

[https://doi.org/10.52326/jes.utm.2025.32\(1\).04](https://doi.org/10.52326/jes.utm.2025.32(1).04)  
UDC 003.2:512.643



## EFFICIENT STORAGE AND COMPRESSION OF COVERING ARRAYS USING ADVANCED ENCODING TECHNIQUES

Petru Cervac\*, ORCID: 0000-0001-7488-015X,  
Viorica Sudacevschi, ORCID: 0000-0003-0125-3491

*Technical University of Moldova, 168 Stefan cel Mare Blvd., Chisinau, Republic of Moldova*

\* Corresponding author: Petru Cervac, [cervac.petru@doctorat.utm.md](mailto:cervac.petru@doctorat.utm.md)

Received: 02. 23. 2025

Accepted: 03. 24. 2025

**Abstract.** This paper introduced a novel storage format for covering arrays, designed to optimize file size through efficient compression techniques. The proposed format employed Asymmetric Numeral System (ANS) encoding for array data, as well as Run-Length Encoding (RLE) and Variable Length Encoding (VLE) for metadata storage. The goal was to provide a compact, standardized format that facilitates easier sharing and utilization of covering arrays across different applications. Experimental evaluations on a dataset of 21964 covering arrays from the National Institute of Standards and Technology (NIST) demonstrated that the new format outperforms general-purpose compression algorithms such as ZIP, BZIP2, and XZ in most cases, particularly for larger covering arrays with high parameter counts. While previous work on covering array storage focused on archival and retrieval efficiency, the proposed method significantly reduces storage requirements without loss of structural integrity. The proposed method preserved the combinatorial properties of covering arrays while reducing redundancy, making it a practical alternative for large-scale combinatorial testing applications.

**Keywords:** *covering arrays, data encoding, compression, Asymmetric Numeral System, combinatorial testing.*

**Rezumat.** În această lucrare a fost propus un nou format de stocare a matricelor de acoperire, conceput pentru a optimiza dimensiunea fișierelor prin tehnici eficiente de compresie. Formatul utilizat s-a bazat pe codificarea Sistem Numeric Asimetric (SNA) pentru datele din matrice, alături de Codificarea Lungimii de Rulare (CLR) și Codificarea cu Lungime Variabilă (CLV) pentru stocarea metadatelor. Scopul a fost de a oferi un format standardizat și compact, care să faciliteze partajarea și utilizarea eficientă a matricelor de acoperire în diverse aplicații. Experimentele realizate pe un set de date de 21964 de matrice de acoperire, furnizat de Institutul Național de Standarde și Tehnologie (INST), au arătat că noul format depășește algoritmi de compresie generală, precum ZIP, BZIP2 și XZ, în majoritatea cazurilor, în special pentru matrice mari cu un număr ridicat de parametri. În timp ce lucrările anterioare privind stocarea matricelor de acoperire s-au concentrat pe eficiența arhivării și recuperării, abordarea propusă a redus semnificativ cerințele de stocare fără a compromite integritatea

structurală. Metoda prezentată a păstrat proprietățile combinatoriale ale matricelor de acoperire și elimină redundanțele, oferind o alternativă practică pentru aplicațiile de testare combinatorială la scară largă.

**Cuvinte-cheie:** *matrice de acoperire, codificare a datelor, compresie, sistem numeric asimetric, testare combinatorială.*

## 1. Introduction

Covering arrays (CA) are a powerful tool in combinatorial testing (CT), used to minimize the number of test cases required to verify a system. This approach leverages the observation that, at least for certain types of software, failures often involve combinations of relatively few parameters [1]. By ensuring that all  $t$ -way combinations of parameters are checked, a tester can significantly reduce the test suite's size while maintaining comprehensive coverage. Known as  $t$ -wise testing, this strategy ensures that all combinations of  $t$  parameters are included in the test suite. For example, 2-way testing covers all pairs of parameters, 3-way testing includes triples, and so on. Exhaustive testing would require  $k^v$  ( $k$  - number of parameters;  $v$  - number of values per parameter) test cases, while  $t$ -wise reduces their number to approximately  $v^t \log k$  ( $t$  - combinatorial strength) [2]. For a system with 10 parameters, each having 20 values, exhaustive testing would require  $10^{20}$  test cases, whereas pair-wise testing reduces this to approximately 921. CT was successfully used for verification of various systems of diverse complexity, such as compilers, protocols, networks interfaces, Graphical User Interface systems, web applications and others [3].

