Improving Matrix-vector Multiplication via Lossless Grammar-Compressed Matrices

Paolo Ferragina
Giovanni Manzini
Department of Computer Science,
University of Pisa
name.surname@unipi.it

Travis Gagie
Faculty of Computer Science,
Dalhousie University
travis.gagie@dal.ca

Dominik Köppl
M&D Data Science Center, Tokyo
Medical and Dental University
koeppl.dsc@tmd.ac.jp

Gonzalo Navarro
IMFD, Department of Computer
Science, University of Chile
gnavarro@dcc.uchile.cl

Manuel Striani
Department of Sciences and
Technological Innovation,
University of Eastern Piedmont
manuel.striani@uniupo.it

Francesco Tosoni
Department of Computer Science,
University of Pisa
francesco.tosoni@phd.unipi.it

ABSTRACT

As nowadays Machine Learning (ML) techniques are generating huge data collections, the problem of how to efficiently engineer their storage and operations is becoming of paramount importance. In this article we propose a new lossless compression scheme for real-valued matrices which achieves efficient performance in terms of compression ratio and time for linear-algebra operations. Experiments show that, as a compressor, our tool is clearly superior to gzip and it is usually within 20% of xz in terms of compression ratio. In addition, our compressed format supports matrix-vector multiplications in time and space proportional to the size of the compressed representation, unlike gzip and xz that require the full decompression of the compressed matrix. To our knowledge our lossless compressor is the first one achieving time and space complexities which match the theoretical limit expressed by the k-th order statistical entropy of the input.

To achieve further time/space reductions, we propose column-reordering algorithms hinging on a novel column-similarity score. Our experiments on various data sets of ML matrices show that our column reordering can yield a further reduction of up to 16% in the peak memory usage during matrix-vector multiplication.

Finally, we compare our proposal against the state-of-the-art Compressed Linear Algebra (CLA) approach showing that ours runs always at least twice faster (in a multi-thread setting), and achieves better compressed space occupancy and peak memory usage. This experimentally confirms the provably effective theoretical bounds we show for our compressed-matrix approach.

PVLDB Reference Format:
Paolo Ferragina, Giovanni Manzini, Travis Gagie, Dominik Köppl, Gonzalo Navarro, Manuel Striani, and Francesco Tosoni. Improving Matrix-vector Multiplication via Lossless Grammar-Compressed Matrices. PVLDB, 15(10): 2175 - 2187, 2022.
doi:10.14778/3547305.3547321

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547321

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitlab.com/manzai/mm-repair.

1 INTRODUCTION

Matrix operations have always been important in scientific computing and engineering, and they have become even more so with the widespread adoption of ML and deep-learning tools. Very large matrices do not just present scalability challenges for their storage: they also consume a large amount of bandwidth resources in server-to-client transmissions, as well as in CPU/GPU-memory communications. Hence matrix compression appears as an attractive choice. Common simple heuristics for shrinking ML models are generally based on *lossy* compression, like low and ultra-low precision storage, sparsification (i.e., reduction of the number of non-zero values), and quantisation (i.e., reduction of the value domain). Unfortunately, lossy compression schemes often impair the ML model accuracy in a data — and algorithm — specific manner, thus requiring an attentive and manual application.

For this reason *lossless* compression represents a better alternative for achieving "automated" space savings. It is data-independent and does not require any *a priori* knowledge about the input data. In addition, if some problem domain is not sensitive to the use of a particular lossy technique, we can apply lossy compression followed by the lossless one, therefore achieving the best of both worlds. Unfortunately, traditional one-dimensional lossless compression techniques such as Huffman, Lempel-Ziv, bzip, Run-Length Encoding (RLE) often perform poorly on matrices, in that they are not able to unfold the (sometimes hidden) dependencies or redundancies between rows and columns. Moreover, they usually require the full-matrix decompression for performing the needed linear-algebra operations, thus the space reduction is only achieved in the storage or transmission, but not in the more critical computation phase.

Recently, some authors [12–14] proposed new lossless compression schemes for matrices which not only save space, but also manage to speed up linear-algebra operations, and matrix multiplication in particular. These results apply mainly to large, sparse matrices: the algorithms in [12, 13] are designed for matrices coming from ML domains, while the ones in [14] are specialised in representing binary adjacency matrices of web and social graphs.

In this paper we continue the line of research introduced in [12, 13], called *Compressed Linear Algebra* (CLA). The authors use relatively simple compression techniques (e.g., Offset-List Encoding, Run-Length Encoding, Direct Dictionary Coding) preceded by a *compression-planning* phase partitioning the columns of the input matrix into groups that can be effectively compressed together. Since ML matrices often exhibit hidden correlations (see, for instance, [13]), the combination of a careful compression planning, which is done only once, together with simple compression techniques yields good compression and fast linear-algebra operations. To improve performances, the CLA system also deploys row- and column-partitioning techniques to make the compression more cache-friendly and suitable for multithreading.

We design and experiment new lossless compression schemes for large matrices, which achieve the best performance when the input matrices are either sparse or contain a relatively small number of *distinct* values. A fundamental feature of our contribution is that our lossless compression algorithms guarantee that:

- the compression ratio is bounded in terms of the k-th order empirical entropy of the compressed sparse row/value (CSRV) representation of the input matrix; and
- the cost of the right and left matrix-vector multiplication is proportional to the size of the *compressed* matrix.

As just mentioned, saving simultaneously both time and space is not new [12–14, 22], but to our knowledge our approach is the first achieving bounds for the time and space complexities that match the theoretical limit expressed by the k-th order statistical entropy. Given its theoretical properties, our grammar-based algorithm may be used not only as a stand-alone compression tool for matrices, but also as a new powerful compression option within the CLA framework, or a similar system, in lieu of their simpler compressors.

Technically speaking, our starting point is the CSRV representation of a matrix, which is a simple modification of the well-known compressed sparse row (CSR) representation [31]. The CSRV representation is more effective than CSR when the input matrix contains relatively few distinct values. In Section 3 we show that we can compress this representation using a grammar compressor [21] so that we can later compute the right and left matrix-vector multiplication by working directly upon the compressed matrix, and within time and working space proportional to the compressed size of that matrix. We tested our proposal in practice with a prototype described in Section 4 using the RePair [28] grammar compressor over eight matrices resulting from real ML problems. As for the compression ratios, the experiments show our tool is clearly superior to gzip, and that it is usually within 20% of xz; in addition, our solution is designed to offer support for matrix-vector multiplications directly over the compressed file, whereas gzip and xz cannot.

To measure the space usage of our matrix-multiplication algorithms, we tested a sequence of left and right vector-matrix multiplications and found that the peak memory usage for our multithreaded algorithms is for most inputs between 6% and 50% of the size of the uncompressed matrix. These results confirm the theoretical finding that grammar compression can indeed save a significant amount of space during the computation, and therefore allows us to work with larger data sets in internal memory.

In the second part of the paper we add an algorithmic step to our grammar-based compression scheme to obtain an even greater space saving. As pointed out in [12], ML matrices often exhibit correlations between columns; this phenomenon is likely to make the same combination of values appear in the same columns in multiple rows. Most compressors are able to exploit the presence of identical values only when they occur in contiguous columns. Nonetheless, in real-world data sets correlated columns often appear far apart from each other. For this reason, the matrix compression scheme of CLA [12] features a preliminary step aimed at discovering groups of correlated columns; then, such groups are compressed independently of one another, possibly choosing a different compressor for each group. We hence study the problem of column reordering under the hypothesis that the subsequent compression phase is implemented via a grammar compressor. The column-reordering problem for binary, categorical, and general matrices attracted a lot of interest in the past because of its applications to compressing tables arising from several contexts, such as data warehouses [5, 6, 35], biological experiments [1], mobile data [17], and graph DBs [20], just to cite a few. Discovering dependency relations among matrix columns and finding the order that guarantees the smallest compressed output is an NP-hard problem in its general form (cf. e.g. [5]). Thus, all of the papers above use heuristics to efficiently find appropriate column permutations. In all cases, the key step lies in defining a proper measure of *column similarity* accounting for the special features of the problem and of the compressor at hand.

In Section 5, we present a column-similarity score designed for our lossless grammar-based compressors for matrices. Then, we describe four new column-reordering algorithms that hinge on this score and, to boost compression, we apply them to row blocks which are finally compressed individually. We test the effectiveness of this combination over the same eight ML matrices mentioned before. Experimental figures show that, without worsening the running time, we can achieve a further reduction of up to 16% of the peak memory usage during matrix-vector multiplications.

As a final contribution of this paper, we compare our matrix compressor to the one of Compressed Linear Algebra (CLA) system, which constitutes the state-of-the-art in this setting [12, 13]. As for compression, experiments show that our approach is more effective than CLA over 7 matrices (out of the 8 we tested), with an (absolute) space improvement of up to 10%. The space improvement is even greater if we consider the peak memory usage during matrix-vector multiplications, being a factor between 3.14 and 19.12. In terms of running time, CLA is always at least two times slower than our compressors. These results were obtained using 16 threads for our compressors, whereas CLA was set to use all the available threads (the testing machine supports up to 80 independent threads).

Summing up, our experiments show that: (1) our grammar-based compressors for matrices do indeed achieve a better space reduction than the state of the art, and (2) our theoretical results ensuring that the number of operations is bounded by the size of the compressed matrix translate into algorithms that are also fast in practice; indeed for the most compressible matrices experiments show our algorithms are even faster than the algorithm working directly on the uncompressed matrix. As a final note, we point out that CLA is a general framework offering compressed linear algebra for ML systems which, by design, is not tied to a particular compression technique. Hence, we envision our compressors could be adopted not only as stand-alone compression tools for matrices but also as a new powerful compression option within the CLA framework.

