TYPE Methods
PUBLISHED 23 January 2025
DOI 10.3389/fbinf.2024.1489704

OPEN ACCESS

EDITED BY
Ali Kadhum Idrees,
University of Babylon, Iraq

REVIEWED BY
Soham Sengupta,
St. Jude Children's Research Hospital,
United States
Bhavika Mam,
Independent Researcher, Palo Alto, CA,
United States

*CORRESPONDENCE
Anas Al-okaily,
aa.12682@khcc.jo

RECEIVED 01 September 2024
ACCEPTED 26 December 2024
PUBLISHED 23 January 2025

CITATION
Al-okaily A and Tbakhi A (2025) A novel lossless encoding algorithm for data compression–genomics data as an exemplar.
Front. Bioinform. 4:1489704.
doi: 10.3389/fbinf.2024.1489704

A novel lossless encoding algorithm for data compression–genomics data as an exemplar

Anas Al-okaily[1]* and Abdelghani Tbakhi[2]

[1]Department of Cell Therapy and Applied Genomics, King Hussein Cancer Center, Amman, Jordan,
[2]Department of Pathology and Molecular Medicine, McMaster University, Hamilton, ON, Canada

Data compression is a challenging and increasingly important problem. As the amount of data generated daily continues to increase, efficient transmission and storage have never been more critical. In this study, a novel encoding algorithm is proposed, motivated by the compression of DNA data and associated characteristics. The proposed algorithm follows a divide-and-conquer approach by scanning the whole genome, classifying subsequences based on similarities in their content, and binning similar subsequences together. The data is then compressed into each bin independently. This approach is different than the currently known approaches: entropy, dictionary, predictive, or transform-based methods. Proof-of-concept performance was evaluated using a benchmark dataset with seventeen genomes ranging in size from kilobytes to gigabytes. The results showed a considerable improvement in the compression of each genome, preserving several megabytes compared to state-of-the-art tools. Moreover, the algorithm can be applied to the compression of other data types include mainly text, numbers, images, audio, and video which are being generated daily and unprecedentedly in massive volumes.

KEYWORDS

compression, Huffman encoding, LZ, genomics, BWT

1 Introduction

The importance of data compression, a fundamental problem in computer science, information theory, and coding theory, continues to increase as global data quantities expand rapidly. The primary goal of compression is to reduce the size of data for subsequent storage or transmission. There are two common types of compression algorithms: lossless and lossy. Lossless algorithms guarantee exact restoration of the original data, whereas lossy algorithms do not. Such losses are caused, for instance, by the exclusion of unnecessary information, such as metadata in video or audio that will not be observed by users.

Data exist in different formats including text, numbers, images, audio, and video. Several coding algorithms and the corresponding variants have been developed for textual data, the main focus of this paper. This includes the Huffman Huffman (1952), Shannon Shannon (2001), Shannon-Fano Fano (1949), Shannon-Fano-Elias Cover (1999), Lempel-Ziv (LZ77) Ziv and Lempel (1977), Burrows-Wheeler transform Burrows and Wheeler (1994) and Tunstall (1968) algorithms. The Huffman algorithm includes several variants: minimum-variance Huffman, canonical Huffman, length-limited Huffman, nonbinary Huffman, adaptive Huffman, Faller-Gallager-Knuth (an adaptive Huffman) Knuth (1985),

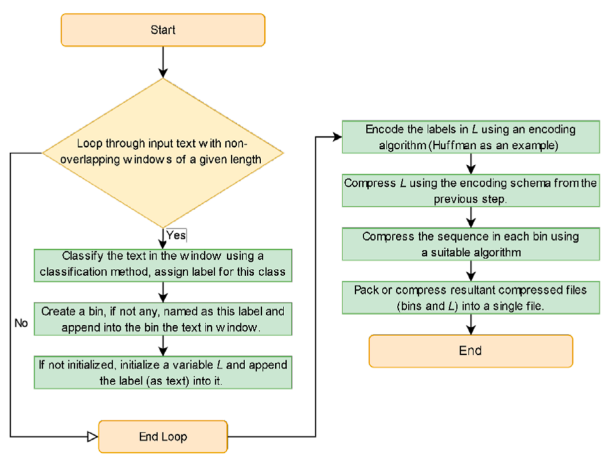
Start
Loop through input text with non-overlapping windows of a given length
Yes
Classify the text in the window using a classification method, assign label for this class
Create a bin, if not any, named as this label and append into the bin the text in window.
If not initialized, initialize a variable *L* and append the label (as text) into it.
No
End Loop
Encode the labels in *L* using an encoding algorithm (Huffman as an example)
Compress *L* using the encoding schema from the previous step.
Compress the sequence in each bin using a suitable algorithm
Pack or compress resultant compressed files (bins and *L*) into a single file.
End