CT remains a niche testing technique, although it is a well-established field of research for the last 30 years. One of the major barriers to a practical adoption of CT is the lack of a standardized format for covering arrays [3]. Tools for covering arrays generation, such as ACTS [4], CTWedge [5], CAgene [6] and others, use bespoke output formats, which hampers the efficient sharing and reuse of covering arrays among researchers and practitioners. Each implementation of CT often requires generating covering arrays from scratch, which is impractical, especially for larger arrays, given that covering arrays' generation is an NP-hard problem [7]. While there have been efforts to address this issue by storing and sharing pre-generated covering arrays [8, 9], a standardized format remains the most effective solution. Striving for a such a standard would significantly enhance the practicality and adoption of CT.

The only effort to standardize the storage for covering arrays thus far is by Leithner and Simos [10]. In their work, they store covering arrays in 2 files: a \*.cca file for the array itself and a \*.ccmeta file for metadata. They also introduced a method to bundle multiple covering arrays into a single archive with a \*.ca2, alongside a tool for managing these files. Their paper briefly explored the compression of covering arrays, comparing their tool to general-purpose algorithms, though improving compression ratios was not a primary focus. Although the cost of modern storage has significantly decreased, making file size less critical in many applications, the pursuit of smaller file sizes remains a compelling research challenge. Inspired by their approach, this work proposes a new format for covering array, aimed at minimizing the file size. The format employs three compression techniques: Asymmetric Number System (ANS) encoding for covering arrays [11], and Run-Length Encoding (RLE) and Variable Length Encoding (VLE) for metadata. Testing the new format against a database of 21964 covering arrays hosted by NIST [8] demonstrated superior compression ratios compared to existing general-purpose algorithms for almost all cases.

The remainder of this work is structured as follows. In Section 2, the existing format for storage of covering arrays is reviewed, along with Leithner and Simos's format that inspired the current paper. Section 3 describes the new techniques used in the development of the new format. Section 4 exposes changes to the Leithner and Simos's format and the techniques used to minimize the size of the generated covering arrays. It also highlights the prototype implementation is detailed. In Section 5, the experimental results of the proposed method are presented and compared to three general-purpose compression algorithms: zip, bzip2, and xz. Section 6 summarizes the work and provides an outlook on future research.

## 2. Related work

The compression of covering arrays is a relatively unexplored area; the notable exception being the work by Leithner and Simos [10]. In their work, they proposed a dual-file format to store covering array data: the *cca* file for the array itself and the *ccmeta* file for metadata. Both formats are binary, designed to enhance storage efficiency. The compression process begins by converting covering arrays from common formats, such as CSV, into a streamlined "raw covering array", stripped of extraneous information. Each parameter  $i$  in the row is then encoded using  $\lceil \log_2 v_i \rceil$  bits, where  $v_i$  represents the number of values for parameter  $i$ . The size of a row is calculated by:

$$\sum_{i=1}^k \lceil \log_2 v_i \rceil, \quad (1)$$

where:  $k$  – the number of parameters;  $v_i$  – the number of values for parameter  $i$ ;  
The total size of the covering array file is calculated as:

$$N \sum_{i=1}^k \lceil \log_2 v_i \rceil, \quad (2)$$

where:  $N$  – the number of rows.

The metadata file stores the values for  $N$ ,  $t$  and each  $v_i$ . Authors allocate 64 bits for  $N$ , 8 bits for  $t$ , and 16 bits for each  $v_i$ . An additional 64 bits are used for miscellaneous information, including magic bytes, version number, and list termination. Consequently, the metadata file requires a total of  $16k + 136$  bits.

