UDC 004.42:004.38:004.5:681.5

https://doi.org/10.52058/2786-6025-2025-13(54)-1676-1688

Liashuk Taras Hryhorovych Ph.D. in Physics and Mathematics, Senior Lecturer, Department of Information Technologies and Modeling, Rivne State University of the Humanities, https://orcid.org/0000-0002-2242-7537

Babych Stepaniia Mykhailivna Ph.D. in Engineering, Associate Professor, Department of Information Technologies and Modeling, Rivne State University of the Humanities, https://orcid.org/0000-0003-2145-6392

Siaskyi Volodymyr Andriyovych Ph.D. in Engineering, Associate Professor, Department of Information Technologies and Modeling, Rivne State University of the Humanities, https://orcid.org/0000-0002-2648-4934

Kindrat Pavlo Vadymovych Ph.D. in Law, Associate Professo, Department of Digital Technologies and Methods of Teaching Informatics, Rivne State University of the Humanities, https://orcid.org/0000-0003-0351-3349

MECHANISMS AND PROSPECTS OF CLASSICAL LOSSLESS COMPRESSION ALGORITHMS IN THE ARCHITECTURE OF OPERATING SYSTEMS, SOFTWARE AND HARDWARE IMPLEMENTATIONS OF COMPUTER-PHYSICAL SYSTEMS

Abstract. A systematic analysis of classical lossless compression algorithms – RLE, Huffman coding, LZ77 and LZ78 – is conducted. The research focuses on studying the mechanisms of their operation, key characteristics, advantages and disadvantages, as well as on substantiating their role as fundamental components of modern computing systems.

In particular, it is demonstrated that RLE remains indispensable for eliminating sequential redundancy in homogeneous data, while the Huffman method is a reference entropy coding algorithm that eliminates frequency redundancy. The LZ77 and LZ78 algorithms serve as the basis for dictionary compression that eliminates structural redundancy.

It is shown that in modern computing systems none of the basic algorithms is used in a “pure” form. Instead, hybrid architectures dominate, in which LZ mechanisms are inextricably integrated with entropy coding.

Based on this analysis, the integration of these algorithms into the architecture of operating systems (kernel image and ZRAM memory compression, Btrfs/ZFS file

systems, network protocols, caching of graphics and user interface elements) and software (archivers, media formats) is systematized.

The most promising directions for further development and application of classical methods are identified: hybridization (creation of adaptive dictionaries), parallelism (using GPU for high-performance data arrays) and hardware acceleration (FPGA/ASIC). The potential of LZ algorithms in hardware implementations for computer-physical systems and their methodological role in optimizing the control of quantum states is investigated.

Keywords: lossless compression algorithms, dictionary/entropy coding, data archiving, operating systems, software, computer-physical systems.

Ляшук Тарас Григорович кандидат фізико-математичних наук, старший викладач кафедри інформаційних технологій та моделювання, Рівненський державний гуманітарний університет, https://orcid.org/0000-0002-2242-7537

Бабич Степанія Михайлівна кандидат технічних наук, доцент кафедри інформаційних технологій та моделювання, Рівненський державний гуманітарний університет, https://orcid.org/0000-0003-2145-6392

Сяський Володимир Андрійович кандидат технічних наук, доцент кафедри інформаційних технологій та моделювання, Рівненський державний гуманітарний університет, https://orcid.org/0000-0002-2648-4934

Кіндрат Павло Вадимович кандидат юридичних наук, доцент кафедри цифрових технологій та методики навчання інформатики, Рівненський державний гуманітарний університет, https://orcid.org/0000-0003-0351-3349

МЕХАНІЗМИ ТА ПЕРСПЕКТИВИ КЛАСИЧНИХ АЛГОРИТМІВ СТИСНЕННЯ БЕЗ ВТРАТ В АРХІТЕКТУРІ ОПЕРАЦІЙНИХ СИСТЕМ, ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ ТА АПАРАТНИХ РЕАЛІЗАЦІЯХ КОМП'ЮТЕРНО-ФІЗИЧНИХ СИСТЕМ

Анотація. Проведено системний аналіз класичних алгоритмів стиснення без втрат – RLE, кодування Хаффмана, LZ77 та LZ78. Дослідження зосереджено на вивченні механізмів їхньої роботи, ключових характеристик, переваг та недоліків, а також на обґрунтуванні їхньої ролі як фундаментальних складових сучасних обчислювальних систем.