1.1 Transparency and Reproducibility

All source files of our algorithms, as well as the scripts to reproduce the experimental results, are available at the repository https://gitlab.com/manzai/mm-repair. The data sets are available at the public Kaggle repository [26].

2 THE COMPRESSED SPARSE ROW/VALUE REPRESENTATION

Given a matrix $M \in \mathbf{R}^{n,m}$ with n rows and m columns, the compressed sparse row (CSR) representation [31] is a classic scheme taking advantage of the matrix sparsity. If the matrix M contains t non-zero elements, the CSR representation consists of 1) a length-t array nz listing the non-zero elements row-by-row; 2) a length-t array idx storing for each element in nz its column index; 3) a length-n array first such that first$[1] = 0$, and first$[i]$ with $2 \leq i \leq n$ equals the number of non-zero terms in the first $i-1$ rows (this information is used for partitioning the elements of nz by rows).

If the number of *distinct* non-zero values is relatively small, then it is more space efficient to introduce an additional array $V[1, k]$ containing the distinct non-zero elements of M and to store in nz not the actual non-zero values but their indices in V. If there are, say, fewer than 2^{16} distinct non-zero elements, then each entry in nz takes only 2 bytes instead of the 8 bytes of a double: this saving can more than compensate for the extra cost of storing the array V. This representation as a whole is called CSR-IV in [22].

In this paper we introduce a new representation, called *Compressed Sparse Row/Value* (CSRV), by making two minor modifications to the above scheme. Firstly, we combine the two length-t arrays nz and idx in a single vector of pairs S, such that for $i = 1, \ldots, t$, entry $S[i]$ contains the pair of integers (nz$[i]$, idx$[i]$). Secondly, instead of storing a separate array first we include its information in S by storing a special symbol \$ immediately after the last non-zero entry of each row. As a result, the array S now has length $t + n$, which can be obtained by scanning the matrix M row-by-row: for each entry $M[i][j] \neq 0$ we append to S the pair $\langle \ell, j \rangle$, where ℓ is the index in V such that $V[\ell] = M[i][j]$. In addition, at the end of each row we append to S the special symbol \$. During the scanning, for each nonzero $M[i][j]$ we need to retrieve the index ℓ such that $V[\ell] = M[i][j]$, or to add $M[i][j]$ to V if no such index exists. Storing the association between values in V and their index in a hash table with constant amortised time per operation, we have the following result.

Lemma 2.1. *The construction of the CSRV representation of a matrix $M \in \mathbf{R}^{n,m}$ takes $O(mn)$ time.* □

Figure 1 reports an example in which the elements of V are sorted according to their size, but any other ordering (or no ordering at all) would have worked equally well. Also, the elements of S within the same row can be reordered without loss of information; this latter property will be used in Section 5 to improve compression.

Given the CSRV representation of matrix M and a vector $x[1, m]$, it is straightforward to perform the $y = Mx$ multiplication with a single scan of S. To begin with, we initialise the vector $y[1, n]$ to zero. Then, during the scan of row i, when we encounter the pair $\langle \ell, j \rangle$ we add the value $V[\ell] \cdot x[j]$ to the entry $y[i]$. The occurrences of the symbol \$ allow us to keep track of the current row. We can similarly compute with a single scan of S the left-multiplication $x^t = y^t M$: firstly, we initialise $x[1, m]$ to zero; then, during the scan of row i, when we encounter the pair $\langle \ell, j \rangle$ we add the value $y[i] \cdot V[\ell]$ to the entry $x[j]$. Hence, either right and left multiplications can be computed in $O(|S|) = O(n + t)$ time. Hereinafter we use the notation (S, V) to denote the CSRV representation outlined above.

3 GRAMMAR-COMPRESSED MATRICES

We show how to compress the CSRV representation (S, V) of a matrix M with an algorithm that, for compressible matrices, provably yields a reduction in both the space occupancy and in the cost of the left and right matrix-vector multiplication operations.

Recall that a grammar-compressed representation for a string T over an alphabet of terminal symbols Σ is a context-free grammar that generates only T [7]. For simplicity, we assume the grammar is a so-called *straight-line program* [24] (SLP), that is, it consists of a set of rules of the form $L_i \to R_{i_1} R_{i_2}$, where L_i is a nonterminal and each of R_{i_1} and R_{i_2} can be either a terminal (i.e., an element of the base alphabet Σ), or a nonterminal. The grammar generates only T, implying that each nonterminal appears as the left-hand side of a single rule; we can thus identify each rule with the nonterminal on its left-hand side. Given a nonterminal N_j, its *expansion*, denoted by $\exp(N_j)$, is defined as the (unique) sequence obtained by repeatedly applying the substitution rules of the SLP grammar until we are left with a string of Σ. Thus one can leverage a SLP to represent T as a succinct sequence C of nonterminals; one can then retrieve T from C by expanding the nonterminals. The grammar compressor outputs a set of rules and a special nonterminal whose expansion generates only the input string T. If the rules are q, the nonterminals $N_1, \ldots, N_q$ are q too, and we can number them so that if N_i appears in the right-hand side of N_j, then $i < j$.

One can define the size of a grammar as the sum of the lengths of the right-hand sides of the rules. The same text T can be generated by many different grammars, and finding the smallest one is NP-complete [7, 33]. Yet, the compressors producing irreducible grammars, among them Greedy, LongestMatch [21], RePair [23], and Sequential [21], are guaranteed to produce an output whose size is bounded by $|T|H_k(T) + o(|T|H_k(T))$ bits for any $k \in o(\log_\sigma |T|)$, where σ is the size of the input alphabet and $H_k(T)$ is the order-k statistical entropy of the input T [30]. Up to lower order terms, then, these grammar compressors are as good as the best statistical encoders that compress the input on the basis of the frequencies of k-tuples of symbols. Grammar compressors are also very effective for compressing strings with many repetitions: in this case their output size can be within a logarithmic factor from the output of the best compressors based on LZ-parsing; see [29] for details.

To compress a CSRV representation (S, V) we apply a grammar compressor to the sequence S. We modify the compressor so that it never uses the special terminal symbol \$ in any rule. This guarantees that the expansion of any nonterminal N_k only contains pairs $\langle i, j \rangle$.

$$\begin{bmatrix} 1.2 & 3.4 & 5.6 & 0 & 2.3 \\ 2.3 & 0 & 2.3 & 4.5 & 1.7 \\ 1.2 & 3.4 & 2.3 & 4.5 & 0 \\ 3.4 & 0 & 5.6 & 0 & 2.3 \\ 2.3 & 0 & 2.3 & 4.5 & 0 \\ 1.2 & 3.4 & 2.3 & 4.5 & 3.4 \end{bmatrix}$$

$$\begin{aligned} V &= [1.2 \quad 1.7 \quad 2.3 \quad 3.4 \quad 4.5 \quad 5.6] \\ S &= \langle 1{,}1\rangle\,\langle 4{,}2\rangle\,\langle 6{,}3\rangle\,\langle 3{,}5\rangle\,\$\,\langle 3{,}1\rangle\,\langle 3{,}3\rangle\,\langle 5{,}4\rangle\,\langle 2{,}5\rangle\,\$ \\ &\quad \langle 1{,}1\rangle\,\langle 4{,}2\rangle\,\langle 3{,}3\rangle\,\langle 5{,}4\rangle\,\$\,\langle 4{,}1\rangle\,\langle 6{,}3\rangle\,\langle 3{,}5\rangle\,\$ \\ &\quad \langle 3{,}1\rangle\,\langle 3{,}3\rangle\,\langle 5{,}4\rangle\,\$\,\langle 1{,}1\rangle\,\langle 4{,}2\rangle\,\langle 3{,}3\rangle\,\langle 5{,}4\rangle\,\langle 4{,}5\rangle\,\$ \end{aligned}$$

Figure 1: A matrix and its CSRV representation. In the array S the symbol $\langle 3{,}1\rangle$ stands for an occurrence of the value $V[3] = 2.3$ in column 1. Note that the same value in column 3, is represented instead by $\langle 3{,}3\rangle$. Only the same values in the *same column* are represented by the same pair $\langle i{,}j\rangle$.

$$\begin{aligned} \mathcal{R} = \{ & N_1 \to \langle 3{,}3\rangle\,\langle 5{,}4\rangle \quad N_2 \to \langle 1{,}1\rangle\,\langle 4{,}2\rangle \quad N_3 \to \langle 3{,}1\rangle\, N_1 \\ & N_4 \to \langle 6{,}3\rangle\,\langle 3{,}5\rangle \quad N_5 \to N_2\, N_4 \quad N_6 \to N_3\,\langle 2{,}5\rangle \\ & N_7 \to N_2\, N_1 \quad N_8 \to \langle 4{,}1\rangle\, N_4 \quad N_9 \to N_7\,\langle 4{,}5\rangle \ \} \\ C = {} & N_5\,\$\,N_6\,\$\,N_7\,\$\,N_8\,\$\,N_3\,\$\,N_9\,\$ \end{aligned}$$

Figure 2: The set of rules $\mathcal{R}$ and the final string C whose expansion is the sequence S from Figure 1.

As a result, the output of the grammar compressor applied to S consists of a set of rules $\mathcal{R}$ and a string

$$C = N_{i_1}\$N_{i_2}\$\cdots N_{i_n}\$ \tag{1}$$

such that each N_{i_j} is a nonterminal whose expansion is the sequence of pairs representing the non-zero elements of row j. In the same sense, the expansion of the string C (i.e., expanding each of its nonterminals) is the sequence S. An example of a grammar representing the string S of Figure 1 is given in Figure 2. In the following we write $(C, \mathcal{R}, V)$ to denote the grammar representation of (the CSRV representation of) a matrix M.