FIGURE 1
A flowchart illustrating the compression algorithm

and Vitter (an adaptive Huffman) Vitter (1987). The LZ algorithm also includes several variants, such as LZ78 Ziv and Lempel (1978), Lempel-Ziv-Welch (LZW) Welch (1984), Lempel-Ziv-Stac (LZS) Friend (2004), Lempel-Ziv-Oberhumer (LZO) Oberhumer (2008), Lempel-Ziv-Storer-Szymanski (LZSS) Storer and Szymanski (1982), Lempel–Ziv-Ross-Williams (LZRW) Williams (1991), and the Lempel–Ziv–Markov chain algorithm (LZMA) Ranganathan and Henriques (1993). Additional techniques involve arithmetic encoding Langdon (1984), range encoding Martín (1979), move-to-front encoding (also referred to as symbol ranking encoding) Ryabko (1980); Bentley et al. (1986), run-length encoding Capon (1959), delta encoding, unary encoding, context tree weighting encoding Willems et al. (1995), prediction by partial matching Cleary and Witten (1984), context mixing Mahoney (2005), asymmetric numeral systems (also called asymmetric binary encoding) Duda (2013), length index preserving transform Awan and Mukherjee (2001), and dynamic Markov encoding Cormack and Horspool (1987).

Compression algorithms can be classified based on the methodology used in the algorithm, such as entropy, dictionary, predictive, and transform-based methods. These methods have been described extensively in several recent studies Gopinath and Ravisankar (2020); Kavitha (2016); Uthayakumar et al. (2018); Kodituwakku and Amarasinghe (2010), however, a brief description for each method is provided in the Supplementary Material.

Genomics (DNA/RNA) data is a type of textual information with several unique characteristics. First, the alphabet consists only of A, C, G, and T characters representing the four nucleotides: adenine, cytosine, guanine, and thymine, respectively. Second, DNA data contain repeat sequences and palindromes. Third, the size of genomics data can be very large, relative to most media files. The human genome, for instance, consists of more than three billion nucleotides (specifically 3,272,116,950 bases (https://www.ncbi.nlm.nih.gov/grc/human/data?asm=GRCh38.p13) requiring more than 3 gigabytes of storage), and the sequencing is typically conducted with high depth (30–100x) to sequence the same region several times for more accurate reading. As such, sequencing genomic data (especially for humans) is currently being performed for research and diagnostic purposes in daily basis (the number of bases sequenced from December 1982 through August 2024 was 29,643,594,176,326 (https://www.ncbi.nlm.nih.gov/genbank/statistics/)). Several algorithms have been developed to compress these data, which can be divided into vertical and horizontal techniques Grumbach and Tahi (1994). Vertical mode algorithms utilize a reference genome/source, while horizontal mode algorithms are reference-free.

Genomic data are stored in different formats, including FASTA Lipman and Pearson (1985), FASTQ Cock et al. (2010), and SAM Li et al. (2009), with FASTA being the most common and also the primary focus of this paper. Several comparative studies for compressing FASTA files have been published in recent years Kryukov et al. (2020); Hosseini et al. (2016); Mansouri et al. (2020); Bakr and Sharawi, 2013; Jahaan et al. (2017). Genomic sequences typically consist of millions or billions of sequenced reads, with lengths in the hundreds, stored with the quality of each base in a primarily FASTQ format. A common DNA data processing strategy involves aligning the sequenced reads with a reference genome. The output is the reads themselves, with their base qualities and alignment results for each read, stored in a SAM format. Surveys of compression tools for SAM and FASTQ data are available in the literature Bonfield and Mahoney (2013); Hosseini et al. (2016).

The small alphabet found in DNA data simplifies the compression process. However, the problem remains challenging due to the discrete, uniform distribution (frequencies) of bases in DNA data. Efficient compression relies mainly on repetitiveness in the data and encoding as few characters/words as possible, since encoding more characters costs more bits-per-character. If the characters are uniformly distributed in the text, their repetitions will also be distributed uniformly and encoding only a fraction of them (to decrease the bits-per-character) will lead to low compression outcomes. The application of Huffman encoding, for instance, produces 2-bit assignments for each base. The algorithm will then produce an inefficient/suboptimal compression result that does not utilize repetitions found in the DNA data. Motivated by this issue, we introduce in this work a lossless and reference-free encoding algorithm.

