A Novel Lossless Encoding Algorithm for Data Compression

Abdelghani Tbakhi (atbakhi@khcc.jo)
King Hussein Cancer Center
Anas Al-okaily
King Hussein Cancer Center https://orcid.org/0000-0002-1188-8266

Article

Keywords: data compression, encoding algorithm, DNA data

Posted Date: May 24th, 2021

DOI: https://doi.org/10.21203/rs.3.rs-499681/v1

A Novel Lossless Encoding Algorithm for Data Compression

Anas Al-okaily[1], Abdelghani Tbakhi[1]

[1]*Department of Cell Therapy & Applied Genomics, King Hussein Cancer Center, Amman, 11941, Jordan.*

Data compression is a challenging and increasingly important problem. As the amount of data generated daily continues to increase, efficient transmission and storage has 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 similarity patterns, and binning similar subsequences together. The data are then compressed in 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 considerable improvement in the compression of each genome, preserving several megabytes compared with state-of-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.

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, while 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. This study focuses specifically on lossless compression. Data exist in different formats including text, numbers, images, audio, and video. Several coding algorithms and corresponding variants have been developed for textual data, the primary focus of this paper. This includes the Huffman [1], Shannon [2], Shannon-Fano [3], Shannon-Fano-Elias [4], Lempel-Ziv (LZ77)[5], the Burrows-Wheeler transform [6], and Tunstall [7] algorithms. The Huffman algorithm includes several variants: minimum variance Huffman, canonical Huffman, length-limited Huffman, non-binary Huffman, adaptive Huffman, Faller-Gallager-Knuth (an adaptive Huffman) [8], and Vitter (an adaptive Huffman) [9]. The LZ algorithm also includes several variants, such as LZ78 [10], Lempel-Ziv-Welch (LZW) [11], Lempel-Ziv-Stac (LZS) [12], Lempel-Ziv-Oberhumer (LZO) [13], Lempel-Ziv-Storer-Szymanski (LZSS) [14], Lempel–Ziv-Ross-Williams (LZRW) [15], and the Lempel–Ziv–Markov chain algorithm (LZMA) [16]. Additional techniques involve arithmetic encoding [17], range encoding [18], move-to-front encoding (also referred as symbol ranking encoding) [19,20], run-length encoding [21], delta encoding, unary encoding, context tree weighting encoding [22], prediction by partial matching [23], context mixing [24], asymmetric numeral systems (also called asymmetric

binary coding) [25], length index preserving transform [26], and dynamic Markov encoding [27].

Common tools used to implement one or more encoding algorithms and compress textual data are listed and described in Table S1. 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, pbzip2, pigz, ppm, qzip, rans static, rzip, snzip, srank, xz, zlib, zip, zpipe, and zstd. The performance of these tools is promising, but further improvements are still possible and necessary, due to the high volume of data that being generated worldwide. The results produced by these tools were compared with those of the proposed technique as part of the study. 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 [28–32], however, a brief description for each method is provided in the Supplementary Information.

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 [33] requiring more than three gigabytes of storage). As such, sequencing genomic data (especially for humans) is currently being performed for research and diagnostic purposes in daily basis. In addition, this sequencing is typically conducted with high

depth (30-100x) to sequence the same region several times in order to make reading the DNA regions more accurate. This generates massive quantities of data on a daily basis. For example, the number of bases sequenced from December 1982 through February 2021 was 12,270,717,209,612 [34]. Several algorithms have been developed to compress these data, which can be divided into vertical and horizontal techniques [35]. Vertical mode algorithms utilize a reference genome/source, while horizontal mode algorithms are reference-free. The algorithm proposed in this study is a reference-free (horizontal) model.

Genomic data are stored in different formats, including FATSA [36], FASTQ [37], and SAM [38], 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 [39–43]. 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 [40,44].

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 a novel lossless, reference-free encoding algorithm that is applicable to any data type (i.e., numbers, images, audio, and video), in addition to genomic or textual data.

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, context weighting tree, predictive partial matching, or dynamic Markov compression. Even if these sequences are located within the same sliding window, 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.