3.1 Right Multiplication for Grammar-Compressed Matrices

In this section we show that, given a grammar representation $(C, \mathcal{R}, V)$ of a matrix M, we can compute the right multiplication $y = Mx$ in $O(|\mathcal{R}| + |C|)$ time using $O(|\mathcal{R}|)$ words of auxiliary space. In the following we use S to denote the expansion of C, so that (S, V) is the CSRV representation of M.

Definition 3.1. Given a vector $x[1, m]$ and a pair $\langle \ell{,}j\rangle \in S$ we define

$$\mathrm{eval}_x(\langle \ell{,}j\rangle) = V[\ell] \cdot x[j];$$

(recall that the pair $\langle \ell{,}j\rangle$ represents the value $V[\ell]$ stored in column j of matrix M). Similarly, for a nonterminal N_i whose expansion is $\langle \ell_1{,}j_1\rangle\,\langle \ell_2{,}j_2\rangle\cdots\langle \ell_h{,}j_h\rangle$ we define

$$\mathrm{eval}_x(N_i) = \sum_{k=1}^{h} \mathrm{eval}_x(\langle \ell_k{,}j_k\rangle) = \sum_{k=1}^{h} V[\ell_k]\; x[j_k]. \tag{2}$$

From the above definition we immediately get

Lemma 3.2. *If the grammar contains the rule $N_i \to AB$, then* $\mathrm{eval}_x(N_i) = \mathrm{eval}_x(A) + \mathrm{eval}_x(B)$. □

Lemma 3.3. *Given the representation $(C, \mathcal{R}, V)$ of a matrix $M \in \mathbf{R}^{n,m}$ with $C = N_{i_1}\$\cdots N_{i_n}\$$, if $y = Mx$ then it holds that $y[r] = \mathrm{eval}_x(N_{i_r})$, for $r = 1, \ldots, n$.*

Proof. We have $y[r] = \sum_{i=1}^{m} M[r][i] \cdot x[i]$. By construction, the expansion of the nonterminal N_{i_r} is the sequence of pairs $\langle \ell_1{,}j_1\rangle\cdots\langle \ell_h{,}j_h\rangle$ representing all the non-zero elements of row r where, for $k = 1, \ldots, h$, ℓ_k denotes the position in V containing the value $M[r][j_k]$. Thus

$$y[r] = \sum_{k=1}^{h} M[r][j_k] \cdot x[j_k] = \sum_{k=1}^{h} V[\ell_k] \cdot x[j_k] \; = \; \mathrm{eval}_x(N_{i_r}).$$

□

Theorem 3.4. *Given the grammar-compressed CSRV representation $(C, \mathcal{R}, V)$ of a matrix $M \in \mathbf{R}^{n\times m}$ and a vector $x \in \mathbf{R}^m$, we can compute $y = Mx$ in $O(|C| + |\mathcal{R}|)$ time using $O(|\mathcal{R}|)$ words of auxiliary space.*

Proof. To compute $y = Mx$, we introduce an auxiliary array $W[1, q]$, where $q = |\mathcal{R}|$, such that $W[i] = \mathrm{eval}_x(N_i)$. Because of Lemma 3.2 and of the rule ordering, we can fill W with a single pass over $\mathcal{R}$ in time $O(q)$: the value $W[i] = \mathrm{eval}_x(N_i)$ is the sum of two terms that can be either of the form $\mathrm{eval}_x(\langle h{,}k\rangle)$ or $\mathrm{eval}_x(N_j)$ with $j < i$. In the former case $\mathrm{eval}_x(\langle h{,}k\rangle) = V[h] \cdot x[k]$; in the latter case $\mathrm{eval}_x(N_j) = W[j]$ for some already-computed entry, since $j < i$. One may indeed observe that N_i's are ranked by the time they are computed. After filling W, we use Lemma 3.3 to determine the components of the output vector y. □

3.2 Left multiplication for grammar-compressed matrices

We now show that, given the grammar representation $(C, \mathcal{R}, V)$ of a matrix M, we can compute the left multiplication $x^t = y^t M$ with an algorithm symmetrical to the one for the right multiplication and within the same time and space bounds.

Definition 3.5. For any $\langle \ell{,}j\rangle \in S$ we define $\mathrm{rows}(\langle \ell{,}j\rangle)$ as the set of rows whose CSRV representation contains $\langle \ell{,}j\rangle$. Note that $k \in \mathrm{rows}(\langle \ell{,}j\rangle)$ if, and only if, the expansion of the nonterminal $N_{i_k} \in C$ contains the pair $\langle \ell{,}j\rangle$ or, equivalently, $M[k][j] = V[\ell]$.

For the example in Figure 1, we have $\mathrm{rows}(\langle 1{,}1\rangle) = \{1, 3, 6\}$ since $\langle 1{,}1\rangle$ represents the value 1.2 that appears in column 1 of those three rows. Similarly, $\mathrm{rows}(\langle 3{,}1\rangle) = \{2, 5\}$.

Definition 3.6. Given a vector $y[1,n]$, for any $\langle i,j\rangle \in S$ we define $\mathsf{sum}_y(\langle i,j\rangle)$ as

$$\mathsf{sum}_y(\langle i,j\rangle) = \sum\nolimits_{k \in \mathrm{rows}(\langle i,j\rangle)} y[k]$$

Lemma 3.7. *Given the CSRV representation (S,V) of matrix $M \in \mathbf{R}^{n\times m}$, let S' be the set of distinct symbols in S (i.e., without duplicates). If $x^t = y^t M$ then, for $j = 1,\ldots,m$, it holds that*

$$x[j] = \sum\nolimits_{\langle i,j\rangle \in S'} V[i] \cdot \mathsf{sum}_y(\langle i,j\rangle)$$

(one should notice that the summation involves only pairs in S' with second component j).

Proof. Since

$$x[j] = \sum\nolimits_{\ell=1}^{n} y[\ell] \cdot M[\ell][j],$$

the value $x[j]$ depends only upon the non-zero elements in column j. Each nonzero in column j is represented by a symbol $\langle i,j\rangle$ and has its corresponding value encoded by some entry $V[i]$. If $\langle i,j\rangle$ occurs at row r in column j, then $y[r]$ is multiplied by $V[i]$, and this holds for all rows containing $\langle i,j\rangle$. One can aggregate these multiplications and write them as $V[i] \cdot \mathsf{sum}_y(\langle i,j\rangle)$. The lemma follows by iterating this argument over all distinct non-null values $V[i]$ occurring in column j, and therefore over all pairs $\langle i,j\rangle \in S'$. □

We now show that the notions of rows and sum can be naturally extended to nonterminals.

Definition 3.8. Given the representation $(C,\mathcal{R},V)$ of a matrix $M \in \mathbf{R}^{n\times m}$, for each nonterminal N_j we define $\mathrm{rows}(N_j)$ as the set of row indices ℓ such that N_j appears in the expansion of N_{i_ℓ}. In other words, $\mathrm{rows}(N_j)$ denotes the rows whose compression makes use of N_j. We also define $\mathsf{sum}_y(N_j) = \sum_{\ell \in \mathrm{rows}(N_j)} y[\ell]$.

In the following we make the natural assumption that the grammar does not contain *useless* rules, that is, if the grammar contains the rule $N_i \to AB$, then N_i appears in the right-hand side of some other rule (whose left-hand side will be some N_j with $j > i$), or N_i appears in the final string C (or both).

Lemma 3.9. *For any symbol α (terminal or nonterminal), let $\mathcal{R}_\alpha$ denote the set of nonterminals N_j's such that their defining rule $N_j \to AB$ contains α in their right-hand side (i.e., $A = \alpha$ or $B = \alpha$), and let $\mathcal{I}_\alpha$ denote the set of row indices ℓ such that $N_{i_\ell} = \alpha$ (hence $\ell \in \mathcal{I}_\alpha$ iff the expansion of α coincides with the ℓ-th row). Then,*

$$\mathsf{sum}_y(\alpha) = \sum\nolimits_{N_j \in R_\alpha} \mathsf{sum}_y(N_j) \;\; + \;\; \sum\nolimits_{\ell \in \mathcal{I}_\alpha} y[\ell]. \qquad (3)$$

Proof. Since each occurrence of α is either the right-hand side of a single rule, or coincides with some N_{i_α}, we have

$$\mathrm{rows}(\alpha) = \left\{ \bigcup\nolimits_{N_j \in \mathcal{R}_\alpha} \mathrm{rows}(N_j) \right\} \bigcup_{\alpha} \mathcal{I}_\alpha$$

and the lemma follows by induction on the number of steps in the derivation of α. □

In view of Lemma 3.7, to compute $x^t = y^t M$, we need to compute $V[i] \cdot \mathsf{sum}_y(\langle i,j\rangle)$ for all $\langle i,j\rangle \in S'$. To this end we first compute sum_y for nonterminals and then we use Lemma 3.9 to derive the values $\mathsf{sum}_y(\langle i,j\rangle)$. In our implementation we introduce an auxiliary array $W[1,q]$, where $q = |\mathcal{R}|$, such that at the end of the computation $W[i]$ contains $\mathsf{sum}_y(N_i)$. To explain: we initially set $x[1,m]$ to zero, and we set $W[1,q]$ to zero as well, except for the entries $W[i_\ell]$ that we initialise to $y[\ell]$ for every nonterminal N_{i_ℓ} in the final string C (this accounts for the terms in the second summation of (3)). Next, we scan the set of rules *backwards* from q to 1; for every rule $N_j \to AB$ we proceed as follows:

- if A (or B) is equal to another nonterminal N_i (necessarily with $i < j$) we increase $W[i]$ by the value $W[j]$;
- if A (or B) is equal to a terminal $\langle h,k\rangle$ we increase $x[k]$ by $V[h] \cdot W[j]$.