2 Methods

The following observations can be inferred from a careful analysis of DNA. First, many regional (local) sub-sequences (assume a length of 100 bases) contain non-uniform or skewed distributions. Second, similar sub-sequences (which provide better compression results if encoded together) are often distant from each other. This distance is typically longer than the length of sliding windows (usually in kilobases/kilobytes) commonly used in algorithms such as LZ or far apart from previous sequence/symbol used in prediction models such as context weighting tree, predictive partial matching, or dynamic Markov compression. Even if these sequences are located within the same sliding window or previous sequences/symbols, they are often sufficiently distant from each other, which leads to inefficient compression and encoding. These two observations were the motivation behind the design of the following encoding algorithm.

```
1:  Initialize a sequence named as $L$ where labels of bins will b stored.
2:  loop through $T$ with non-overlapping windows of length $w$, where at each window:
3:      Classify the sequence in the window using a classification method. In this work, the method is based on
        content of the sequence and Huffman tree (described more in section 2.3). Once classified, create a unique
        label $l$ that denotes the class of the sequence.
4:      if there is no bin with label $l$ then
5:          Create a bin labeled as $l$ and initialize the sequence of this bin to empty string.
6:      Concatenate sequence of the window with the sequence of the bin with label $l$.
7:      Concatenate $l$ with $L$.
8:  Encode all labels collected in $L$ using Huffman encoding. These labels can be set dynamically, designed in
    advance, or can be set by the user or the implementer.
9:  Compress $L$ using the encoding schema of the previous step. The time cost of this step is $O(\frac{t}{w})$.
10: Compress the sequence in each bin using an algorithm suitable for the content and/or the label of the bin.
11: Pack or archive, with or without further compression, all resultant compressed files (bins and $L$) together into a
    single file.
12: ▷ The time cost for the loop in steps 2 is $O(\frac{t}{w}c)$ where $c$ is the cost of classifying each non-overlap subsequence
    of length $w$ and $t$ is the length of the text. If $c \leq w$, then the total cost for this step can be $O(t)$.  ◁
13: ▷ The need of L in step 7 is to store the queue/order of the sequences' labels. This order will be needed during
    the decompression process to restore the sequences (that are distributed into the bins) to their original
    placements in the input data. The expected length of L is $\frac{t}{w}b$ where $b$ is the max length of any label of any
    bin. Now, as the number of bins is selected and designed in advanced and must be much less than $\Sigma^w$, so
    that the max length of any label of any bin will be much less and no more than $w$. As $w$ value has to be large
    by design, this guarantees that the total length of L is no more than $t$.  ◁
14: ▷ The time cost for step 8 is $O(BlogB)$ where B is the number of labels in L and equal to $\frac{t}{w}$. So, the total cost
    is $O(\frac{t}{w}logt)$. If $logt \leq w$, then the total cost van be $O(t)$.  ◁
```

Algorithm 1. Compression algorithm.

```
1: if all bins and $L$ were compressed/archived into a single file then
2:     Decompress/Unarchive the single file.
3: Decompress each bins file and $L$ with the compression tool that was used to compress each of them.
4: for each bin do
5:     Initialize a variable named as counter
6: loop sequentially through labels in $L$, where at each label $l$
7:     Extract a sequence of length $w$ from bin with label $l$. Extraction of the sequence must start from the position
       equal to the value of counter of bin with label $l$.
8:     Increment counter (of bin with label $l$) by the value of $w$ plus 1.
9:     Output the extracted sequence to the output stream.
```

Algorithm 2. Decompression algorithm.

2.1 Compression algorithm

Given a string T of length t, containing characters from a fixed alphabet of length Σ, and a window-length w, the description of the proposed algorithm is stated in algorithm 1 and is illustrated in Figure 1.

2.2 Decompression algorithm

Decompression algorithm is the inverse of compression algorithm and is described in algorithm 2. As the total length of the sequences in all bin is $O(t)$ and the total length of L is also $O(t)$ (as described in algorithm 1), the decompression of all bins will cost no more than t. Hence, the time and memory costs of decompression all bins and L is linear.

This algorithm can be applied not only to DNA or textual data, but to archiving processes and other data types namely numbers, images (binning for instance similar pixels instead of similar subsequences as in text), audio (binning for instance similar subbands/frequency-ranges), and video (binning for instance similar images)). Sub-binning or nested-binning processes can also be applied.

This design facilitates organizing and sorting the input data using a divide-and-conquer method by creating bins for similar data and encodes/compresses data in the same bin that are better if compressed together, to achieve better compression results with a minor increase in time costs. In the case of more random/divergent input data, which is the common case, this algorithm avoid relying on a single encoding or compression technique (as in entropy methods), being dependent on the previous sequences and their randomness (as in prediction methods), requiring construction of dictionaries dependent also on the previous sequences and their randomness (as in dictionary methods), or potentially introducing inefficient transformation due to the randomness of the data (as in transform methods). In contrast, the proposed algorithm divides the data into groups of similar segments, regardless of their position in the original sequence, which decreases the randomness in the data and contributes in organizing the input data by binning the similar data together to ultimately handling the compression process more efficiently.