Зокрема, продемонстровано, що RLE залишається незамінним для усунення послідовної надмірності в однорідних даних, тоді як метод Хаффмана

є еталонним алгоритмом ентропійного кодування, що усуває частотну надмірність. Алгоритми LZ77 та LZ78 слугують базою для словникового стиснення, що усуває структурну надмірність.

Показано, що в сучасних обчислювальних системах жоден із базових алгоритмів не використовується в «чистому» вигляді. Замість цього, домінують гібридні архітектури, в яких LZ-механізми нерозривно інтегровані з ентропійним кодуванням.

На основі такого аналізу, систематизовано інтеграцію цих алгоритмів в архітектуру операційних систем (стиснення образу ядра та пам'яті ZRAM, файлові системи Btrfs/ZFS, мережеві протоколи, кешування графіки і елементів інтерфейсу користувача) та програмне забезпечення (архіватори, медіа-формати).

Визначено найбільш перспективні напрямки подальшого розвитку та застосування класичних методів: гібридизація (створення адаптивних словників), паралелізм (використання GPU для високопродуктивних масивів даних) та апаратне прискорення (FPGA/ASIC). Досліджено потенціал LZ-алгоритмів у апаратних реалізаціях для комп'ютерно-фізичних систем та їхня методологічна роль в оптимізації керування квантовими станами.

Ключові слова: алгоритми стиснення без втрат, словникове/ентропійне кодування, архівація даних, операційні системи, програмне забезпечення, комп’ютерно-фізичні системи.

Problem statement. In the context of the exponential growth of digital data volumes, the problem of effective information resource management is becoming of paramount importance. The limitations of network bandwidth and physical storage media necessitate continuous improvement of data storage and transmission mechanisms. Although modern hybrid compression algorithms demonstrate high efficiency ratios, their architecture is rooted in the fundamental principles established by classical dictionary and entropy-based methods. Understanding the mechanisms, advantages, and limitations of these basic algorithms is a necessary condition for their further integration, optimization, and adaptation to the requirements of modern operating systems (OS), software, and architectural solutions at the physical level. Therefore, there is a need to systematize and analyze the role of these classical approaches within the context of modern computing paradigms, as well as to define their future development and modification paths to ensure maximum compression efficiency.

Analysis of recent research and publications. Within the context of lossless compression research, an analysis of recent publications [1] clearly demonstrates that modern compression engineering represents a further evolution of the fundamental ideas established in the seminal works of D. Huffman, J. Ziv, and A. Lempel. In

particular, a significant portion of research over the last decades [2] has focused on the optimal integration of dictionary-based mechanisms with entropy coding. The author relies on hybrid models that have proven their effectiveness in software environments, as well as on contemporary research in high-speed algorithms that demonstrate novel approaches to dictionary management and entropy coding.

Despite significant progress in developing universal hybrid algorithms, certain aspects of the general problem remain unresolved, particularly regarding specialized applications. Specifically, there are gaps in systematizing the selection criteria for base algorithms for specific system tasks where the balance between speed and compression ratio is critical [3]: for instance, in OS kernel image compression, memory page swapping (ZRAM), or block-level filesystem operations (Btrfs, ZFS). The role of RLE and its advanced analogs as extremely fast pre-filters also remains insufficiently explored. Furthermore, the dynamic advancement of hardware acceleration (FPGA, ASIC) and the necessity to adapt to new data types (such as bioinformatics or quantum computing) necessitate a clear classification and analysis of the modification prospects for these classical mechanisms [4].