The crucial observation is that when we reach the rule $N_j \to AB$ we have already computed in $W[j]$ the correct value $\mathsf{sum}_y(N_j)$ since we have already accounted for all terms in Lemma 3.9, namely the nonterminals in the final string C and all rules containing N_j in their right-hand side (by our assumptions these rules will be numbered higher than j). Using our strategy, the value $\mathsf{sum}_y(N_j)$ is added to $\mathsf{sum}_y(A)$ and $\mathsf{sum}_y(B)$, affecting their corresponding values in W if they are nonterminal, or being accumulated in the proper entry of x if they are terminals.

Theorem 3.10. *Given the grammar-compressed CSRV representation $(C,\mathcal{R},V)$ of a matrix $M \in \mathbf{R}^{n\times m}$ and a vector $y \in \mathbf{R}^n$, we can compute $x^t = y^t M$ within $O(|C| + |\mathcal{R}|)$ time using $O(|\mathcal{R}|)$ words of auxiliary space.* □

We point out that we do not require that in the array S, compressed to C and $\mathcal{R}$, the pairs relative to the same row are ordered according to column index, as we arranged them in Figure 1. To help the compression, we could instead reorder the pairs in other ways: this would not impact upon the design of our multiplication algorithms. In Section 5, we analyse the compression improvement obtained by reordering the columns of M *globally*, i.e., reordering the elements in each row using the same permutation. As for future work, we plan to analyse the general problem in which the elements in each row are reordered independently of all other rows.

4 IMPLEMENTATION AND EXPERIMENTS

We now describe a prototype of our matrix-multiplication algorithm for grammar-compressed matrices. We derive different representations with different time/space trade-offs, so that in the end we will eventually define a *family* of grammar-compression algorithms.

Given a matrix $M \in \mathbf{R}^{n,m}$ we first build the CSRV representation (S,V) as described in Section 2. We implemented this representation by storing the sequence S as an array of 32-bit unsigned integers: the symbol \$ is encoded by the integer 0, while the pair $\langle i,j\rangle$ is encoded by the integer $1 + im + j$ (recall $0 \le j < m$ is the column index). The entries of V are represented as 8-byte doubles, so the total space usage amounts to $4|S| + 8|V|$ bytes. In the following we call this representation csrv and we use it as a baseline for our tests.

To build the grammar representation $(C,\mathcal{R},V)$ we compress the 32-bit integer sequence S using the RePair algorithm [23], which runs in $O(|S|)$ time, using $O(|S|)$ words of space, and achieves a compression ratio bounded by the high-order statistical entropy of S (see Sect. 3). RePair works by repeatedly finding the most frequent pair of consecutive symbols AB, replacing all their occurrences by a new nonterminal N_i, and appending the rule $N_i \to AB$ to the current rule set. We modified RePair so that it never builds a rule involving the symbol \$, as required by our construction. RePair stops when there are no more pairs of consecutive symbols appearing more than once. Thus, the final string C has not necessarily the form $N_{i_1}\$N_{i_2}\$\cdots N_{i_n}\$$ discussed in the previous section; instead C is usually longer and may even include terminals $\langle i,j\rangle$. We could add additional rules to obtain a final string C with exactly $2n$ symbols as above; but since this does not help compression or running times we use RePair's final string as C, adding the (simple) necessary modifications to the multiplication algorithm.

In addition to the final string C, RePair produces a set of rules $\mathcal{R}$ where, as we saw, each rule is represented by a symbol pair. In its naïve representation, RePair outputs $|C|+2|\mathcal{R}|$ 32-bit integers overall.[1] However, this is quite a wasteful representation: if the largest nonterminal is represented by the integer $N_{\max}$, we can represent C and $\mathcal{R}$ using packed arrays with entries of $w = 1 + \lfloor \log_2 N_{\max} \rfloor$ bits. What's more, some symbols might be more frequent than others in C or $\mathcal{R}$, so we can save additional space by using a variable-length representation via an entropy coder. We have thus experimented with the following variants of RePair compression, which induce three corresponding variants of our matrix compression algorithm:

- re_32**:** C and $\mathcal{R}$ are represented as 32-bit integer arrays. This is the fastest, but less space-efficient representation.
- re_iv**:** C and $\mathcal{R}$ are represented as packed arrays, with entries of $1 + \lfloor \log_2 N_{\max} \rfloor$ bits (see above). In our implementation we used the class `int_vector` from the sdsl-lite library [16].
- re_ans**:** $\mathcal{R}$ is represented via a packed array as above, whereas C is compressed using the `ans-fold` entropy coder from [28].

All the above variants store the array V uncompressed. Clearly, more complex representations are possible, offering even larger compression achievements. However, the reader should notice two important points. Firstly, we want to efficiently support matrix-vector multiplication: looking at the algorithms in Section 3 we see that the left-multiplication algorithm scans the rules in $\mathcal{R}$ backwards, and only a few compressors provide fast right-to-left access to uncompressed data. In addition, the compression of C and $\mathcal{R}$ is secondary: we expect the largest saving from the use of the grammar compressor and reordering techniques introduced in Section 5.

4.1 Multi-threaded implementation

To take advantage of modern multi-core architectures, matrix multiplication algorithms usually split the input matrices into blocks; indeed, most operations on the individual blocks can be easily carried out in parallel on a multi-thread machine. Since for ML matrices the number of observations (rows) is much larger than the number of features (columns), we implemented a representation in which the input matrix is partitioned into blocks of rows. Given a parameter $b > 1$, an $r \times c$ matrix M is partitioned into b blocks of size $\lceil r/b \rceil \times c$ (except for the last block which might have fewer rows). With this setting, the right multiplication $y = Mx$ consists of b independent right multiplications each one involving a single block. The b left multiplications computing $x^t = y^t M$ are independent too; in a final step the b resulting row vectors are summed together.

Our grammar-based representations can be easily adapted to work with distinct blocks of rows. After computing the CSRV representation (S, V), we partition the vector S into b subvectors $S_1, \ldots, S_b$, so that S_i contains the encoding of the non-zero elements of the i-th row block. We thereby grammar-compress each subvector S_i using RePair; the resulting string C_i and rule set $\mathcal{R}_i$ are then further compressed as described before. Notice that the value array V is unique and shared by all matrix blocks.

4.2 Some experimental figures

We executed all our experiments on a machine equipped with 80 Intel(R) Xeon Gold 6230 CPUs running @ 2.10 GHz, with 360 GB of RAM. We measured running times and peak space usages with the Unix tool `time`. Table 1 reports the features of our data set; it includes all the matrices from [12, 13] and two other matrices (*Susy* and *Optical*) coming from the ML repository [11] thus offering a wide spectrum of matrix-types that allow us to better investigate the algorithmic features and performance of all algorithms we tested. For uniformity's sake, we represent the entries of all matrices as 8-byte doubles, so the uncompressed and full representation of a matrix takes a total of rows × cols × 8 bytes. If such representation is compressed with gzip and xz, with their default compression level, the resulting compressed files have the sizes reported in columns 6 and 7 of Table 1. Column 8 reports the size of the csrv representation, while the last three columns report the sizes of the three variants of our RePair compressor described above. All sizes are given as a percentage of the ratio between the size of the compressed and the uncompressed matrix representations (rows × cols × 8 bytes), hence a smaller percentage corresponds to a better compression.

We emphasise that some of the matrices, namely *Susy*, *Higgs*, and *Optical*, are not really sparse, having more than 92% non-zero elements. The classical CSR representation, where each non-zero entry takes 12 bytes, would take, on these data sets, more space than the uncompressed representation. Our csrv representation, that takes advantage of repeated values, is already obtaining some compression; in particular for *Optical*, which has fewer distinct nonzeros, csrv shows a reduced space footprint compared to gzip. Further space reduction is obtained by our advanced grammar-based compressors, even for non-sparse matrices, thus achieving space reduction on a larger class of matrices with some structure.

The comparison between the csrv and re_32 output sizes is of interest to see some indication of the effectiveness of grammar compression. At one extreme, we see that re_32 does not provide for *Susy* any additional compression to the csrv representation, suggesting that there are not many pairs of adjacent non-zero values occurring many times in different rows. At the other extreme, re_32 provides for *Census* a six-fold better compression, and re_iv and re_ans achieve a compression even better than the state-of-the-art tool xz. Moreover, our most sophisticated encoder, re_ans, is significantly better than gzip, with the only exception being *Susy*.

Let us now turn our attention to our main interest, namely reducing *both* space usage and running time for the matrix multiplication operations. Standard compressors, like gzip and xz, need to fully decompress the compressed matrix in order to perform any operation on it. Hence, the cost of any operation is at least proportional to the size of the *uncompressed* matrix; conversely, in the previous section we proved that using grammar compression, left and right multiplications can be carried out in time proportional to the size

[1]Notice that for a rule $N_i \rightarrow AB$, we have to encode only A and B because the nonterminals N_i have increasing ids.