Note that the continued application of sub-binning processes will eventually reduce the randomness/divergence of the data and improve the compression results, by obtaining data that are optimal or suboptimal for compression. These processes will require additional time costs, but these costs will still be practical at low sub-binning depth and feasible at high sub-binning depths, especially for small data or the compression of large data for archiving. Therefore, sub-binning will eventually provide more control, organization, and possibly a deterministic solution to encoding and compression problems. Further analysis and investigations are also provided in the Supplementary Material.

The encoding algorithm is named after both authors and Pramod K. Srivastava (Professor in the Department of Immunology, University of Connecticut School of Medicine) for honoring him as he was an advisor to the first author. As such, it is named the Okaily-Srivastava-Tbakhi (*OST*) algorithm.

2.3 OST-DNA

This is the first implementation of the OST algorithm which accepts DNA data as input which can be denoted as OST-DNA. Bin labels are computed using a Huffman tree encoding. The reason for selecting Huffman algorithm since the label of larger bin must be more frequent in L, hence encode this label with shorter codes (while the label of smaller bins with longer codes).

For example, if the Huffman tree for a subsequence produces the following encoding schema: G:0, A:10, T:110, and C:111, then the label will be GATC_1233 label (1 indicates G is encoded using 1 bit, A using 2 bits, and so on). The number of bits used to encode a character gives a general indication of its frequency compared to the other characters. The number of bins can be reduced by decreasing the label length as follows. To produce a label length of 1, we used the first base of the Huffman tree and its bit length. As such, the above Huffman encoding schema will be represented by G_1. If the bin label length is 2, then the label will be GA_12. This clearly decreases the number of labels, but at the cost of decreasing the similarity among sequences in the same bin therefore their compression. Note that this classification method (bin labeling) is suitable for DNA data as its alphabet is small. For data with larger alphabets, same or other classification methods might be sought such as binning sequences that contain some letter/s most frequently.

As the windows do not overlap, each base in T will be read in $O(1)$ time. The cost of constructing a Huffman tree for a subsequence is then $O(\Sigma log\Sigma)$, requiring the construction of $O(\frac{t}{w})$ Huffman trees for all non-overlap subsequences in T. The total cost hence is $O(\frac{t}{w}\Sigma log\Sigma)$. In order to allow for the acquisition of non-uniform distributions for the characters in Σ (the pigeonhole principal), the value of w must be larger than that of $\Sigma log\Sigma$, noting that Σ is a constant. As such, the total cost of the compression process of OST-DNA can be $O(t)$.

Since the value of w is fixed in this version of OST-DNA, Huffman trees are constructed once for each window sequence. In the case of a variable w where the window will be extended until the sequence label matches a bin label, it is not efficient to calculate Huffman trees for the entire sequence at every extension, hence adaptive Huffman trees can be applied instead.

Generally, the compressed bin files and L can be collected into a single file using an archiver which could perform further compression. However, this process was omitted in this study to demonstrate the efficiency of the OST algorithm without any further compression that may be produced by the archiving process.

3 Results

We implemented OST-DNA using the python language. We used the same dataset applied to another benchmark Kryukov et al. (2020) in order to test and compare the compression results using OST-DNA with the tools listed in Supplementary Table S1 in Supplementary Material. The dataset consists of seventeen genomes, as shown in Supplementary Tables S2, S3 in Supplemental Methods, ranging in size from 50 KB to 13 GB with a total size of 16,773.88 MB, orgin from different species (virus, bacteria, protist, fungus, algae, animal, plant), and were sequenced using Illumina, 454, SOLID, PacBio, Sanger dideoxy, or mixed technolgies.

The following preprocessing steps were applied to each tested genome. All new lines, header lines, lowercase bases, and bases not identified as A, C, G, T, or N, were removed. This produced a one-line sequence for each genome, containing only the four capitalized DNA bases and the letter "N". Character "N″ represents uncalled/undetermined base during sequencing or assembling process. Assembled genomes may contain also bases with lowercase that represent soft-masked sequences. In compression process, these sequences are converted to capital case with recording their start/end coordinates so that during decompressing process their original case is restored. As the tested genomes contain low number of soft-masked sequences and as this study is using genomics data for testing purposes, these sequences were just removed from the genomes. The python script used to perform these preprocessing steps and the size of each genome, before and after applying the script, are provided in Supplementary Table S4 in Supplementary Material.