Authors evaluated their tool against general-purpose compression algorithms, zip and bzip2. The results indicate their tool outperforms zip and is competitive with bzip2 [10]. Their implementation was a prototype, suggesting that further optimizations could yield even better compression ratios.

## 3. Techniques for Improved Compression

In this section, the techniques used for the new method for compression of covering arrays are described. 3 techniques were employed: ANS-encoding for the compression of the array itself, RLE and VLE for the metadata.

ANS is a family of number systems where each symbol of a number can take a value from a set of different size. Time keeping is an example of an ANS. Each component of a time (seconds, minutes, hours, days, etc.) takes a value from a different set. There are between 1 ... 100 seconds, minutes, hours: 0 ... 59, days: 0 ... 366 etc. ANS is in the contrast with more familiar symmetric number systems, such as decimal, binary or hexadecimal, where all symbols take values from the same alphabet. An encoder utilizing an ANS tries to encode the

information into a single natural number  $x$ . Duda was the first one to explore the use of an ANS for data compression [11]. It has become more widely used since then, being incorporated in a few commercial offerings from Facebook, Microsoft, Google, and others.

RLE is a lossless data compression method where sequences of repeated data are stored as a single instance of the data along with a count of its consecutive repetitions, instead of storing the entire sequence. Several tools and algorithms use RLE for data compression, particularly in image formats like TIFF and GIF, where repeated color sequences are common. RLE is also used as part of compression schemes in video codecs like MPEG and in some lossless data compressors, where it helps optimize storage by encoding repeated data sequences efficiently.

VLE is a form of lossless data compression in which the number of bytes used to encode a number is not fixed. Examples of widely recognized VLE techniques include Huffman coding, Lempel–Ziv coding, arithmetic coding, and context-adaptive variable-length coding. The most widely use of VLE is for the encoding of characters for alphabets that exceed 256 characters. The ISO-2022 [12] family of standards define a variable length encoding for characters that use from 1 to 4 bytes to encode characters in various languages such as Japanese [13], Chinese [14], Korean [15], etc. The most popular encoding based on the VLE is the UTF-8-character encoding, which uses 1 to 4 bytes to encode a character [16].

#### 4. Methodology

To increase information density, each row of the array undergoes ANS encoding. Instead of treating each parameter separately, the approach interprets each row as a number in an ANS with alphabets  $v_1, \dots, v_k$ . The row encoded in this system is then converted to its decimal equivalent and written to the file. This approach reduces the number of bits needed to store each row compared to encoding each column individually. In the original format, the total bits required to store a row is calculated by Eq. (1). With ANS encoding, the number of bits is calculated by the following formula:

$$\left\lceil \sum_{i=1}^k \log_2 v_i \right\rceil \quad (3)$$

For example, consider a mixed covering array with 10 variables, each having 10 values. The original format requires  $10 \lceil \log_2(10) \rceil = 40$  bits, while the ANS encoding requires  $\lceil 10 \log_2(10) \rceil = 34$  bits, resulting in a 15% reduction in the bits needed to store a single row. Conversion from a number in ANS to a number in decimal is done using the following formula:

$$\sum_{i=1}^k \left( s_i \prod_{j=i+1}^k v_j \right), \quad (4)$$

where:  $k$  – the number of symbols in the representation;  $s_i$  – symbol at the position  $i$ ;  $v_j$  – the alphabet size for the position  $j$ .

The *ccmeta* format proposed in the original paper is effective for covering arrays with a small and varied number of parameters. However, during testing, the method proved inefficient for uniform or mostly uniform covering arrays with many parameters. In tests with uniform covering arrays up to 2000 parameters, the *ccmeta* file size often matched or exceeded the *cca* file size. To address this, the approach applies RLE to parameter encoding. For example, a covering array with  $k = 10$  and  $v_1 = v_2 = \dots = v_{10} = 2$  can be represented

as {10, 2} instead of {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}, reducing memory usage from 10 to 2 bytes. To maximize the number and length of runs, the approach sorts the columns in the ascending order. This is allowed as the definition of covering array doesn't restrict the column order. Then, the process counts occurrences of each value and stores them alongside their counts.