Table 1: Matrices used in our experiments and the compression ratio achieved by the tools described in the text; a smaller percentage corresponds to a better compression. The column *nonzeros* reports the percentage of non-zero elements over the total, while column #|*nonzeros*| reports the number of *distinct* non-zero values.

matrix	rows	cols	*nonzeros*	#|*nonzeros*|	gzip	xz	csrv	re_32	re_iv	re_ans
Susy [11]	5 000 000	18	98.82%	20 352 142	53.27%	43.94%	74.80%	74.80%	69.91%	66.63 %
Higgs [11]	11 000 000	28	92.11%	8 083 943	48.38%	31.47%	50.46%	46.91%	41.38%	38.05 %
Airline78 [2]	14 462 943	29	72.66%	7 794	13.27%	7.01%	38.06%	14.84%	11.13%	9.27 %
Covtype [11]	581 012	54	22.00%	6 682	6.25%	3.34%	11.95%	7.21%	4.52%	3.87 %
Census [11]	2 458 285	68	43.03%	45	5.54%	2.79%	22.25%	3.24%	2.02%	1.53 %
Optical [11]	325 834	174	97.50%	897 176	53.54%	27.13%	50.62%	40.70%	35.81%	34.31 %
Mnist2m [4]	2 000 000	784	25.25%	255	6.46%	4.25%	12.69%	7.47%	5.84%	5.33 %
ImageNet [8]	1 262 102	900	30.99%	824	5.52%	3.63%	11.72%	6.41%	4.00%	3.86%

of the *compressed* matrix. To measure the practical impact of this theoretical result, we considered 500 iterations of the computation

$$y_i = M\,x_i, \quad z_i^t = y_i^t\,M, \quad x_{i+1} = \frac{z_i}{\|z_i\|_\infty} \tag{4}$$

where $\|z_i\|_\infty$ is the largest modulus of the components of z_i. The above computation consists of 500 alternated left and right matrix multiplications and mimics, e.g., the most costly operations of conjugate gradient method used for least square computations.

For the above iterative scheme we report in Table 2 the *average time per iteration* and the *peak memory usage*, as measured by the Unix tool `time`. In addition to the single-threaded algorithms, we tested versions using 4, 8, 12, and 16 threads. The first two columns in Table 2 report the peak memory usage and average iteration time for the single-threaded version of re_iv and re_ans, for which the input matrix is not partitioned and it is therefore grammar-compressed as a single unit. As expected for both algorithms the peak memory usage of the single-threaded version of re_iv and re_ans is somewhat larger than the compressed size reported in Table 1. Indeed, according to Theorems 3.4 and 3.10 in addition to the space for the input and output vectors our algorithms use as a working space an (uncompressed) array of $|\mathcal{R}|$ 8-byte doubles. However, the difference between peak memory usage and compressed matrix size is less than 7% of the uncompressed matrix size, with the only exception of *Higgs* ($\approx$ 9%). Unfortunately, the time per iteration of the single-threaded version is disappointing especially for the larger matrices. Hence, we have investigated the use of multiple threads by partitioning the matrix into a number of row-blocks equal to the number of threads as discussed in Section 4.1.

Figure 3 shows the increase of the peak memory usage (first row) and the decrease of the running time (second row) as the number of threads increases for re_ans and re_iv. We see that, with the exception of the most compressible inputs (i.e. *Covtype* and *Census*), with 16 threads the peak memory usage is always less than 1.5 times the peak memory usage of the single-threaded version (for the most compressible inputs the overheads of the computation dominate over the storage of the compressed matrix). Notice also that for *Higgs* the space usage of the multi-threaded versions of re_iv and re_ans is smaller than for the single-threaded version: the reason is that this file is better compressed when split into distinct blocks (this usually happens when the blocks have little structure in common). Comparing the plots at the top of Figure 3, we see that for re_iv the memory overhead of using multiple threads grows slower than for re_ans. Hence, although re_iv is a simpler and usually less effective compressor, it uses less space than re_ans when working with 16 threads as shown by the last two columns of Table 2.

As far as time efficiency is concerned, Figure 3 (bottom left) shows that for re_ans using 4 threads the speedup is close to 100% (time ratio is 1/4), when using 8 threads the speedup is still close to the optimal (i.e. 1/8) except for *Census* and *Susy*. As expected, a larger number of threads only helps re_ans with the largest inputs: for *Higgs*, *Airline78* and *Mnist2m* with 16 threads the speedup is still within 12.66 and 14.90. On the other hand, for *Covtype*, which is the smallest input, re_ans does not achieve any improvement by going from 8 to 16 threads. For re_iv the speedup follows a similar trend (Figure 3 bottom right). We notice that for 4 and 8 threads the speedup is smaller than for re_ans, but for 16 threads the speedup is larger than 11 for all inputs except, again, for the small *Covtype*.

Table 2 summarises the statistics for the iterative computation of Eq. (4) with csrv and our grammar compressors using 16 threads. The results show that even for a multi-threaded computation the overall space usage can still be a small fraction of the uncompressed size. Indeed, the peak memory usage of the grammar compressors is up to 3× smaller than for csrv (i.e. for *Census*) and for 5 inputs it is less than 20% of the original uncompressed size. As we will discuss in Section 5.4, such impressive compression rates come together with a reduced average time per iteration compared to the state-of-the-art tool CLA; in some cases we operate even faster than over the uncompressed (dense) matrix representation.

The combined analysis of the peak memory usage versus running time highlights some interesting points. Considering all the algorithms running with 16 threads we see that, not surprisingly, the simpler compressed representations usually lead to faster matrix multiplications. Among the grammar compressors, the fastest algorithm is re_32 in which the string C and the rule set $\mathcal{R}$ are represented by 32-bit integers. The more sophisticated encoders re_iv and re_ans achieve better compression but they are slower. This is in accordance with the theoretical results: according to Theorems 3.4 and 3.10 the cost of matrix-vector multiplication is $O(|C|+|\mathcal{R}|)$ time; re_iv and re_ans use compressed representations

Table 2: Peak memory usage and average time per iteration in seconds for the computation of 500 iterations of Eq. (4). The memory usage is expressed as the percentage of the size of the full uncompressed matrix.

	re_iv 1 thread		re_ans 1 thread		csrv 16 threads		re_32 16 threads		re_iv 16 threads		re_ans 16 threads	
matrix	peak mem	time	peak mem	time	peak mem	time	peak mem	time	peak mem	time	peak mem	time
Susy	76.15%	3.89	73.40%	4.88	80.66%	0.26	80.63%	0.27	77.45%	0.35	82.67%	0.45
Higgs	50.30%	8.28	47.12%	11.03	54.12%	0.36	52.04%	0.42	47.01%	0.62	44.90%	0.74
Airline78	17.16%	2.88	15.40%	3.94	41.57%	0.17	24.72%	0.15	19.21%	0.25	19.28%	0.31
Covtype	9.42%	0.05	10.16%	0.07	14.60%	0.01	13.09%	0.01	17.10%	0.01	17.29%	0.01
Census	4.37%	0.12	4.11%	0.19	23.88%	0.05	6.70%	0.01	6.14%	0.01	8.03%	0.02
Optical	39.83%	0.73	39.23%	1.08	51.70%	0.04	46.56%	0.04	45.00%	0.06	56.72%	0.09
Mnist2m	7.33%	7.09	6.85%	9.87	12.83%	0.20	11.31%	0.42	8.19%	0.60	8.30%	0.78
ImageNet	5.21%	4.56	5.21%	4.58	6.95%	0.38	6.95%	0.39	6.95%	0.41	6.95%	0.39

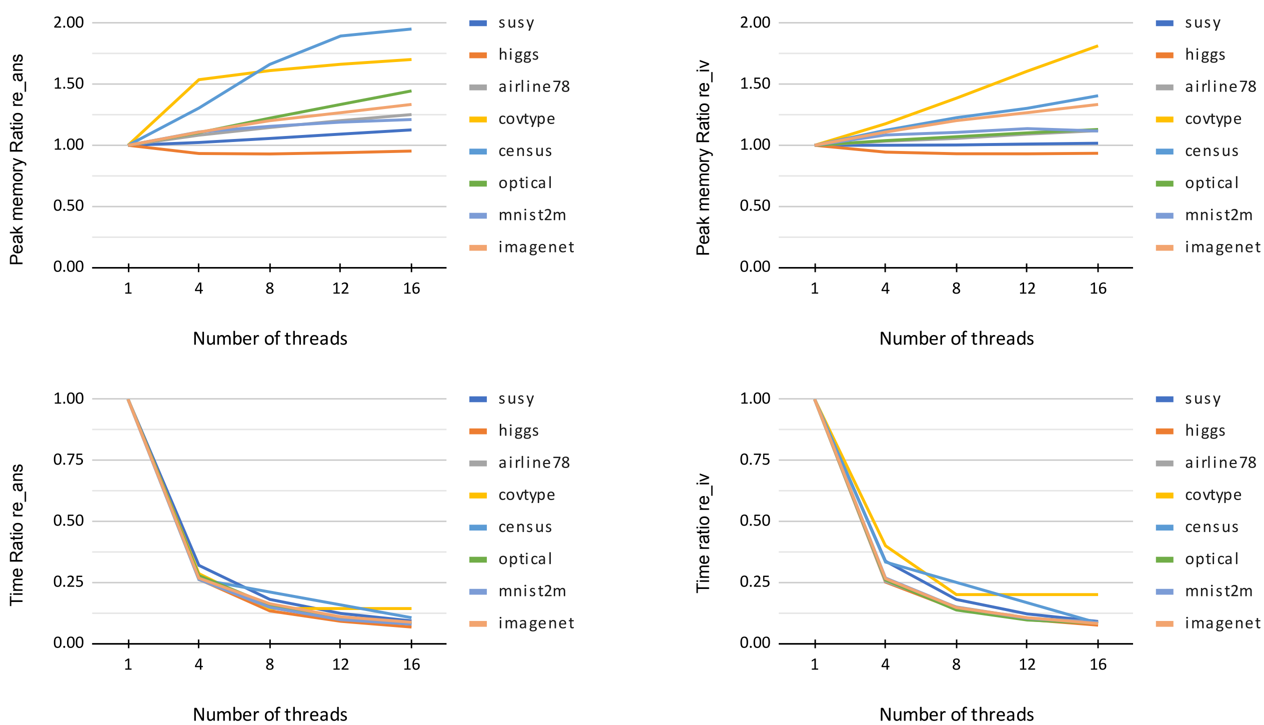

Figure 3: Peak memory usage (up) and running time (bottom) of the multi-threaded version of the matrix multiplication algorithm using the re_ans (left) and re_iv (right) compressors. The Y-axis reports the ratio between time and space of the multi-threaded version of re_ans or re_iv versus the single-threaded version of the same algorithm.