Compression ratio was the primary metric used for evaluating the performance of the proposed algorithm. It is equal to the size of the compressed file divided by the size of the uncompressed (original) file. The original files in this study are the one-line genome files. Other metrics include compression time (seconds), decompression time (seconds), compression speed (the size of the uncompressed file in MB divided by the compression time in seconds-MB/s), and the decompression speed (the size of the uncompressed file in MB divided by the decompression time in seconds-MB/s).

The following tools were selected as they are common tools for compressing textual data and implementing one or more compression algorithms available in the literature. This is meant to test compressing the resultant bins using all available encoding

TABLE 1 Compression performance for each common tool cumulatively over all tested genomes.

Tool	Compression ratio (%)	Compression time (s)	Decompression time (s)	Compression speed (MB/s)	Decompression speed (MB/s)
bcm	20.6217	4,071	3,728	4.1203	0.9279
blzpack	37.2279	227	149	73.8937	41.9098
brotli	16.7848	62,659	103	0.2677	27.3346
bsc	20.1670	2,369	68	7.0806	49.7469
bzip2	24.8765	2,474	1,169	6.7801	3.5695
compress	25.3977	443	168	37.8643	25.3582
freeze	27.4616	6,078	233	2.7598	19.7698
gzip	26.9031	4,211	171	3.9833	26.3900
hook	21.3395	7,803	8,074	2.1497	0.4433
Huffman-codec	27.4015	1,152	401	14.5607	11.4621
lizard	34.8186	11,449	41	1.4651	142.4494
lrzip	14.6446	21,589	378	0.7770	6.4986
lz4	52.7757	161	73	104.1856	121.2677
lzfse	29.3101	1,118	130	15.0035	37.8187
lzip	17.0353	20,079	304	0.8354	9.3996
lzop	47.6212	152	106	110.3545	75.3578
LzTurbo	28.4807	157	46	106.8400	103.8548
ppm	23.8049	5,020	6,314	3.3414	0.6324
qzip	41.9873	1,556	126	10.7801	55.8960
rans	24.0431	144	91	116.4853	44.3182
rzip	24.9315	2,515	1,279	6.6695	3.2697
snzip	45.5241	159	73	105.4961	104.6048
srank	41.2318	794	778	21.1258	8.8897
xz	17.0381	18,666	266	0.8986	10.7442
zip	26.9031	4,165	173	4.0273	26.0849
zlib	26.9187	4,278	117	3.9210	38.5924
zpipe	26.9187	4,283	106	3.9164	42.5973
zstd	26.7482	251	75	66.8282	59.8227

The size of all genomes (in one-line format) is 16,773.88 MB. The tools cmix, lzb, and Nakamichi could not compress large genomes in reasonable time so their cumulative performance could not be presented.

algorithms. The tools namely are: bcm, blzpack, brotli, bsc, bzip2, cmix, compress, freeze, gzip, hook, Huffman-codec, lizard, lrzip, lz4, lzb, lzfse, lzip, lzop, lzturbo, Nakamichi, ppm, qzip, rans static, rzip, snzip, srank, xz, zlib, zip, zpipe, and zstd. Description of each tool is provided in Supplementary Table S1 in Supplementary Material. The cumulative compression results (for all one-line genomes) are

TABLE 2 Compression performance for best window-length and label-length of each of the seven OST-DNA versions cumulatively over all tested genomes.

Tool	Window length	Label length	Compression ratio (%)	Compression time (s)	Decompression time (s)	Compression speed (MB/s)	Decompression speed (MB/s)
bcm	-	-	20.6217	4,071	3,728	4.1203	0.9279
OST-DNA-bcm	250	4	20.1603	12,026	4,388	1.3948	0.7707
brotli	-	-	16.7848	62,659	103	0.2677	27.3346
OST-DNA-brotli	750	4	15.9477	72,315	318	0.2320	8.4121
bsc	-	-	20.1670	2,369	68	7.0806	49.7469
OST-DNA-bsc	250	5	19.7688	10,355	2,160	1.6199	1.5352
bzip2	-	-	24.8765	2,474	1,169	6.7801	3.5695
OST-DNA-bzip2	250	2	24.6689	10,132	1,230	1.6555	3.3642
lrzip	-	-	14.6446	21,589	378	0.7770	6.4986
OST-DNA-lrzip	1,000	1	14.5728	31,911	432	0.5256	5.6584
lzip	-	-	17.0353	20,079	304	0.8354	9.3996
OST-DNA-lzip	750	2	16.8067	30,691	472	0.5465	5.9728
xz	-	-	17.0381	18,666	266	0.8986	10.7442
OST-DNA-xz	750	2	16.7898	29,098	393	0.5765	7.1662

provided in Table 1, while the compression results for each one-line genome are listed in Supplementary Table S6. The most efficient tools in terms of compression ratio were lrzip (saved 14,317.40 MB out of 16,773.88 MB), brotli (13,958.42 MB), lzip (13,916.39 MB), xz (13,915.92 MB), bsc (13,391.09 MB), and bcm (13,314.83 MB). In addition, comparing the results of the commonly used tools (bzip2 and gzip) indicated bzip2 was better, saving 12,601.12 MB.