Table 1

Range of values for prefix		
Code	Bytes	Range
0x xx xx xx	1	00 00 00 00 <sub>16</sub> ... 00 00 00 7f <sub>16</sub>
10 xx xx xx	2	00 00 00 80 <sub>16</sub> ... 00 00 3f ff <sub>16</sub>
11 0x xx xx	3	00 00 40 00 <sub>16</sub> ... 00 1f ff ff <sub>16</sub>
11 10 xx xx	4	00 20 00 00 <sub>16</sub> ... 0f ff ff ff <sub>16</sub>

To store the count of occurrences, the method uses VLE, balancing practicality for arrays with varied, smaller parameters and efficiency for more esoteric cases, such as the 2000-parameter uniform arrays used in testing. The most significant bits in the length indicate the number of bytes used: 0 for 1 byte, 10 for 2 bytes, 110 for 3 bytes, and 1110 for 4 bytes. Table 1 illustrates the value ranges for each suffix. This method draws inspiration from UTF-8 Character Encoding Form [16]. This encoding supports up to  $2^{28}$  numbers with 4 bytes, sufficient for most applications. It can be extended if necessary. Most covering arrays will use 1 byte to represent column value occurrences.

Another improvement in the proposed ccmeta format is the use of a single byte to represent the number of values per column, compared to 2 bytes in the original. This limits column values to 256, sufficient for most arrays used in practice. If future needs require more than 256 values per column, the format can be adjusted using RLE or specifying byte count in the preamble, though this is deemed unnecessarily complex for now. The original file structure remained unchanged, with only the version number increased as suggested by the authors. Figure 1 shows the updated ccmeta file format.

CCMeta magic bytes – 4 bytes
CA <sup>2</sup> version – 2 bytes
Number of rows (N) – 4 bytes
Interaction strength (t) – 1 byte
Length run 1 – 1...4 bytes
Value 1 – 1 byte
⋮
Length run k – 1...4 bytes
Value k – 1 byte
Value terminator (NUL) – 1 byte

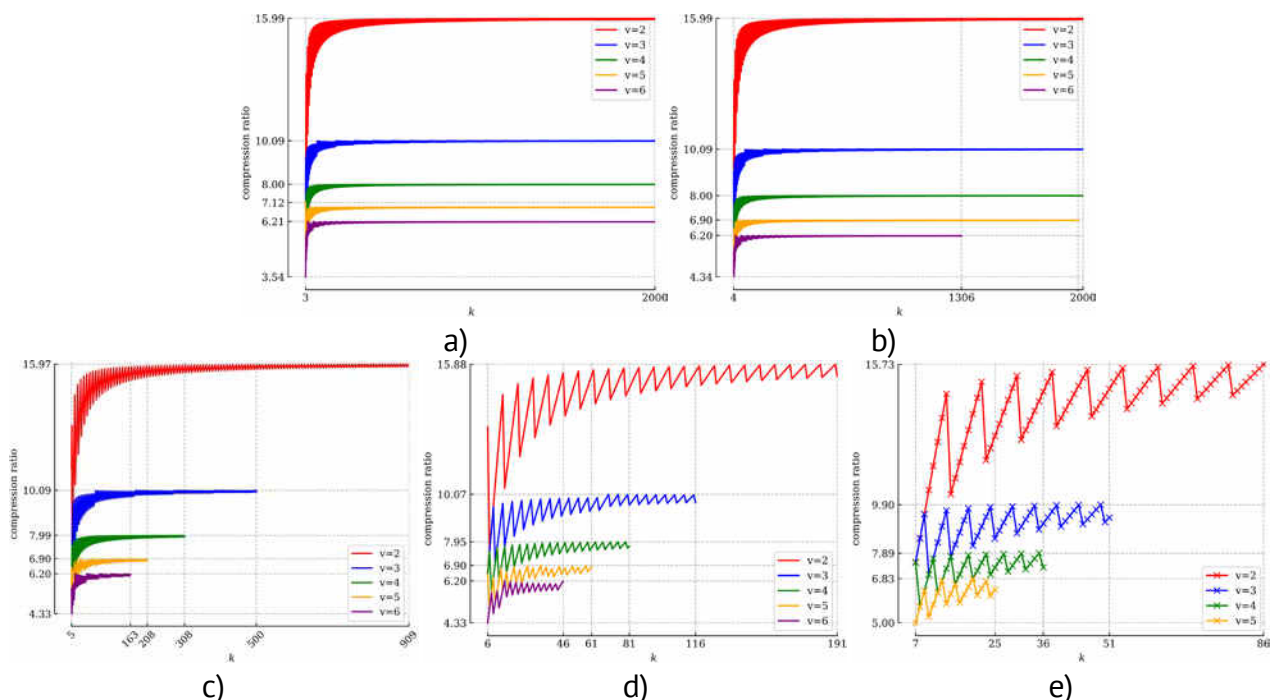
Figure 1. Updated ccmeta file format.