Purpose of the article. The purpose of this article is to systematize the selection criteria between dictionary and entropy coding for OS and software system tasks, and to conduct a comprehensive analysis of classical lossless compression mechanisms. This analysis serves to justify their effectiveness in the design of hardware solutions tailored for the requirements of computer-physical systems.

Presentation of the main research material. For applications where data integrity is critical (specifically within the OS kernel and filesystems), lossless compression is utilized. It ensures perfect, bit-for-bit restoration of information after the inverse transformation. Lossless compression refers to a group of methods that eliminate statistical data redundancy while ensuring absolute process reversibility, allowing for an exact copy of the original data without any distortion. This approach is particularly vital for data types such as documents, source code (e.g., programming code, CSV, JSON, XML), databases, financial reports, log files, and system data (journals, configurations, backups), where even a minor loss of information is unacceptable. Lossless compression is also applied to multimedia data when it is necessary to preserve information in its original uncompressed form (e.g., PNG, FLAC). The fundamental principles of this type of compression involve identifying repeating patterns, replacing these sequences with shorter markers, and constructing tables or dictionaries for optimal encoding. The most common lossless compression algorithms are discussed below.

RLE (Run-Length Encoding) is one of the oldest and simplest data compression methods, based on the principle of replacing sequences of identical symbols with the count of those symbols followed by the symbol itself. Its foundations were known and utilized even before the advent of computers,

particularly in telegraphy and early fax transmission systems for bitmap compression. In computer science, RLE began to be widely applied in the 1960s [5].

To understand its operation, let us consider a case involving a text file containing a string. In this instance, the encoding process will be as follows:

$$\underbrace{\text{AAAAABBBCCDAA}}_{\text{Input string (13 symbols)}} \xrightarrow{\text{RLE}} \underbrace{\text{5A3B2C1D2A}}_{\text{RLE-code (10 symbols)}}.$$

This example demonstrates how data can be effectively compressed (13 → 10). This allows the data to occupy less space on the physical medium and to be easily unpacked into the original string. However, such compression is not always rational. The following example illustrates this:

$$\underbrace{\text{ABCDEFGHIJK}}_{\text{Input string (11 symbols)}} \xrightarrow{\text{RLE}} \underbrace{\text{1A1B1C1D1E1F1G1H1I1J1K}}_{\text{RLE-code (22 symbols)}}.$$

Therefore, it is evident that this algorithm is effective only for data with numerous repetitions and may even increase the output file size. Furthermore, RLE is not suitable for all data types, unlike dictionary-based algorithms or entropy coding. However, the advantages of such an algorithm are clear: simplicity, speed, and losslessness.

The decoding of RLE data is quite trivial and consists of expanding the encoded "count-symbol" pairs back into the original sequence by repeating each symbol the specified number of times.

There are many variations of RLE: classical, marker-based, fixed-length, etc. Each is optimized for different data types: marker-based RLE avoids ambiguity when the markers themselves appear in the data; fixed-length RLE simplifies processing and ensures faster decompression.

Regarding its fields of application, this compression algorithm is designed to work with homogeneous data (containing long sequences of identical values: solid color areas in images, null bytes in memory, etc.), where encoding/decoding speed and minimal computational complexity are critical. At the same time, RLE does not guarantee a high compression ratio – it all depends on the level of sequential data repetition. For this reason, RLE acts as a specialized tool rather than a general compression mechanism, as it is ineffective for high-entropy data (e.g., text/binary files, complex images).

Based on this, at the application level, RLE is used for raster graphics (BMP, TIFF, PCX, Group 3 and Group 4 fax), video (encoding sequential frames and motion vectors), databases (compressing columns with repeating values), network protocols (compressing repetitive packets), game graphics (historically, for compressing sprites and backgrounds for arcade and platform games), microcontrollers (storing static display elements: icons, fonts, and other display components), scientific data (compressing simulation results), and others.