Next, seven versions of OST-DNA were implemented. In each version, one of the seven most efficient tools (bcm, brotli, bsc, lrzip, lzip, xz, and bzip2) is used to compress the bins generated by the OST-DNA algorithm. The command used by each tool to compress the one-line genomes is the same used to compress the bins. Each of these seven versions was applied to each one-line genome. The compression and decompression commands used to run each tool are provided in Supplementary Table S5 in Supplementary Material. The default options for each tool were used to compress the one-line genomes. The same commands (default options) were used to compress the bins. No parallel processing was applied. If a tool apply parallel processing by default, the options were modified to be single/sequential processing. In addition, each of the seven versions were run over window lengths of 25, 50, 100, 125, 150, 250, 500, 750, 1,000, 2,500, 5,000, and 10,000 to investigate the compression results over each of these lengths. Lastly, each of the seven versions were run across label lengths of 1, 2, 3, 4, and 5 to investigate also the results over each of these lengths. The best result in terms of compression ratio over all pairs of window lengths and label lengths and cumulatively (over all 17 one-line genomes) achieved by each OST-DNA version is shown in Table 2. A comparison of the results produced by each OST-DNA version indicated OST-DNA-bcm saved an additional 77.38 MB compared to bcm, OST-DNA-brotli: 140.41 MB, OST-DNA-bsc: 66.79 MB, OST-DNA-bzip2: 34.83 MB, OST-DNA-lrzip: 12.05 MB, OST-DNA-lzip: 38.34 MB, and OST-DNA-xz: 41.65 MB. This demonstrates that the proposed algorithm can improve compression results compared to the corresponding standalone tools. Moreover, the tools that are based on LZ algorithm (OST-DNA-brotli, OST-DNA-lrzip, OST-DNA-lzip, and OST-DNA-xz which are dictionary based algorithms and perform better when the input data is more redundant and the redundancies are closer to each other as the case of the input data in the bins) performed better than the other tools (OST-DNA-bcm, OST-DNA-bsc, and OST-DNA-bzip2 which are based on block sorting and BWT).

The best tool in terms of compression ratio was lrzip, yet OST-DNA-lrzip saved an additional 12.05 MB more than lrzip. In terms of compression time, bsc was the fastest tool. OST-DNA-bsc could save an additional 66.79 MB more than bsc with a practical increase in the compression/decompression times (hence corresponding decrease in compression and decompression speeds). These increases are a result of the time needed for classifying and binning sequences during compression, as well as the need to collect and restore the original genome during decompression. However, they can be decreased significantly as follows. First, the OST-DNA script was not optimized for implementation as it is intended in this study to provide proof of concept. Additional improvements to the script can reduce both the compression and decompression times by increasing the corresponding speeds. In addition, parallel processing, which could further reduce run-time, was not applied during the binning, compression, or decompression steps of OST-DNA. Finally, fewer bins would lead to faster sequence labeling and longer window lengths could speed up both compression and decompression, with a trade-off in the compression ratio.

The compression results for OST-DNA using each of the seven tools for each one-line genome were also better than the results using the corresponding standalone tool. This can be found by comparing the compression results using each OST-DNA version with each window length and each label length for the one-line genomes, as shown in Supplementary Table S7 (the compression results for the standalone tools are provided in Table 1). This is justified due to the fact that if the subsequences that are similar/redundant are long distant from each other (for instance at the beginning and end of the input data), then they will be compressed together using OST algorithm as they will be binned together but this is not the case with other compression methodologies especially if the input data is larger and larger. So, the longer distant similar/redundant subsequences and larger input data, the better advantage for OST algorithm compared to other compression methodologies.

By analyzing the compression results for all OST-DNA versions, using different sequence label lengths and the adopted classification methods in this work (i.e., Huffman tree encoding schema), we found the most efficient results correlated with a window length of 250 to 1,000 bases. This is reasonable, as lengths shorter or longer than this will yield a uniform distribution of bases in the sequence. However, dynamic window lengths can be more practical and feasible given the additional costs for encoding the lengths. We found efficient label lengths to be 2 and 4. This is reasonable as increased label lengths produce more bins and more similarities among sequences in the same bin. Compression is more efficient when sequences in a bin are more similar. Supplementary Table S8 shows compression results for each version of OST-DNA for each window and each label length cumulatively applied to all one-line genomes. Further analysis at the bin level, rather than the genome level, is provided in Supplementary Table S9.