The method presented in this paper was implemented in C# since it is the language most familiar to the authors. The prototype is capable of parsing files in CSV and ACTS format. The prototype provides only the compression capability. The prototype is hosted on GitHub [17].

## 5. Evaluation

There are 2 challenges in analyzing the capabilities of the proposed method. The first challenge regards the representation of the covering array. A covering array can be represented in the “*extended*” form or in the “*shortened*” form. The definition of the covering arrays states each column should have a value from a set with no more than  $v$  values; however, it does not restrict the values in the set; values can be strings of arbitrary values. A covering array in “*extended*” form: [(Windows, Chrome), (Windows, Firefox), (Linux, Chrome), (Linux, Firefox)] is equivalent to the covering array in “*shortened*” form: [(0, 0), (0, 1), (1, 0), (1, 1)]. The proposed method is technically a lossy compression scheme; it preserves the structure of the covering array, but removes the information about the individual values; it is converting a CA in “*extended*” form to a CA in “*shortened*” form. The proposed method may appear more efficient when applied to covering arrays in their extended form rather than in their shortened form, as the resulting file will be smaller. However, this perception is misleading, as the resulting file technically contains less information than the covering array in extended form. To mitigate the issue, the evaluation focused only on covering arrays in shortened form.

The second challenge regards the data set of covering arrays. The proposed method is not restricted to covering arrays and can compress any random CSV value. A random CSV file is less structured than a covering array in the form of a CSV file; it can contain more rows than an actual covering array; it can have duplicated rows; it may not contain all combinations. This might serve as an advantage or disadvantage to the general-purpose compression algorithms used for comparison. To mitigate the issue, the testing restricted the data set to covering arrays only, though this reduced the total set of possible covering arrays.



**Figure 2.** Compression ratio of the new format for various  $t, k, v$ , where:  $t$  – interaction strength;

$k$  – number of parameters;  $v$  – number of values per parameter:

a)  $t = 2$ ; b)  $t = 3$ ; c)  $t = 4$ ; d)  $t = 5$ ; e)  $t = 6$ .

The test dataset consisted of 21964 covering arrays hosted by NIST. Covering arrays are stored in a custom format named ACTS. The dataset consists of various covering arrays with  $t$  and  $v$  between 2 and 6 and  $k$  up to 2000. The size and variety of the dataset allowed us to analyze the new compression scheme under diverse conditions. Although big, this dataset is not ideal, as it does not contain mixed covering arrays, which the proposed method can work with, and also more frequently used in the real world.

The proposed approach was compared against zip, bzip2 and xz, which are widely used, mature, general purpose, lossless compression algorithms. zip uses the DEFLATE algorithm [18], bzip2 uses the Burrows-Wheeler algorithm [19], and xz uses the Lempel-Ziv-Markov chain algorithm [20]. The utilities were configured to generate the smallest possible output files although, this might not fully reflect the day-to-day use of these tools.

The evaluation focused solely on output file size, ignoring other factors such as speed. The size of the \*.ccmeta file was also disregarded, as it remained under 20 bytes for any analyzed covering array and was negligible compared to the size of the \*.cca file.