In the context of the OS, RLE is important for caching graphics and UI elements (e.g., icons, window elements, system cursors), as well as for the frame

buffer when most of the screen remains static (e.g., BIOS, console mode, OS splash screen). Additionally, RLE is critically important for remote desktop protocols (e.g., VNC, RDP), where compressing changing screen regions during network transmission provides significant bandwidth savings if most of the screen remains static.

At the memory management level, although modern operating systems typically use more complex LZ-algorithms for swapping, RLE can be applied for very fast pre-compression of blocks known to contain long sequences of zeros or other homogeneous data. This allows for the optimization of more complex encoders or the avoidance of redundant disk writes.

The limitations of RLE regarding high-entropy data necessitated the development of more universal methods that account for not only direct repetitions but also the stochastic nature of symbol occurrences in the input data stream. One such fundamental method is **Huffman Coding** [6], based on information theory. Although it is significantly more complex than RLE, it is considerably more effective. It is one of the most optimal entropy compression algorithms, allowing for data size reduction by utilizing variable-length bit codes for different symbols. Symbols that occur less frequently receive longer codes, while more frequent ones receive shorter codes.

The Huffman coding algorithm involves the following stages:

- frequency calculation: the frequency of occurrence for each symbol in the input stream is calculated;
- creation of leaf nodes: a "leaf node" (a tree node with no children) is created for each unique symbol, containing the symbol itself and its frequency. All nodes are placed in a priority queue (min-heap), where the lowest frequencies have the highest priority, meaning they are extracted first;
- construction of the binary tree: the two nodes with the lowest frequencies are merged into a new node, the frequency of which equals the sum of its children's frequencies (any two of the smallest can be chosen; the order may affect the tree's appearance but not the code lengths). The newly created node is added back into the queue. These steps are repeated until only one node remains in the queue, which becomes the root of the Huffman tree;
- symbol encoding within the tree: typically, the left child node is encoded as 0, and the right child as 1 (or vice versa, provided the sequence remains consistent). Consequently, the most frequent symbols are assigned the shortest binary codes, while the rarest symbols receive the longest codes;
- encoding the output character string: each symbol in the original string is assigned its corresponding binary code.

For example, given the input string "ABRACADABRA", Huffman coding will result in the following output:

$$\underbrace{\text{ABRACADABRA}}_{\text{Input string (11 symbols)}} \xrightarrow{\text{Huffman Codding}} \underbrace{0\ 110\ 111\ 0\ 100\ 0\ 101\ 0\ 110\ 111\ 0}_{\text{Huffman-code (23 bits)}}.$$

Thus, if Huffman coding were not used and each unique symbol were encoded with an equal number of bits, then in the string "ABRACADABRA", encoding 5 different symbols would require at least $\log_2 5 = 3$ bits per symbol. Consequently, the total number of bits would be $= 11 \cdot 3 = 33$. The advantage is clear: 23 bits versus 33 bits.

The constructed Huffman tree is used for decoding. Bits from the encoded sequence are read one by one, starting from the root of the tree. Each bit indicates which branch to follow (0 – left, 1 – right). As soon as a leaf node is reached, the symbol is recognized. Then the process repeats from the root for the next symbol. Due to the prefix property, this process is always unambiguous.

The advantages of this algorithm are losslessness (full restoration of the original data), optimality (provides the maximum possible compression for given symbol frequencies under the condition of using prefix coding), and simplicity (the algorithm is relatively easy to understand and program).

At the same time, the disadvantages of the algorithm are as follows: decoding requires a code table (or the Huffman tree itself; which adds overhead to the file size, especially for short data); inefficiency for uniform data (if all symbols occur with approximately the same frequency, the algorithm may increase the file size due to the overhead of storing the code table); two passes (the algorithm requires two passes over the data: one for frequency calculation and the second for encoding, which is not an optimal method, especially for stream processing).