Compression results produced by applying each OST-DNA version to each bin, using the same window and label lengths but with different genomes, were considerably different (see Supplementary Table S10). This was not the case for bins produced using the same label length and same genome, but with different window lengths (see Supplementary Table S11). This means that sequences with the same label but from different genomes differed significantly (even though their labels were the same). This observation suggests the need to find a set of labels or labeling steps that could compress sequences from any source (genome) with similar efficiency, to improve the compression results further. In other words, sequences that share a label from this set would be compressed at a similar rate, regardless of the source (genome) from which they were derived. This set of labels could also be used better archival of multiple genomes.

The current version of OST-DNA compresses each bin using a specific tool. However, this is not optimal. Finding a tool that optimally compresses each bin, or a novel algorithm that is customized for efficient compression based on the bin content or label, could further improve the overall performance.

4 Discussion

Note that the aim of this implementation of the proposed algorithm is to proof-the-concept. Academic and commercial versions and after careful sophistication and customized methods will be sought in the near future.

The binning/bucketing approach was suggested to compress NGS sequencing reads as these reads must be overlapped due to the similarity in the sequenced genome or the amplification step in the sequencing process. OST algorithm on the other hand, propose an approach to compress a single genome/text. There are algorithms proposed to compress a single genome but they rely on compression-by-referencing approach which compress the genome based on the similarities with another public genome. While OST algorithm does not relay on any external resources and take advantage of the similarities inside the genome itself using classification-then-binning approach. Moreover, for general texts (general alphabets such english language) there is no reliable reference that can be used by the compression-by-referencing approach, while OST algorithm is still applicable for any general text.

Data availability statement

Source code of the seven OST-DNA tools are available at https://github.com/aalokaily/OST.

Author contributions

AA: Writing–original draft. AT: Writing–review and editing.

Funding

The author(s) declare that no financial support was received for the research, authorship, and/or publication of this article.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fbinf.2024.1489704/full#supplementary-material

References

Awan, F. S., and Mukherjee, A. (2001). “Lipt: a lossless text transform to improve compression,” in *Proceedings international Conference on information Technology: Coding and computing (IEEE)*, 452–460.

Bakr, N. S., and Sharawi, A. A. (2013). Dna lossless compression algorithms. *Am. J. Bioinforma. Res.* 3, 72–81. doi:10.5923/j.bioinformatics.20130303.04

Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. (1986). A locally adaptive data compression scheme. *Commun. ACM* 29, 320–330. doi:10.1145/5684.5688

Bonfield, J. K., and Mahoney, M. V. (2013). Compression of fastq and sam format sequencing data. *PloS one* 8, e59190. doi:10.1371/journal.pone.0059190

Burrows, M., and Wheeler, D. J. (1994). *A block-sorting lossless data compression algorithm*. Citeseer.

Capon, J. (1959). A probabilistic model for run-length coding of pictures. *IRE Trans. Inf. Theory* 5, 157–163. doi:10.1109/tit.1959.1057512

Cleary, J., and Witten, I. (1984). Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.* 32, 396–402. doi:10.1109/tcom.1984.1096090

Cock, P. J., Fields, C. J., Goto, N., Heuer, M. L., and Rice, P. M. (2010). The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids Res.* 38, 1767–1771. doi:10.1093/nar/gkp1137

Cormack, G. V., and Horspool, R. N. S. (1987). Data compression using dynamic markov modelling. *Comput. J.* 30, 541–550. doi:10.1093/comjnl/30.6.541

Cover, T. M. (1999). *Elements of information theory*. John Wiley and Sons.

Duda, J. (2013). Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. arXiv preprint arXiv:1311.2540

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Trans. Inf. theory* 21, 194–203. doi:10.1109/tit.1975.1055349

Fano, R. M. (1949). *The transmission of information*. Research Laboratory of Electronics: Massachusetts Institute of Technology.

Fraenkel, A. S., and Kleinb, S. T. (1996). Robust universal complete codes for transmission and compression. *Discrete Appl. Math.* 64, 31–55. doi:10.1016/0166-218x(93)00116-h

Friend, R. C. (2004). Transport layer security (TLS) protocol compression using lempel-ziv-stac (LZS). *RFC* 3943. doi:10.17487/RFC3943

Gopinath, A., and Ravisankar, M. (2020). “Comparison of lossless data compression techniques,” in *2020 international Conference on inventive computation technologies (ICICT) (IEEE)*, 628–633.