of C and $\mathcal{R}$: this reduces the peak memory usage but not the number of arithmetic operations. As for the csrv representation (see column 3 in Table 2), we notice that different input matrices can have very different behaviours. For *Airline78*, e.g., re_32 uses much less space than csrv but shows only a modest improvement in running time. For *Mnist2m*, re_32 shows a modest reduction in space but an increase in running time; the most sophisticated re_iv and re_ans tools achieve greater compression but they are significantly slower. Finally, for *Census* we have a four-times reduction in space for re_32 and re_iv with a five-fold improvement in running time. Since users want the fastest algorithm than can run in the available memory, we conclude that all compressors should be considered; indeed an interesting problem would be the design of a mechanism for selecting the best options given the user's constraints.

Table 3 reports the compression-time performance of our approaches using 16 threads. Comparing the running times for csrv and re_32 we see that computing the CSRV representation costs more than computing the grammar. In fact, the former takes $O(mn)$

Table 3: Compression times in seconds for the 16-thread version of our algorithms and CLA.

matrix	csrv	re_32	re_iv	re_ans	CLA
Susy	58.97	61.79	62.14	112.10	–
Higgs	136.99	148.34	149.21	223.73	39.86
Airline78	42.65	58.33	58.85	71.29	5.45
Covtype	1.55	1.77	1.83	2.23	8.52
Census	10.94	13.22	13.29	14.12	8.52
Optical	15.84	17.81	18.01	34.38	12.25
Mnist2m	82.95	104.75	105.68	147.10	118.97
Imagenet	63.66	84.87	85.44	104.44	177.39

time, while the latter takes time proportional to the number of nonzero elements. In addition, the computation of the CSRV representation involves the whole matrix and it is done by a single thread, while the grammar compression is done in parallel using a thread for each row block. Comparing the running times for re_32, re_iv and re_ans we see that, not surprisingly, computing the packed-array representation of C and $\mathcal{R}$ is relatively inexpensive, while compressing C with the `ans-fold` entropy coder may take more time (see *Susy* and *Higgs*). As a reference we also report compression times for CLA (discussed in Section 5.3) which is usually faster than our proposals. However, it is worth saying that compression is done *once* while the compressed matrix is later used many times, so construction speed was actually not a main goal of this paper.

Finally, we point out that there are avenues for improving our algorithms. For example, in our tests we used the same compressor for each row block of the input matrix: we could use different compressors to compress different blocks, or use the CSRV representation for the blocks which are hard to compress (a similar idea, applied to blocks of columns, is used within CLA). Another avenue for improvement is the reordering of the elements of the array S as discussed at the end of Section 3: some promising results in this direction are presented in the next section.

5 COLUMN REORDERING FOR GRAMMAR COMPRESSION

In this section we show how the reordering of the columns of the input matrix improves the performance of our grammar compressor. As we mentioned at the end of Section 3, reordering the columns is only one of the possible preprocessing operations that can be applied to the input matrix without affecting our multiplication algorithms. We start our investigation with this technique as it was already studied in the related area of table compression [5, 35]. Grammar compression for the CSRV representation replaces pairs of symbols appearing adjacently and in many rows with a single nonterminal. Hence, we aim at reordering matrix columns so that *correlated* columns appear adjacent to each other. To this end, we define the *column similarity* as the number of identical symbol pairs (cf. the formal definition in the next subsection). This similarity score estimates the compressible fraction of every column pair, hence modelling the compression performance of a tool like RePair when two columns are placed one adjacent to the other in the final ordering. This conservative, yet simple idea, achieves an effective performance as proved by our experiments. We use the column-similarity score within four novel column-reordering algorithms and measure their impact on the performance of our compressed matrix-vector multiplication algorithm.

5.1 The column-column similarity matrix

Given the input matrix $M \in \mathbf{R}^{n,m}$ we define the $m \times m$ column-column similarity matrix CSM as follows. For each pair of column indices i and j, with $1 \le i \ne j \le m$, we build the sequence of pairs

$$P_{ij} = \langle M[1][i], M[1][j] \rangle, \langle M[2][i], M[2][j] \rangle \dots \\ \dots \langle M[n][i], M[n][j] \rangle$$

and we define RP_{ij}^{NZ} as the number of repetitions of pairs of nonzero elements in the sequence P_{ij} (note we only consider pairs in which both elements are nonzeros). For example, for the matrix of Fig. 1 it holds $RP_{12}^{NZ} = 2$ because P_{12} contains only one nonzero pair, i.e. $\langle 1.2, 3.4 \rangle$, which has two repetitions; and $RP_{13}^{NZ} = 1$ because P_{24} contains two non-zero pairs, i.e. $\langle 1.2, 5.6 \rangle$ and $\langle 1.2, 2.3 \rangle$, but only one repetition of $\langle 1.2, 2.3 \rangle$.

So, we define the similarity between columns i and j as the ratio

$$\mathsf{CSM}[j][i] = \mathsf{CSM}[i][j] = \frac{RP_{ij}^{NZ}}{n}.$$

From the previous example we have $\mathsf{CSM}[1][2] = 2/6 = 0.\bar{3}$, and $\mathsf{CSM}[1][3] = 1/6 = 0.1\bar{6}$.

The computation of $\mathsf{CSM}[i][j]$ can be done in $O(n)$ expected time by inserting each pair in a hash table, thus taking $O(m^2 n)$ time over all column pairs. An alternative procedure taking $O(m^2 n \log n)$ time consists in collecting all pairs and sorting them in order to easily count duplicates. The sorting-based approach turned out to be very fast in practice and hence we chose it our experiments.

The storage of CSM takes $\Theta(m^2)$ words if we use a full-sized representation. We also experimented with two heuristics for reducing that space bound to $O(mk)$, where k is a user-defined sparsity parameter. The first heuristic consists of building a sparse CSM matrix, called *locally-pruned* column-column similarity matrix CSM_p, in which we maintain only the k greatest column-column similarity scores for each column. The second heuristic builds a globally-pruned column-column similarity matrix CSM_P by keeping the top-(mk) similarity scores among all the entries of CSM. The space complexity is still $O(mk)$, but now the pruning is performed *globally* over all entries of the original matrix.

5.2 Column-reordering approaches

Once we have computed the column-similarity matrix CSM either in its full or sparse version, we leverage it to find a column reordering that helps grammar compression. We investigated four different column-reordering algorithms working upon the weighted graph G whose adjacency matrix is either CSM (consisting of $\Theta(m^2)$ edges) or CSM_P (consisting of $\Theta(mk)$ edges). They are described below.

The **Lin-Kernighan heuristic (LKH)** is a heuristic for the *Travelling Salesman Problem* (TSP). Though the algorithm is approximate, the implementation in [18, 19] computes the best known solution for a series of large-scale instances with unknown optima.

We model column reordering as an instance of a (symmetric) TSP stated on the graph G above. Each of the m columns in the original matrix M corresponds to a different city in the TSP; the distance between pairs of cities (columns) is given by the corresponding entry in the matrices CSM or CSM_P (negated, since the TSP is a minimisation problem and we want to maximise total similarity). The TSP solution will specify an ordering of M's columns. We used the ANSI C implementation of LKH available at: http://webhotel4.ruc.dk/~keld/research/LKH/ (version 2.0.9).

The **PathCover** approach reduces column reordering to the problem of finding a set of maximum weighted paths in the above-mentioned graph G; it requires that these paths "cover" all of its nodes and they are disjoint. We introduce this approach since TSP is NP-hard, but we do not necessarily need to impose its strong constraint of forming a single Hamiltonian path. We may indeed concentrate our algorithmic effort upon the subset of compressible columns [32], leaving aside those columns that do not exhibit significant redundancies. PathCover returns a set of partial reorderings (induced by the found paths) that yield a full reordering if concatenated together. The approach is a reminiscence of the *single linkage* algorithm used in hierarchical clustering [25, Ch. 17]. We implement PathCover using a variant of the Kruskal's algorithm for Minimum Spanning Trees [9]. We scan G's edges by decreasing weights, and add edges to the solution only if they form disjoint paths. Though in Python, our code runs very fast in practice.

PathCover+ is a variant of PathCover in which the column-column similarity matrix is dynamically updated as follows. Let (u_{r-1}, u_r) be the heaviest edge selected by the PathCover algorithm, and assume that it extends a covering path to form $\mathcal{P} = (u_1, ..., u_{r-1}, u_r)$. Then, for each node v adjacent to some node $u_j \in \mathcal{P}$, we recompute the new weight $w(v, u_j)$ the minimum among the weights from v to any node in $\mathcal{P}$. Thus, the weighting corresponds to coalescing the path $\mathcal{P}$ into a macro-node and making the link from v to $\mathcal{P}$ as the minimum weighted edge from v to any node $u \in \mathcal{P}$. We implemented PathCover+ in Python following a procedure similar to Sybein's MST algorithm [27].