The primary role of Huffman coding at the system level is to provide the entropy compression stage for data that has been pre-processed by dictionary methods (LZ77-derivatives). For example, in modern algorithms (e.g., Zstd) used in ZRAM or Btrfs, the tokens generated at the LZ-stage are often encoded using a principle similar to the Huffman method to ensure maximum bit efficiency. Furthermore, direct use of the Huffman method is found in some Linux distributions and specialized OSs for compressing bootable kernel images.

Regarding application software, the Huffman method is used to ensure optimal compression of data already prepared by other algorithms, specifically: JPEG (compression of DCT coefficient tables), PNG (uses DEFLATE, which includes Huffman), MP3 (partially), MPEG (encoding variable DCT coefficients and motion vector data), ZIP (as one of the compression methods), encoding of text files and logs, and facsimile machines (CCITT Group 3, Group 4). Additionally, some data tables in complex font formats (e.g., TTC, OpenType) used by the OS for text rendering are compressed using this algorithm to reduce font file size. Moreover, in some network protocols and transmission formats (e.g., HTTP compression), RLE or Huffman can be used as part of a general compression mechanism to optimize traffic.

While RLE and Huffman methods effectively eliminate sequential and frequency redundancy, further increasing the compression ratio requires the use of dictionary algorithms. They provide a more powerful mechanism for identifying and replacing long, repeating sequences of symbols in the data stream, the foundations of which are the Lempel-Ziv family of algorithms, specifically LZ77 and LZ78. These algorithms use a dictionary approach, replacing repeating sequences of symbols (phrases) with references to previous occurrences of these phrases or entries in a dynamic dictionary. The key difference between them lies in how they manage this dictionary.

The **LZ77** algorithm **(Lempel-Ziv 1977)** [7], known as the sliding window compression method, utilizes a fixed memory area that constantly moves across the data. This window is functionally divided into two parts: the search buffer, which acts as a dynamic dictionary containing the most recently processed data, and the look-ahead buffer, which holds the symbols currently awaiting encoding. A critical encoding parameter is the match length range – the minimum and maximum length of a sequence that can be encoded as a reference. Since these parameters (window size and match length range) are not standardized in "pure" LZ77, they vary depending on the specific algorithm modification and its implementation goals (speed versus compression ratio).

When the algorithm finds the longest match for the current sequence of symbols within the search buffer, it replaces it with a short reference – a triplet (token), which is an encoded representation of the found sequence in the format (offset, length, next_symbol). The offset indicates the location of the match in the buffer (the distance backward); notably, if match lengths are identical, the largest offset (i.e., the earliest found phrase) is chosen to optimize dictionary usage (this is done to ensure that as many symbols as possible remain in the dictionary for potentially longer matches later).

The length specifies the number of repeating symbols, and the next_symbol is the first symbol following the match. If no match is found, the algorithm generates a triplet with zero length followed by the current symbol. After outputting the token (regardless of whether a match was found), the sliding window shifts forward by the number of matched symbols plus one. This approach effectively compresses data by converting long repetitions into short codes.

The key characteristics of this algorithm are:

- the dictionary is implicit and dynamic: consisting only of the recently processed portion of the data;
- the dictionary size is fixed: by the size of the sliding window;
- decoding speed is typically higher than encoding speed.

The example below illustrates the LZ77 encoding of the string "ABRACADABRA":

$$\underbrace{\text{ABRACADABRA}}_{\text{Input string (11 symbols)}} \xrightarrow{\text{LZ77}} \underbrace{(0,0,\text{A})(0,0,\text{B})(0,0,\text{R})(3,1,\text{C})(5,1,\text{D})(7,4,\text{''})}_{\text{LZ77-code (6 triplets)}},$$

which results in 6 triplets at the output instead of the 11 original input symbols.

In turn, the LZ77 decoding algorithm is the inverse of the encoding process (which is characteristic of all lossless compression algorithms).