Table 2

Compression ratio limit for parameter $v$	
$v$	Compression ratio in limit
2	15.99
3	10.09
4	8.00
5	7.12
6	6.21

**Note:**  $v$  – number of values per parameter

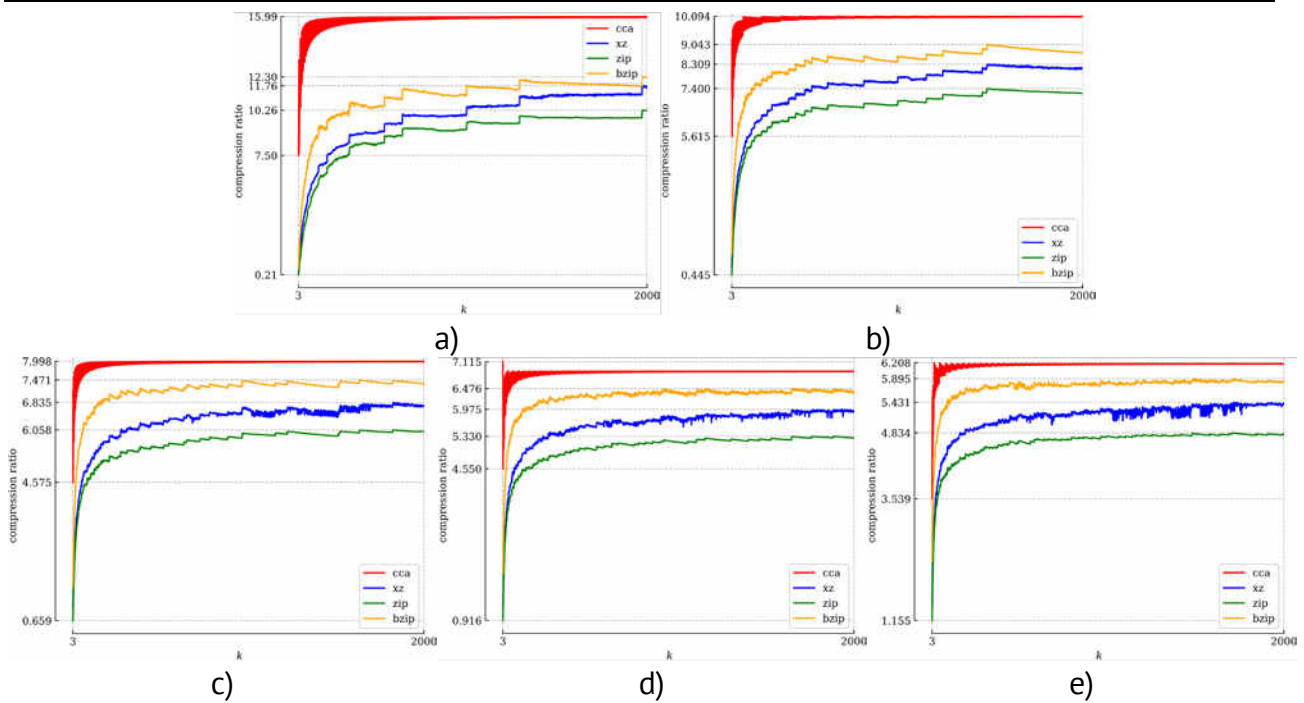
Figure 2 shows the compression ratio of the new format for the whole dataset. Compression ratio is the bigger for smaller values of  $v$ . It performs best for  $v = 2$  and worst for  $v = 6$ .  $t$  does not influence the compression ratio, which is not surprising. In the long run, the compression ratio for each  $v$  reaches a limit. Table 2 shows the compression ratio limit for each value of  $v$ .