The **Maximum Weighted Matching (MWM)** approach determines a weighted matching $\mathcal{M}$ of the graph G. By definition, $\mathcal{M}$ is a subgraph of G such that no two edges share common vertices and the sum of the edge weights is maximum among all possible matchings in G. The best exact MWM algorithm exhibits $\Theta(m^3)$ time complexity [15]. For our column-reordering purpose, we generate a bipartite graph B_G with $2m$ nodes and $\binom{m}{2}$ edges weighted according to the column-column similarity entries. To clarify, for each column pair i, j, with $i < j$, we insert an edge in B_G that connects the i-th node to the j-th node and assign to it weight $\mathrm{CSM}[i][j]$ or $\mathrm{CSM}_P[i][j]$. Choosing that edge corresponds to assuming that the i-th column precedes the j-th column in the reordering. After the MWM is computed, we use this predecessor-successor relation to determine the final column reordering. Notice we cannot induce cycles, as we assumed that edges (i, j) are oriented, namely $i < j$. If the matching size $|\mathcal{M}|$ is lower than the number of columns m in M, then MWM does not induce a single column-reordering sequence, but rather a set of shorter disjoint column-reordering sequences: we thus concatenate these partial reordering sequences in an arbitrary order to get a full reordering. We implemented MWM in C++ using the Boost library (https://www.boost.org).

5.3 Experimenting with column reordering

We conducted a set of experiments using the matrices reported in Table 1 to analyse the time and space performance of the column-reordering approaches described above. After applying the column-reordering algorithm we compressed the reordered matrix using re_ans from Section 4. We report the results only for the methods LKH, PathCover, and MWM, since the PathCover+ method always resulted in a worse compression performance.

The three column-reordering algorithms exhibit quite different time performances. PathCover is faster than MWM, and their time performance is dominated by the construction of the column-similarity matrix. LKH is the most time-consuming one and its running time slightly varies with the setting of LKH heuristic (faster solutions correspond to worse results), but in any case LKH is orders of magnitude slower than the other approaches.

In terms of space performance, we found that the locally-pruned version of the CSM matrix usually outperforms the full matrix or the globally-pruned matrix. Table 4 reports the compression achieved by this approach on the whole (unpartitioned) matrices for the three reordering algorithms and for three different values of the sparsity parameter k. We see from Table 4 that for *Susy* the three reordering algorithms exhibit the same performance, and LKH slightly wins over *ImageNet*. PathCover is superior over three matrices, while MWM is the winner for the remaining three. LKH is often very close to the best compression but, given its larger computational cost, we conclude that it is not a competitive solution. Overall, reordering columns is advantageous up to 16.35% over *Covtype*, and up to 10.26% over *Airline78*, as indicated in column "gain", where we report the space reduction induced by column-reordering *with respect to* the version without reordering.

Next, to measure the effectiveness of the reordering techniques for the matrix multiplication operations we performed the following experiment. We partitioned each input matrix into 16 blocks of rows as described in Section 4.1. Then, we applied to each block of rows the best column-reordering according to Table 4 (in either case with sparsity parameter $k = 16$) followed by re_ans, and selected the column-reordering algorithms yielding the best compression (so each block can be subjected to a different permutation).[2] With such reordered-and-compressed matrix, we performed our benchmark computation (Eq. 4) and recorded the peak memory usage and the average time per iteration. We reiterated the same procedure for re_iv and reported the results in Table 5. Comparing these results to those in Table 2 we see that reordering helps to reduce the peak space and, to a lesser degree, the average running time too.

Although the benefits of reordering might appear small in absolute terms they can be, again, significant in *relative terms* with respect to the size of the compressed matrix. To see this, Figure 4 shows, for the two algorithms and for each input matrix, the percentage improvements in the peak memory usage, computed as $(p_o - p_r)/p_o$ where p_o and p_r are respectively the peak memory usage for the original and for the reordered matrix. We see that there is an interesting memory-usage reduction for half of the inputs: for *Airline78*, *Covtype*, *Census* and *ImageNet* compressed by re_iv and re_ans we observed a memory usage reduction between roughly

[2] As we observed at the end of Section 3.2, we do not need to store the column permutation because every pair in S stores the original column of each element.

Table 4: Compression achieved by our column-reordering algorithms, with the locally-pruned CSM matrix, followed by the algorithm re_ans. Compression ratios should be compared to those of re_ans without reordering from Table 1; these are shown in parentheses in the rightmost column where we also report the *relative* space reduction achieved by the best permutation (highlighted in red).

matrix		LKH	PathCover	MWM	gain
Susy	k=4	66.57%	66.57%	66.57%	0.09% (66.63%)
	k=8	66.57%	66.57%	66.57%	
	k=16	66.57%	66.57%	66.57%	
Higgs	k=4	38.03%	38.00%	37.99%	0.36% (38.05%)
	k=8	37.92%	38.00%	37.98%	
	k=16	38.02%	38.04%	**37.92%**	
Airl.	k=4	9.63%	9.21%	10.17%	10.26% (9.27%)
	k=8	8.65%	9.52%	**8.32%**	
	k=16	9.43%	8.34%	9.63%	
Covt.	k=4	3.74%	3.30%	4.19%	16.35% (3.87%)
	k=8	3.51%	**3.24%**	3.72%	
	k=16	3.25%	3.26%	3.72%	
Census	k=4	1.37%	1.39%	1.37%	3.40% (1.53%)
	k=8	1.33%	1.37%	1.41%	
	k=16	1.31%	**1.30%**	1.39%	
Optical	k=4	33.23%	**32.60%**	33.19%	4.99% (34.31%)
	k=8	32.68%	33.03%	33.26%	
	k=16	33.22%	32.89%	32.95%	
Mn.2m	k=4	5.29%	5.31%	5.32%	0.73% (5.33%)
	k=8	5.29%	5.31%	**5.29%**	
	k=16	5.30%	5.31%	5.30%	
Im.Net	k=4	3.84%	3.87%	3.90%	2.14% (3.86%)
	k=8	3.82%	3.84%	3.88%	
	k=16	**3.78%**	3.81%	3.86%	

5% and 15% of the original memory usage. The running times for these experiments are reported in Table 5. Remarkably, for *Airline78* such memory reduction translates to a 25% reduction in the average running time. Note also that sometimes reordering does not help: for *Mnist2m* reordering does not change the peak memory usage but instead induces a small (5%) increase in the running time for both algorithms; and for *Susy*, the reordering slightly increases the peak memory, with no significant changes in the running time.

5.4 Matrix-vector multiplication efficiency

In this section we are interested in evaluating the peak memory usage versus the speed of matrix-vector multiplication over five approaches: two based on our compressors re_iv and re_ans applied over the block-wise optimally reordered matrix (described in the previous section), CLA [12, 13], and two baselines storing in RAM the `gzip`-compressed matrix or the uncompressed matrix. In our experiments we set CLA to use all the available threads (80 in our test machine) while the other approaches used 16 threads partitioning the input matrix into 16 row-blocks as in Section 4.1. Results are reported in Table 5. Columns size and PM are respectively the

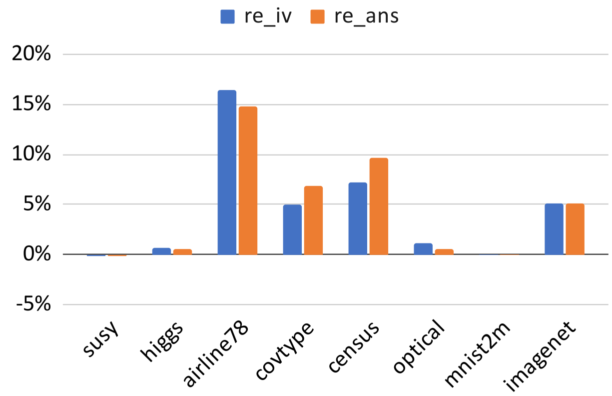

Figure 4: Percentage (relative) improvements in terms of the peak memory usage for the reordered matrices as resulting from the data reported in Table 5 for re_iv and re_ans.

size of the compressed matrix and the peak memory usage during the multiplication algorithm expressed as a percentage with respect to the size of the uncompressed matrix (we omitted the size for the "uncompressed" algorithm since it was obviously 100%). Column time is the time in seconds for a single iteration of Eq. (4) averaged over 500 iterations. PM and time were measured using the Unix tool `time` except for CLA as discussed below.

The comparison with CLA faces some technical hurdles: we implemented our tools in small self-contained C/C++ programs, while CLA is available inside Apache SystemDS [34], a complete ML system written in Java and designed to run possibly on top of Apache Spark. SystemDS's algorithms are expressed in a high-level language with an R-like syntax: such scripts are parsed and analysed before the actual computation starts. In addition, SystemDS does not store the compressed matrix on disk: matrices are compressed from scratch at every execution; since the compression algorithm has a randomised component the compressed representation can change from one execution to the next one. For these technical reasons, the time for CLA includes compression time, the compressed matrix size was derived from SystemDS logs, and the peak memory usage for the matrix-vector multiplication phase alone has been computed "forcing" the Java garbage collector using a procedure suggested by CLA's authors. Note that for the matrix *Susy*, CLA was unable to complete the computation due to a Java runtime exception.