Among the advantages of LZ77, it can be highlighted that this algorithm performs compression on-the-fly, without constructing a complete dictionary. At the same time, it works well for repeating fragments of any length.

On the other hand, the disadvantages include the complexity of implementation due to the sliding window and high RAM requirements when a large window size is used.

It is worth noting that "pure" LZ77 is rarely used in its original form. Primarily, it serves as the foundation for many modern lossless compression algorithms, such as LZO/LZSS/LZ4 (LZ77-modifications), Deflate (LZ77 + Huffman), LZMA (LZ77-modification + arithmetic coding), Zstd (LZ77-modification + FSE), and others.

This algorithm and its modifications (mostly Deflate) are widely applied in various fields, specifically in: archive formats and data compression, including Gzip/ZIP and the Zlib library; image and multimedia formats PNG/PDF, for compressing raster data and reducing the size of embedded images/fonts, respectively; computer networks and web protocols (server-side HTTP compression of web content (HTML, CSS, JS) into Gzip format, before sending it to the client).

Regarding the system level, in OS architecture, modifications of this algorithm are applied for: compressing OS kernel images and initial filesystems to accelerate boot time; reducing the size of the executable file or installation package, including the compression of icons and interface elements, which is necessary for the compactness of distributions and software; compressing memory pages (LZ4/LZO) that are not actively used, which helps reduce physical memory usage; compressing data at the block level in modern filesystems (e.g., Btrfs, ZFS), which not only saves disk space but also increases I/O operation speed; etc. Furthermore, virtualization software (e.g., VMware, Hyper-V) utilizes LZ77-derivatives for efficient storage and rapid transfer of large virtual disk and memory images, which serves as the basis for migration mechanisms and filesystem snapshots.

The primary reason for such popularity of LZ77 is the asymmetry of speed: decoding occurs significantly faster than encoding. This makes it ideal for cases where data needs to be compressed once but rapidly unpacked multiple times (e.g., archives, files for internet download).

While the LZ77 algorithm relies on a sliding window for dynamic searching of already encoded phrases, its efficiency is limited by the fixed size of this window. In contrast, **LZ78 (Lempel-Ziv 1978)** [8] introduced a fundamentally different dictionary mechanism: instead of a moving buffer, it uses an explicit, independent

table (dictionary). This allows for storing and repeatedly referencing all previously found phrases, regardless of their proximity to the current position. This dictionary is built dynamically: the LZ78 algorithm reads the input stream and adds new phrases to the dictionary as they appear.

At each step, the algorithm searches for the longest prefix (analogous to the longest match in the search buffer for LZ77) – the longest sequence of symbols at the beginning of the unencoded data that is already contained in the dictionary – and encodes it as a reference to an index in the table. In the event a match is found, the algorithm outputs a pair of values (dictionary_index, next_symbol), where the index of the found prefix in the dictionary (dictionary_index) is a reference to an already known string, and the next_symbol is the first symbol that did not match. The found prefix, followed by the next symbol, is then added as a new entry to the dictionary. If no match is found, the algorithm adds the current symbol to the dictionary and outputs a pair with a zero index.

Among the key characteristics of this approach, the following can be highlighted:

- the dictionary is explicit and dynamic: increasing in size as new phrases are added;
- the dictionary is constructed and references repeating phrases: using an index;
- it can achieve a higher compression ratio for data with high repeatability than LZ77: since its dictionary can grow to include very long, common phrases from the entire file.

Thus, the LZ78 encoding of "ABRACADABRA" results in 7 pairs at the output instead of 11 symbols:

$$\underbrace{\text{ABRACADABRA}}_{\text{Input string (11 symbols)}} \xrightarrow{\text{LZ78}} \underbrace{(0,\text{A})(0,\text{B})(0,\text{R})(1,\text{C})(1,\text{D})(1,\text{B})(3,\text{A})}_{\text{LZ78–code (7 pairs)}}.$$

During decoding, the dictionary is initially empty. Pairs (index, next_symbol) are read sequentially, and for each pair, a new entry is created: new_entry = dict[index] + next_symbol (where dict[0] = '').