Table 3

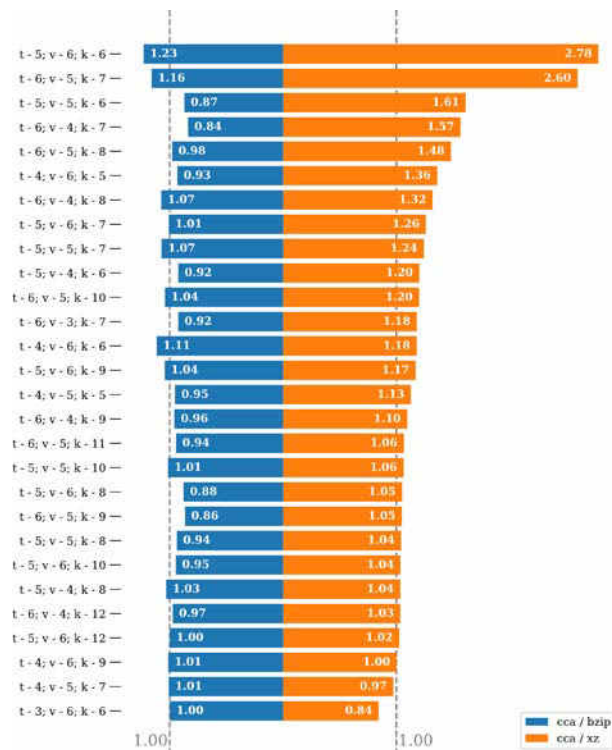
Number of swings to the top	
$v$	Number of swings to the top
2	8
3	5
4	4
5	3-4
6	3

**Note:**  $v$  – number of values per parameter

One peculiar observation is the “zigzag” behavior of the compression ratio. The compression ratio swings from smaller compression ratio to bigger one. The number of times the compression ratio comes to the maximum depends on the value of  $v$ . Table 3 shows the number of swings required to reach the top. This happens due to the way the ANS encoding is working. There is a maximum number of information that can be fit into a number. When the number is reached, the storage space has to be expanded. During the storage expansion, space is not used efficiently which explains the lower compression ratio.



**Figure 3.** Comparison ratio of the new format, xz, zip and bzip2 for  $t = 2$  and various values of  $v$ : a)  $v = 2$ ; b)  $v = 3$ ; c)  $v = 4$ ; d)  $v = 5$ ; e)  $v = 6$ .



**Figure 4.** Covering arrays for which the compression ratio of the new algorithm is smaller than general-purpose compression tools.

The compression ratio of the proposed method is in general bigger than the compression ratio of the general-purpose compression algorithms. Figure 3 show the compression ratio between the proposed method and the general-purpose algorithms. It performs best for  $v = 2$  and big values of  $k$ . The compression ratio increases as the  $k$  increases, reaching a limit. The compression ratio decreases as  $v$  increases. The proposed method, in general, performs worse for smaller covering arrays, although outperforming the



general-purpose compression algorithms. Overall, the existing compression algorithms did surprisingly well. The best performing general-purpose compression tool for covering arrays was bzip2, followed by xz and zip.

There are 28 covering arrays for which a general-purpose compression algorithm produces a smaller output file. xz and bzip2 outperforms the proposed method in the range  $k: 5 \dots 12, v: 3 \dots 6, t: 4 \dots 6$ . Figure 4 summarizes the finding.

## 6. Conclusions and Future Work

This paper presented an improvement to the CA2 format presented by Leithner and Simos by employing ANS Encoding for the CCA files and RLE and VLE for the ccmata files. These techniques showed great results compared to general purpose compression algorithms for most of the analyzed inputs, although the advantage diminishes with the increase in the number of values per column. General purpose compression algorithms, especially xz, performed better for covering array with higher  $v$  and small  $k$ , which is expected to be the majority of covering arrays used in day-to-day operations. An important aspect of the future work is performance analysis and the use of other flavors of ANS such as tANS. Decompression aspect of the method should also be implemented.

**Acknowledgement.** This paper presents the scientific results obtained within the project 020404, "Innovations in Biomedical Engineering: Advanced Technologies and Applications for Data Acquisition, Processing, and Analysis," carried out within the Department of Computer Science and Systems Engineering at the Technical University of Moldova.