Compression algorithm Given a string S of length s, containing characters from a fixed alphabet of length Σ, a window of length w, and a label sequence L initialized as empty, the proposed algorithm can be expressed as follows.

1. Scan S with non-overlapping windows of length w. In each window:

   (a) Using a classification method, classify the sequence in the window based on its content. Create a unique label l that denotes this classification.

   (b) Create, if not any, a bin labeled l. Then, concatenate sequence of the window with the sequence in this bin.

   (c) Concatenate l with L. The purpose of L is to store the queue/order of the sequences' labels. This order is needed during the decompression process to restore the sequences (distributed in the bins) to their original placements in the input data.

   The time cost for this step is $O(\frac{sc}{w})$ where c is the cost of classifying each sequence of length w. If c is less than or equal to w, the cost of this step is $O(s)$.

2. Once the scanning of S is completed, encode all collected labels in L using Huffman encoding. The time cost for this step is $O(BlogB)$ where B is the number of labels. These labels can be set dynamically or designed in advance, and they can be set by the user or the implementer.

3. 3. Compress L using the encoding schema from the previous step. The time cost of this step is $O(\frac{s}{w})$.

4. Compress the sequence in each bin using an algorithm suitable for the content and/or the label of the bin.

5. Pack or archive, with or without further compression, all resultant compressed files (bins and L) together into a single file.

Note the value of w can be fixed or variable. If w is a variable, the window is extended character by character until the label of the sequence in the window matches one of the bin labels. The length of the window must then be added to L (after the label string for each window) or placed in a separate stream that stores lengths and their order. Lastly, lengths can be encoded using a universal code for integers (such as Levenshtein coding [45], Elias coding [46] (delta, gamma, or omega), exponential-Golomb code [47], Fibonacci code [48], Stout code [49]) or using a suitable encoding algorithm such as unary, binary, or Huffman.

Decompression algorithm Decompression is the inverse of compression and can be described as follows.

1. If all bins and L were compressed/archived into a single file, decompress/unarchive them.
2. Decompress each bins file and L with the compression tool that was used to compress them.
3. Initialize a counter for each bin.
4. Sequentially read labels from L. At each read, extract a sequence of length w from the bin with label equal to the read label. Extraction of the sequence must start from the position equal to the value of the counter associated with that bin. Increment *counter* by the value of w plus 1 and then output the extracted sequence to the output stream.

The time and memory cost of decompression is linear.

This algorithm can be applied not only to DNA or textual data, but to archiving and other data

types as well (e.g., numbers, images, audio, and video). Sub-binning or nested-binning processes can also be applied to each bin.

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 of compressed together, to achieve better compression results with a minor increase in time costs. In the case of more random/divergent data, which is common, this is done to avoid relying on a single encoding or compression technique (as in entropy methods), being dependent on the randomness of previous sequences (as in prediction methods), requiring construction of dictionaries dependent on the randomness of previous sequences (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 and contributes in organizing the input data 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 Information.

This encoding algorithm is named in honor of Pramod Srivastava (Professor in the Department of Immunology, University of Connecticut School of Medicine) who served as an advisor to the first author. As such, it is named the Okaily-Srivastava-Tbakhi (*OST*) algorithm.

OST-DNA The first implementation of the OST algorithm (OST-DNA) accepts DNA data as input. Bin labels are computed using a Huffman tree encoding strategy. 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.

As the windows do not overlap, each base in *S* 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 $O(\frac{s}{w})$ Huffman trees for all subsequences in S. The total cost hence is $O(\frac{s\Sigma log\Sigma}{w})$. 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 the compression process of OST-DNA is $O(s)$.

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 may be produced by the archiving process.

Results

We implemented OST-DNA using the python language. We used the same dataset applied to another benchmark [39] in order to test and compare the compression results from OST-DNA with the tools listed in Table S1. The dataset consists of seventeen genomes, as shown in Tables S2 and S3. The main purpose of this implementation and benchmarking is to provide a proof-of-concept version of the proposed algorithm for academic use.