Thus, when encoding the string "ABRACADABRA", LZ77 wins here due to the long repetition of "ABRA" and its mechanism of referencing previous matches. However, if another string is taken, such as "ABCABCABCABCABC", the LZ78 algorithm will show better results because it forms a dictionary of typical patterns ("ABC", "ABCABC", etc.), and subsequent compression occurs through the reuse of dictionary indexes without the need for exact position matching as in LZ77. Consequently, the efficiency of the LZ77 and LZ78 algorithms is determined by the structure of the input data: LZ77 compresses texts with local, closely located repetitions better, while LZ78 is more effective for sequences with regular or distant repetitions.

The advantage of the LZ78 algorithm is its simplicity and speed of implementation, as well as the fact that it does not require a search window. The disadvantages include the possibility of uncontrolled dictionary size growth. Also, there is no internal referencing mechanism characteristic of LZ77, which prevents referencing recently encoded data and leads to excessive fragmentation.

Like its predecessor LZ77, LZ78 is used in the OS, software, and their components indirectly. Primarily, it serves as the basis for one of the most popular compression formats, LZW, which is indispensable for the GIF/TIFF graphic formats. LZ78/LZW also find applications in less common but important areas, notably the original compress utility in UNIX-based systems. In the past, LZW was used in some proprietary data compression schemes integrated into early versions of operating systems.

Regarding the prospects of the RLE/Huffman/LZ77/LZ78 algorithms discussed above, in modern computing systems, none of the basic algorithms are used in "pure" form. Instead, modified or hybrid architectures dominate, where the LZ77 dictionary mechanism is combined with entropy coding to achieve the best result. Consequently, the promising directions for the development and research of these algorithms are:

- hybridization and adaptation: further hybridization to create multi-stage algorithms. One of the most modern algorithms, Zstd, already demonstrates the effectiveness of combining the LZ77 dictionary approach with high-speed FSE entropy coding. Future hybrids should focus on: data-driven learning – using pre-trained dictionaries adapted for specific data types (logs, genomic sequences, code) to increase the compression ratio and speed; integration with RLE – using methods similar to RLE and its advanced variants (Delta Encoding) as a first, extremely fast filter to remove simple repetitions before applying more complex LZ-algorithms, especially for scientific and time-series data;
- hardware acceleration and parallelism: shifting the computational load from the CPU to specialized hardware and parallel architectures: FPGA/ASIC – particularly relevant for LZ4 and LZO, where the simple structure allows for throughput of tens of GB/s, which is vital for high-performance filesystems where compression/ decompression must not create I/O latency, as well as for hardware-level data compression in SSDs and network cards; GPU – especially relevant for high-performance data arrays (e.g., scientific simulations, video encoding, machine learning) that do not fit into the CPU cache; stream processing – adapting algorithms for continuous streaming compression without the need to load the entire file into memory, which is key for network protocols and remote services;
- new fields of application: quantum computing – researching the possibilities of adapting dictionary and entropy coding principles for efficient management and compression of states in quantum systems; bioinformatics – creating specialized

algorithms based on LZ77/LZMA for compressing vast amounts of genomic data (DNA/RNA sequences), where repeatability has a clear structural nature ideal for dictionary methods; security and obfuscation (Malware) – using modified LZ-algorithms to obfuscate malicious software and, conversely, to develop high-efficiency intrusion detection systems that analyze compressed network traffic.

Conclusions. The compression algorithms discussed demonstrate a gradual evolution from simple, specialized methods to complex universal systems that reflect the fundamental principles of information theory. RLE is the simplest tool, the efficiency of which depends entirely on local, sequential redundancy; it is indispensable for the rapid processing of homogeneous data but is completely unsuitable for high-entropy streams.

