., Manzini, G., Restivo, A., Rosone, G., Sciortino, M.: Block sorting-based transformations on words: beyond the magic BWT. In: Hoshi, M., Seki, S. (eds.) Developments in Language Theory, pp. 1–17. Springer International Publishing, Cham (2018)

Giancarlo, R., Manzini, G., Restivo, A., Rosone, G., Sciortino, M.: The alternating BWT: an algorithmic perspective. Theoret. Comput. Sci. **812**, 230–243 (2020)

Giancarlo, R., Manzini, G., Restivo, A., Rosone, G., Sciortino, M.: A new class of string transformations for compressed text indexing. Inf. Comput. **294**, 105068 (2023). https://doi.org/10.1016/j.ic.2023.105068

Gibney, D.: Algorithms and lower bounds for ordering problems on strings. (Doctoral dissertation, University of Central Florida) (2021). https://stars.library.ucf.edu/etd2020/507/

Kempa, D., Kociumaka, T.: Dynamic suffix array with polylogarithmic queries and updates. In: Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing, (pp. 1657–1670). New York, NY, USA: Association for Computing Machinery (2022). https://doi.org/10.1145/3519935.3520061

Kim, D.K., Sim, J.S., Park, H., Park, K.: Constructing suffix arrays in linear time. J. Discr. Algorithms **3**(2), 126–142 (2005). https://doi.org/10.1016/j.jda.2004.08.019

Knuth, D.E.: The Art of Computer Programming/Volume 2, Seminumerical algorithms. (3rd ed.). Reading: Addison-Wesley (1998a)
Knuth, D.E.: The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching. Boston: Addison Wesley Longman Publishing Co., Inc (1998b)
Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. J. Discr. Algorithms **3**(2), 143–156 (2005). https://doi.org/10.1016/j.jda.2004.08.002
Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nat. Methods **9**, 357–359 (2012). https://doi.org/10.1038/nmeth.1923
Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows–Wheeler transform. Bioinformatics **25**(14), 1754–1760 (2009)
Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics **25**(15), 1966–1967 (2009)
Major, L., Clare, A., Daykin, J.W., Mora, B., Peña Gamboa, L.J., Zarges, C.: Evaluation of a permutation-based evolutionary framework for Lyndon factorizations. In: Bäck, T., et al. (Eds.) Parallel Problem Solving from Nature–PPSN XVI, Springer, Cham (2020)
Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. In: Apostolico, A., Crochemore, M., Park, K. (eds.) Combinatorial Pattern Matching, pp. 45–56. Springer, Berlin (2005)
Mantaci, S., Restivo, A., Rosone, G., Sciortino, M., Versari, L.: Measuring the clustering effect of BWT via RLE. Theoret. Comput. Sci. **698**, 79–87 (2017). https://doi.org/10.1016/j.tcs.2017.07.015
Nishimoto, T., Kanda, S., Tabei, Y.: An optimal-time RLBWT construction in bwt-runs bounded space. In: Bojanczyk, M., Merelli, E., Woodruff, D.P. (Eds.) 49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4–8, 2022, Paris, France (vol. 229, pp. 99:1–99:20). Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Comput. **60**(10), 1471–1484 (2011). https://doi.org/10.1109/TC.2010.188
Petersen, H.: On the language of primitive words. Theoret. Comput. Sci. **161**(1), 141–156 (1996). https://doi.org/10.1016/0304-3975(95)00098-4
Pibiri, G.E.: On weighted k-mer dictionaries. Algorithms Mol. Biol. **18**(1), 1–20 (2023)
Rossi, M., Oliva, M., Bonizzoni, P., Langmead, B., Gagie, T., Boucher, C.: Finding maximal exact matches using the r-index. J. Comput. Biol. **29**(2), 188–194 (2022)
Seward, J.: bzip2 and libbzip2. (1996). http://sourceware.org/bzip2/
Sirén, J., Välimäki, N., Mäkinen, V., Navarro, G.: Run-length compressed indexes are superior for highly repetitive sequence collections. In: Amir, A., Turpin, A., Moffat, A. (eds.) String Processing and Information Retrieval, pp. 164–175. Springer, Berlin (2008)
Syahrul, E., Dubois, J., Vajnovszki, V., Saidani, T., Atri, M.: Lossless image compression using Burrows Wheeler Transform (methods and techniques). In: 2008 IEEE International Conference on Signal Image Technology and Internet Based Systems, pp. 338–343 (2008). https://doi.org/10.1109/SITIS.2008.40
Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D. SciPy., 1.0 Contributors,: SciPy 1.0: fundamental algorithms for scientific computing in python. Nat. Methods **17**, 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2