**Conflicts of interest.** The authors declare no conflicts of interest.

## References

1. Kuhn, D. R.; Kacker, R. N.; Lei, Y. Introduction to Combinatorial Testing, 1st ed.. Taylor & Francis Group, London, United Kingdom, 2016, p. 3.
2. Cohen, D. M.; Dalal, S. R.; Fredman, M. L.; Patton, G. C. The AETG system: an approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.* 1997, 23, pp. 437–444.
3. Nie, C.; Leung, H. A survey of combinatorial testing. *ACM Comput. Surv.* 2011, 43, pp. 1–29.
4. Yu, L.; Lei, Y.; Kacker, R. N.; Kuhn, D. R. ACTS: A Combinatorial Test Generation Tool. In: *IEEE Sixth International Conference on Software Testing, Verification and Validation*, Luxembourg, 2013, pp. 370–375.
5. Gargantini, A.; Radavelli, M. Migrating Combinatorial Interaction Test Modeling and Generation to the Web. In: *IEEE International Conference on Software Testing Verification and Validation Workshop*, Västerås, Sweden, 9-13 April 2018, pp. 308-317.
6. Wagner, M.; Kleine, K.; Simos, D.; Kuhn, R.; Kacker, R. CAGEN: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In: *IEEE International Conference on Software Testing, Verification and Validation Workshops*, Porto, Portugal, 23-27 March 2020, pp. 191-200.
7. Kampel, L.; Simos, D. A survey on the state of the art of complexity problems for covering arrays. *Theor. Comput. Sci.* 2019, 800, pp. 107–124.
8. Forbes, M.; Lawrence, J.; Lei, Y.; Kacker, R. N.; Kuhn, D. R. Refining the In-Parameter-Order Strategy for Constructing Covering Arrays. *J. Res. Natl. Inst. Stand. Technol* 2008, 113, pp. 287-297.
9. A Library of Orthogonal Arrays. Available online: <http://neilsloane.com/oadir/> (accessed on 01.10.2024).
10. Leithner, M.; Simos, D. E. CA2: Practical Archival and Compression of Covering Arrays. In: *IEEE International Conference on Software Testing, Verification and Validation Workshops*, Valencia, Spain, 4-13 April 2022, pp. 63-67.
11. Duda, J.; Tahboub, K.; Gadgil, N. J.; Delp, E. J. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In: *Picture Coding Symposium (PCS)*, Calms, QLD, Australia, 31 May – 3 June 2015, pp. 65-69.
12. ISO/IEC 2022:1994 - Information technology – Character code structure and extension techniques. Available online: <https://www.iso.org/standard/22747.html> (accessed on 03 11 2024).

13. RFC 1468: Japanese Character Encoding for Internet Messages. Available online: <https://www.rfc-editor.org/rfc/rfc1468.html> (accessed on 07.11.2024).
14. RFC 1922: Chinese Character Encoding for Internet Messages. Available online: <https://www.rfc-editor.org/rfc/rfc1922.html> (accessed on 07.11.2024).
15. RFC 1557: Korean Character Encoding for Internet Messages. Available online: <https://www.rfc-editor.org/rfc/rfc1557.html> (accessed on 07.11.2024).
16. UTR#17: Unicode Character Encoding Model. Available online: <https://www.unicode.org/reports/tr17/> (accessed on 07.11.2024).
17. BusHero/CA2. Available online: <https://github.com/BusHero/CA2> (accessed on 01.11.2024);
18. RFC 1951: DEFLATE Compressed Data Format Specification version 1.3. Available online: <https://www.rfc-editor.org/rfc/rfc1951.html> (accessed on 07.11.2024).
19. Seward, J., bzip2 and libbzip2, version 1.0.8. Available online: <https://sourceware.org/bzip2/manual/manual.html> (accessed 05.12.2024).
20. XZ Utilities. Available online: <https://tukaani.org/xz/> (accessed on 01.12.2024).

**Citation:** Cervac, P.; Sudacevschi, V. Efficient storage and compression of covering arrays using advanced encoding techniques. *Journal of Engineering Science*. 2025, XXXII (1), pp. 47-56. [https://doi.org/10.52326/jss.utm.2025.8\(2\).04](https://doi.org/10.52326/jss.utm.2025.8(2).04).

**Publisher's Note:** JES stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:**© 2025 by the authors. Submitted for possible open access publication under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

**Submission of manuscripts:**

[jes@meridian.utm.md](mailto:jes@meridian.utm.md)