In contrast, Huffman coding implements the principle of entropy compression by eliminating frequency redundancy through variable-length codes, ensuring optimality for a given symbol distribution, and serving as a mandatory final stage in most hybrid encoders.

Further increase in the compression ratio was achieved through the LZ77 and LZ78 dictionary methods, which eliminate structural redundancy by replacing repeating phrases with references. Thus, LZ77 with its sliding window is ideal for on-the-fly compression and local repetitions. Meanwhile, LZ78 utilizes a dynamically constructed dictionary, allowing it to effectively capture global, distant repetitions, albeit at the cost of greater complexity.

In modern computing environments, basic compression algorithms are rarely applied in their original form. Instead, modified and/or hybrid architectures dominate to achieve optimal efficiency, where the dictionary mechanism is inextricably integrated with entropy coding. Therefore, the prospects for the development of these algorithms lie in the further optimization of such combinations through hardware acceleration and the development of self-learning, adaptive dictionaries that ensure maximum efficiency for specialized data types, ranging from genomic sequences to virtual environments.

References:

1. Salomon, D. (2006). *Data Compression: The Complete Reference. 4th ed*. Springer.
2. Sayood, K. (2017). *Introduction to Data Compression. 5th ed.* Morgan Kaufmann.
3. Gao, C., Xu, X., Yang, Z., Lin, L., Li, J. (2003). QZRAM: A Transparent Kernel Memory Compression System Design for Memory-Intensive Applications with QAT Accelerator Integration. *Applied Sciences*, 13(18), 10526. https://doi.org/10.3390/app131810526.
4. Mohey, G., Zekry, A., Zakaria, H. (2021). FPGA implementation of Lempel-Ziv data compression. *Int. J. Reconfigurable & Embedded Syst*, 10(2), 99-108. https://doi.org/10.11591/ijres.v10.i2.pp99-108.
5. Rosenfeld, A., Kak, A.C. (1982). *Digital Picture Processing. 2nd ed.* Academic Press.
6. Huffman, D.A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9), 1098-1101. https://doi.org/10.1109/JRPROC.1952.273898.

7.Ziv, J., Lempel, A. (1977). A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, IT-23(3), 337-343. https://doi.org/10.1109/TIT.1977.1055714.

8.Ziv, J., Lempel, A. (1978). Compression of Individual Sequences Via Variable-Rate Coding. *IEEE Transactions on Information Theory*, IT-24(5), 530-536. https://doi.org/10.1109/TIT.1978.1055934.

Література:

1.Salomon, D. (2006). *Data Compression: The Complete Reference. 4th ed*. Springer.

2.Sayood, K. (2017). *Introduction to Data Compression. 5th ed.* Morgan Kaufmann.

3.Gao, C., Xu, X., Yang, Z., Lin, L., Li, J. (2003). QZRAM: A Transparent Kernel Memory Compression System Design for Memory-Intensive Applications with QAT Accelerator Integration. *Applied Sciences*, 13(18), 10526. https://doi.org/10.3390/app131810526.

4.Mohey, G., Zekry, A., Zakaria, H. (2021). FPGA implementation of Lempel-Ziv data compression. *Int. J. Reconfigurable & Embedded Syst*, 10(2), 99-108. https://doi.org/10.11591/ijres.v10.i2.pp99-108.

5.Rosenfeld, A., Kak, A.C. (1982). *Digital Picture Processing. 2nd ed.* Academic Press.

6.Huffman, D.A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9), 1098-1101. https://doi.org/10.1109/JRPROC.1952.273898.

7.Ziv, J., Lempel, A. (1977). A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, IT-23(3), 337-343. https://doi.org/10.1109/TIT.1977.1055714.

8.Ziv, J., Lempel, A. (1978). Compression of Individual Sequences Via Variable-Rate Coding. *IEEE Transactions on Information Theory*, IT-24(5), 530-536. https://doi.org/10.1109/TIT.1978.1055934.