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". The python script used to perform these preprocessing steps and the size of each genome, before and after applying the script, are provided in Table S3. The size of one-line genomes ranged from 50 KB to 13 GB, with a total size of 16,773.88 Megabytes.

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).

Each tool in Table S1 was applied to each one-line genome. The compression and decompression commands used to run each tool are provided in Table S5. The cumulative compression results (for all one-line genomes) are provided in Table 1, while the compression results for each one-line genome are listed in Extended Data Table 1. 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.

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 same command used by each tool to compress one-line genomes was used to compress the bins. Each of these seven versions were run over each window lengths of 25, 50, 100, 125, 150, 250, 500, 750, 1,000, 2,500, 5,000, and 10,000 and across label lengths

of 1, 2, 3, 4, and 5. The best cumulative compression performance achieved by each OST-DNA version is shown in Table 2.

A comparison of the results produced by each OST-DNA tool (i.e., bcm, brotli, bsc, bzip2, lrzip, lzip, and xz) 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 the proposed algorithm can improve compression results compared to the individual corresponding tools.

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 but was 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 Extended Data Table 2. The compression results for the standalone tools are provided in Table 1.

By analyzing the compression results for all OST-DNA versions, using different sequence label lengths and classification methods (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. Additional classification methods can be used to label the bins and improve compression further. Extended Data Table 3 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 Extended Data Table 4.

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 Ex-

tended Data Table 5). This was not the case for bins produced using the same label length and same genome, but with different window lengths (see Extended Data Table 6). 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.

References

1. Huffman, D. A. *et al.* A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* **40**, 1098–1101 (1952).

2. Shannon, C. E. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review* **5**, 3–55 (2001).

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

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

5. Ziv, J. & Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* **23**, 337–343 (1977).

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

7. Tunstall, B. P. *Synthesis of noiseless compression codes*. Ph.D. thesis, Georgia Institute of Technology (1967).

8. Knuth, D. E. Dynamic huffman coding. *Journal of algorithms* **6**, 163–180 (1985).

9. Vitter, J. S. Design and analysis of dynamic huffman codes. *Journal of the ACM (JACM)* **34**, 825–845 (1987).

10. Ziv, J. & Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory* **24**, 530–536 (1978).

11. Welch, T. A. A technique for high-performance data compression. *Computer* 8–19 (1984).

12. Friend, R. C. Transport Layer Security (TLS) Protocol Compression Using Lempel-Ziv-Stac (LZS). RFC 3943 (2004). URL `https://rfc-editor.org/rfc/rfc3943.txt`.

13. Oberhumer, M. Lzo-a real-time data compression library. *http://www. oberhumer. com/opensource/lzo/* (2008).

14. Storer, J. A. & Szymanski, T. G. Data compression via textual substitution. *Journal of the ACM (JACM)* **29**, 928–951 (1982).

15. Williams, R. N. An extremely fast ziv-lempel data compression algorithm. In *[1991] Proceedings. Data Compression Conference*, 362–371 (IEEE, 1991).

16. Ranganathan, N. & Henriques, S. High-speed vlsi designs for lempel-ziv-based data compression. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* **40**, 96–106 (1993).

17. Langdon, G. G. An introduction to arithmetic coding. *IBM Journal of Research and Development* **28**, 135–149 (1984).

18. Martín, G. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video and Data Recording Conference, Southampton, 1979*, 24–27 (1979).

19. Ryabko, B. Y. Data compression by means of a "book stack". *Problemy Peredachi Informatsii* **16**, 16–21 (1980).

20. Bentley, J. L., Sleator, D. D., Tarjan, R. E. & Wei, V. K. A locally adaptive data compression scheme. *Communications of the ACM* **29**, 320–330 (1986).

21. Capon, J. A probabilistic model for run-length coding of pictures. *IRE Transactions on Information Theory* **5**, 157–163 (1959).

22. Willems, F. M., Shtarkov, Y. M. & Tjalkens, T. J. The context-tree weighting method: basic properties. *IEEE transactions on information theory* **41**, 653–664 (1995).