Grumbach, S., and Tahi, F. (1994). A new challenge for compression algorithms: genetic sequences. *Inf. Process. and Manag.* 30, 875–886. doi:10.1016/0306-4573(94)90014-0

Hosseini, M., Pratas, D., and Pinho, A. J. (2016). A survey on data compression methods for biological sequences. *Information* 7, 56. doi:10.3390/info7040056

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proc. IRE* 40, 1098–1101. doi:10.1109/jrproc.1952.273898

Jahaan, A., Ravi, T., and Panneer Arokiaraj, S. (2017). A comparative study and survey on existing dna compression techniques. *Int. J. Adv. Res. Comput. Sci.* 8. doi:10.26483/ijarcs.v8i3.3086

Kavitha, P. (2016). A survey on lossless and lossy data compression methods. *Int. J. Comput. Sci. and Eng. Technol. (IJCSET)* 7.

Knuth, D. E. (1985). Dynamic huffman coding. *J. algorithms* 6, 163–180. doi:10.1016/0196-6774(85)90036-7

Kodituwakku, S., and Amarasinghe, U. (2010). Comparison of lossless data compression algorithms for text data. *Indian J. Comput. Sci. Eng.* 1, 416–425.

Kryukov, K., Ueda, M. T., Nakagawa, S., and Imanishi, T. (2020). Sequence compression benchmark (scb) database—a comprehensive evaluation of reference-free compressors for fasta-formatted sequences. *GigaScience* 9, giaa072. doi:10.1093/gigascience/giaa072

Langdon, G. G. (1984). An introduction to arithmetic coding. *IBM J. Res. Dev.* 28, 135–149. doi:10.1147/rd.282.0135

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., et al. (2009). The sequence alignment/map format and samtools. *Bioinformatics* 25, 2078–2079. doi:10.1093/bioinformatics/btp352

Lipman, D. J., and Pearson, W. R. (1985). Rapid and sensitive protein similarity searches. *Science* 227, 1435–1441. doi:10.1126/science.2983426

Mahoney, M. V. (2005). Adaptive weighing of context models for lossless data compression. *Tech. Rep.* Florida Tech.

Mansouri, D., Yuan, X., and Saidani, A. (2020). A new lossless dna compression algorithm based on a single-block encoding scheme. *Algorithms* 13, 99. doi:10.3390/a13040099

Martín, G. (1979). Range encoding: an algorithm for removing redundancy from a digitised message. *Video Data Rec. Conf.*, 24–27.

Oberhumer, M. (2008). Lzo-a real-time data compression library.

Ranganathan, N., and Henriques, S. (1993). High-speed vlsi designs for lempel-ziv-based data compression. *IEEE Trans. Circuits Syst. II Analog Digital Signal Process.* 40, 96–106. doi:10.1109/82.219839

Ryabko, B. Y. (1980). Data compression by means of a “book stack”. *Probl. Peredachi Inf.* 16, 16–21.

Salomon, D. (2004). *Data compression: the complete reference*. Springer Science and Business Media.

Shannon, C. E. (2001). A mathematical theory of communication. *ACM Sigmob. Mob. Comput. Commun. Rev.* 5, 3–55. doi:10.1145/584091.584093

Storer, J. A., and Szymanski, T. G. (1982). Data compression via textual substitution. *J. ACM (JACM)* 29, 928–951. doi:10.1145/322344.322346

Stout, Q. (1980). Improved prefix encodings of the natural numbers (corresp.). *IEEE Trans. Inf. Theory* 26, 607–609. doi:10.1109/tit.1980.1056237

Tunstall, B. P. (1968). *Synthesis of noiseless compression codes*. Atlanta, FL: Georgia Institute of Technology. Ph.D. thesis.

Uthayakumar, J., Vengattaraman, T., and Dhavachelvan, P. (2018). Swarm intelligence based classification rule induction (CRI) framework for qualitative and quantitative approach: an application of bankruptcy prediction and credit risk analysis. *J. King Saud University-Computer Inf. Sci.* 32, 647–657. doi:10.1016/j.jksuci.2017.10.007

Vitter, J. S. (1987). Design and analysis of dynamic huffman codes. *J. ACM (JACM)* 34, 825–845. doi:10.1145/31846.42227

Welch, T. A. (1984). A technique for high-performance data compression. *Computer* 17, 8–19. doi:10.1109/mc.1984.1659158

Willems, F. M., Shtarkov, Y. M., and Tjalkens, T. J. (1995). The context-tree weighting method: basic properties. *IEEE Trans. Inf. theory* 41, 653–664. doi:10.1109/18.382012

Williams, R. N. (1991). “An extremely fast ziv-lempel data compression algorithm,” in *[1991] proceedings. Data compression conference (IEEE)*, 362–371.

Ziv, J., and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Trans. Inf. theory* 23, 337–343. doi:10.1109/tit.1977.1055714

Ziv, J., and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* 24, 530–536. doi:10.1109/tit.1978.1055934