23. Cleary, J. & Witten, I. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications* **32**, 396–402 (1984).

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

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

26. Awan, F. S. & Mukherjee, A. Lipt: A lossless text transform to improve compression. In *Proceedings International Conference on Information Technology: Coding and Computing*, 452–460 (IEEE, 2001).

27. Cormack, G. V. & Horspool, R. N. S. Data compression using dynamic markov modelling. *The Computer Journal* **30**, 541–550 (1987).

28. Gopinath, A. & Ravisankar, M. Comparison of lossless data compression techniques. In *2020 International Conference on Inventive Computation Technologies (ICICT)*, 628–633 (IEEE, 2020).

29. Kavitha, P. A survey on lossless and lossy data compression methods. *International Journal of Computer Science & Engineering Technology (IJCSET)* **7** (2016).

30. squash. `https://quixdb.github.io/squash-benchmark/`.

31. Uthayakumar, J., Vengattaraman, T. & Dhavachelvan, P. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University-Computer and Information Sciences* (2018).

32. Kodituwakku, S. & Amarasinghe, U. Comparison of lossless data compression algorithms for text data. *Indian journal of computer science and engineering* **1**, 416–425 (2010).

33. humangenomesize. `https://www.ncbi.nlm.nih.gov/grc/human/data?asm=GRCh38.p13`.

34. genbank. `https://www.ncbi.nlm.nih.gov/genbank/statistics/`.

35. Grumbach, S. & Tahi, F. A new challenge for compression algorithms: genetic sequences. *Information Processing & Management* **30**, 875–886 (1994).

36. Lipman, D. J. & Pearson, W. R. Rapid and sensitive protein similarity searches. *Science* **227**, 1435–1441 (1985).

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

38. Li, H. *et al.* The sequence alignment/map format and samtools. *Bioinformatics* **25**, 2078–2079 (2009).

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

40. Hosseini, M., Pratas, D. & Pinho, A. J. A survey on data compression methods for biological sequences. *Information* **7**, 56 (2016).

41. Mansouri, D., Yuan, X. & Saidani, A. A new lossless dna compression algorithm based on a single-block encoding scheme. *Algorithms* **13**, 99 (2020).

42. Bakr, N. S., Sharawi, A. A. *et al.* Dna lossless compression algorithms. *American Journal of Bioinformatics Research* **3**, 72–81 (2013).

43. Jahaan, A., Ravi, T. & Panneer Arokiaraj, S. A comparative study and survey on existing dna compression techniques. *International Journal of Advanced Research in Computer Science* **8** (2017).

44. Bonfield, J. K. & Mahoney, M. V. Compression of fastq and sam format sequencing data. *PloS one* **8**, e59190 (2013).

45. Levenshtein. `http://www.compression.ru/download/articles/int/levenstein_1968_on_the_redundancy_and_delay.pdf`, author=Vladimir Levenshtein, year=1968.

46. Elias, P. Universal codeword sets and representations of the integers. *IEEE transactions on information theory* **21**, 194–203 (1975).

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

48. Fraenkel, A. S. & Kleinb, S. T. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics* **64**, 31–55 (1996).

49. Stout, Q. Improved prefix encodings of the natural numbers (corresp.). *IEEE Transactions on Information Theory* **26**, 607–609 (1980).

Table 1: **Compression performance for each common tool cumulatively over all tested genomes**.

Tool	CP Ratio (%)	Saved Space (MB)	CP Time (seconds)	DP Time (seconds)	CP Speed (MB/s)	DP Speed (MB/s)
bcm	20.6217	13,314.83	4,071	3,728	4.1203	0.9279
blzpack	37.2279	10,529.31	227	149	73.8937	41.9098
brotli	16.7848	13,958.42	62,659	103	0.2677	27.3346
bsc	20.1670	13,391.09	2,369	68	7.0806	49.7469
bzip2	24.8765	12,601.12	2,474	1,169	6.7801	3.5695
compress	25.3977	12,513.70	443	168	37.8643	25.3582
freeze	27.4616	12,167.51	6,078	233	2.7598	19.7698
gzip	26.9031	12,261.19	4,211	171	3.9833	26.3900
hook	21.3395	13,194.41	7,803	8,074	2.1497	0.4433
Huffman-codec	27.4015	12,177.59	1,152	401	14.5607	11.4621
lizard	34.8186	10,933.45	11,449	41	1.4651	142.4494
lrzip	14.6446	14,317.40	21,589	378	0.7770	6.4986
lz4	52.7757	7,921.34	161	73	104.1856	121.2677
lzfse	29.3101	11,857.45	1,118	130	15.0035	37.8187
lzip	17.0353	13,916.39	20,079	304	0.8354	9.3996
lzop	47.6212	8,785.96	152	106	110.3545	75.3578
LzTurbo	28.4807	11,996.56	157	46	106.8400	103.8548
pbzip2	24.9054	12,596.28	2,458	1,193	6.8242	3.5018
pigz	26.9356	12,255.74	4,284	109	3.9155	41.4508
ppm	23.8049	12,780.87	5,020	6,314	3.3414	0.6324
qzip	41.9873	9,730.98	1,556	126	10.7801	55.8960
rans	24.0431	12,740.93	144	91	116.4853	44.3182
rzip	24.9315	12,591.89	2,515	1,279	6.6695	3.2697
snzip	45.5241	9,137.73	159	73	105.4961	104.6048
srank	41.2318	9,857.70	794	778	21.1258	8.8897
xz	17.0381	13,915.92	18,666	266	0.8986	10.7442
zip	26.9031	12,261.19	4,165	173	4.0273	26.0849
zlib	26.9187	12,258.57	4,278	117	3.9210	38.5924
zpipe	26.9187	12,258.57	4,283	106	3.9164	42.5973
zstd	26.7482	12,287.18	251	75	66.8282	59.8227

CP is abbreviation for compression and DP for decompression. The size of all genomes 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.

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	CP Ratio (%)	Saved Space (MB)	CP Time (seconds)	DP Time (seconds)	CP Speed (MB/s)	DP Speed (MB/s)
bcm	-	-	20.6217	13,314.83	4,071	3,728	4.1203	0.9279
OST-DNA-bcm	250	4	20.1603	13,392.21	12,026	4,388	1.3948	0.7707
brotli	-	-	16.7848	13,958.42	62,659	103	0.2677	27.3346
OST-DNA-brotli	750	4	15.9477	14,098.83	72,315	318	0.2320	8.4121
bsc	-	-	20.1670	13,391.09	2,369	68	7.0806	49.7469
OST-DNA-bsc	250	5	19.7688	13,457.88	10,355	2,160	1.6199	1.5352
bzip2	-	-	24.8765	12,601.12	2,474	1,169	6.7801	3.5695
OST-DNA-bzip2	250	2	24.6689	12,635.95	10,132	1,230	1.6555	3.3642
lrzip	-	-	14.6446	14,317.40	21,589	378	0.7770	6.4986
OST-DNA-lrzip	1,000	1	14.5728	14,329.45	31,911	432	0.5256	5.6584
lzip	-	-	17.0353	13,916.39	20,079	304	0.8354	9.3996
OST-DNA-lzip	750	2	16.8067	13,954.74	30,691	472	0.5465	5.9728
xz	-	-	17.0381	13,915.92	18,666	266	0.8986	10.7442
OST-DNA-xz	750	2	16.7898	13,957.57	29,098	393	0.5765	7.1662

CP is abbreviation for compression and DP for decompression. The size of all genomes is 16,773.88 MB.

Competing Interests None declared.

Correspondence Correspondence and requests for materials should be addressed to Anas Al-okaily (email: AA.12682@khcc.jo).

Authors Contribution Both authors contributed equally to this work and hence both names should be considered first name.

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

Supplementary Information is available for this paper

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- Supplementaryinformation.pdf
- ExtendedDataTable1.xlsx
- ExtendedDataTable2.xlsx
- ExtendedDataTable3.xlsx
- ExtendedDataTable4.xlsx
- ExtendedDataTable5.xlsx
- ExtendedDataTable6.xlsx