As for compression, CLA is less effective than re_ans with the only exception of *Higgs*. Compared to re_iv, CLA is clearly superior for *Higgs*, marginally superior for *Covtype* and *Mnist2m*, and less effective for all the other inputs. The PM of CLA exceeded in some cases the dimension of the uncompressed representation and it was always larger than our approaches by a factor from 3.14 (*Higgs*) to 19.12 (*Census*). As for running time, CLA is always at least two times slower than re_ans, and at least three times slower than re_iv (but we should remember that CLA time includes construction).

The `gzip`-based approach decompresses each row block at each iteration and multiplies it for the current vector. The whole computation is done completely in RAM using a thread for each row block;

Table 5: Performance comparison considering compressed space, peak memory usage (PM), and average running time in seconds for matrix-vector multiplication; see text for details. Sizes and PMs are expressed as percentages.

	re_iv 16 threads			re_ans 16 threads			CLA multithread			gzip 16 threads			uncompressed 16 threads	
matrix	size (%)	PM (%)	time (s)	size (%)	PM (%)	time (s)	size (%)	PM (%)	time (s)	size (%)	PM (%)	time (s)	PM (%)	time (s)
Susy	68.99	77.53	0.35	65.99	82.77	0.45	76.14	—	—	53.27	63.09	2.22	106.14	0.17
Higgs	41.63	46.68	0.58	37.44	44.63	0.71	32.74	146.68	2.09	48.38	54.56	5.48	103.74	0.51
Airl.	9.35	16.06	0.17	8.13	16.43	0.23	12.34	120.27	1.17	13.27	17.53	6.27	103.57	0.75
Covt.	4.78	16.25	0.01	4.17	16.11	0.01	4.55	70.15	0.05	6.25	10.26	0.41	103.51	0.03
Census	2.00	5.70	0.01	1.55	7.25	0.02	3.77	108.96	0.16	5.54	7.92	1.89	101.77	0.12
Optical	36.05	44.50	0.06	34.93	56.39	0.09	40.44	176.90	0.20	53.55	57.26	1.00	101.47	0.04
Mn.2m	6.24	8.19	0.64	5.88	8.30	0.82	6.22	47.09	1.98	6.46	6.76	24.96	100.16	0.57
Im.Net	4.70	6.59	0.48	4.28	6.59	0.48	6.67	56.80	0.97	5.52	5.89	10.91	100.16	0.46

anyway, this was by far the slowest algorithm though having the best PM together with our algorithms (no clear winner here), which however are much faster (more than 40× for the most compressible matrices). Finally, the approach storing the uncompressed matrix in RAM has naturally a PM slightly larger than 100% and it is usually the fastest algorithm, except for some highly-compressible files (i.e. *Covtype*, *Census* and *Airline78*) where our approaches are faster.

Summing up, from the above comparisons we can draw some important conclusions about our grammar compressors: (1) they are able to save disk space and PM, thus providing experimental evidence to the theoretical space bounds in terms of the k-th order statistical entropy; (2) they are the fastest among the compressed approaches, and for the most compressible matrices even faster than the uncompressed algorithm, thus providing experimental evidence to the theoretical results ensuring that the number of operations is bounded by the size of the compressed matrix.

6 CONCLUSIONS AND FUTURE WORK

We have presented a grammar-based lossless compression scheme for real-valued matrices that guarantees the size of the compressed matrix is proportional to the k-th order statistical entropy of the Compressed Sparse Row/Value representation. We have shown how to perform left and right matrix-vector multiplications in time and space linear with the size of the compressed matrix representation.

These remarkable properties of our approach open the related problem of reordering the matrix elements for maximising compression. This requires discovering and exploiting the hidden dependencies between elements in ML matrices. As a first step in this direction we have introduced and tested four column-reordering algorithms based upon a new column-similarity score, which takes into account the subsequent grammar-compression stage.

As a future work, we plan to investigate how much row permutation and co-clustering techniques [3, 10, 17] can contribute to achieving even better compression ratios. Moreover, it seems possible to extend the proposed grammar-compressed techniques to deal with semiring-annotated data, thereby computing binary/unary joins efficiently. We can indeed operate upon logical matrices and simulate binary joins by replacing "+" with *OR* and "∗" with *AND*. It would be of interest also to adapt and test our matrix-compression scheme in the context of columnar DBs, which feature multiple data types, such as strings, integers, categorical data, etc. Finally, web and social graphs offer another relevant opportunity for the application of our new compression schemes.

ACKNOWLEDGMENTS

P.F., G.M., M.S., and F.T. have been supported in part by the Italian MUR PRIN project "Multicriteria data structures and algorithms: from compressed to learned indexes, and beyond" (Prot. 2017WR7SHH), P.F. and F.T. also by the EU H2020 projects "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics" (grant #871042) and "HumanE AI Network" (grant #952026); T.G. by NSERC (grant RGPIN-07185-2020); D.K. by JSPS KAKENHI (grants JP21K17701 and JP21H05847); and G.N. by ANID – Millennium Science Initiative Program – Code ICN17_002 (Chile).

REFERENCES

[1] Alberto Apostolico, Fabio Cunial, and Vineith Kaul. 2008. Table Compression by Record Intersections. In *2008 Data Compression Conference (DCC 2008), 25-27 March 2008*. IEEE Computer Society, Snowbird, UT, USA, 13–22. https://doi.org/10.1109/DCC.2008.105

[2] ASA, Statistical Computing & Graphics Sections 2009. Data Expo 2009 – Airline on-time performance. https://community.amstat.org/jointscsg-section/dataexpo/dataexpo2009. [Online; accessed 21-Sep-2021].

[3] Arindam Banerjee, Inderjit S. Dhillon, Joydeep Ghosh, Srujana Merugu, and Dharmendra S. Modha. 2007. A Generalized Maximum Entropy Approach to Bregman Co-clustering and Matrix Approximation. *J. Mach. Learn. Res.* 8 (2007), 1919–1986.

[4] Leon Bottou. 2007. The infinite MNIST dataset. https://leon.bottou.org/projects/infimnist. https://leon.bottou.org/projects/infimnist [Online; accessed 21-Sep-2021].

[5] Alan L. Buchsbaum, Glenn S. Fowler, and Raffaele Giancarlo. 2003. Improving Table Compression with Combinatorial Optimization. *J. ACM* 50 (2003), 825–851.

[6] Deepayan Chakrabarti, Spiros Papadimitriou, Dharmendra S. Modha, and Christos Faloutsos. 2004. Fully automatic cross-associations. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 22-25, 2004*. ACM, Seattle, Washington, USA, 79–88.

[7] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. 2005. The smallest grammar problem. *IEEE Trans. Inf. Theory* 51, 7 (2005), 2554–2576.

[8] Radha Chitta, Rong Jin, Timothy C. Havens, and Anil K. Jain. 2011. Approximate Kernel K-Means: Solution to Large Scale Kernel Clustering. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Diego, California, USA) *(KDD '11)*. Association for Computing Machinery, New York, NY, USA, 895–903. https://doi.org/10.1145/2020408.2020558

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press, One Broadway, 12th Floor, Cambridge, MA 02142.

[10] Inderjit S. Dhillon. 2001. Co-clustering documents and words using bipartite spectral graph partitioning. In *KDD*. ACM, 269–274.

[11] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. https://archive.ics.uci.edu/ml. http://archive.ics.uci.edu/ml [Online; accessed 21-Sep-2021].

[12] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2018. Compressed linear algebra for large-scale machine learning. *VLDB J.* 27, 5 (2018), 719–744.

[13] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2019. Compressed linear algebra for declarative large-scale machine learning. *Commun. ACM* 62, 5 (2019), 83–91. https://doi.org/10.1145/3318221

[14] Alexandre Francisco, Travis Gagie, Susana Ladra, and Gonzalo Navarro. 2018. Exploiting computation-friendly graph compression methods for adjacency-matrix multiplication. In *2018 Data Compression Conference*. IEEE Computer Society Press, 307–314.

[15] Harold N. Gabow. 1976. An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs. *J. ACM* 23, 2 (April 1976), 221–234. https://doi.org/10.1145/321941.321942

[16] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014) (LNCS)*, Vol. 8504. Springer, 326–337.

[17] Bo Han and Bolang Li. 2016. Lossless Compression of Data Tables in Mobile Devices by Using Co-clustering. *Int. J. Comput. Commun. Control* 11, 6 (2016), 776–788.

[18] Keld Helsgaun. 2000. An effective implementation of the Lin—Kernighan traveling salesman heuristic. *European Journal of Operational Research* 126, 1 (2000), 106–130. https://doi.org/10.1016/S0377-2217(99)00284-2

[19] Keld Helsgaun. 2009. General k-opt submoves for the Lin—Kernighan TSP heuristic. *Mathematical Programming Computation* 1, 2 (01 Oct 2009), 119–163. https://doi.org/10.1007/s12532-009-0004-6

[20] David S. Johnson, Shankar Krishnan, Jatin Chhugani, Subodh Kumar, and Suresh Venkatasubramanian. 2004. Compressing Large Boolean Matrices using Reordering Techniques.. In *VLDB*. Morgan Kaufmann, 13–23.

[21] John C. Kieffer and En-Hui Yang. 2000. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory* 46, 3 (2000), 737–754.

[22] Kornilios Kourtis, Georgios I. Goumas, and Nectarios Koziris. 2008. Optimizing sparse matrix-vector multiplication using index and value compression. In *Conf. Computing Frontiers*. ACM, 87–96.

[23] Jesper Larsson and Alistair Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.

[24] Markus Lohrey. 2012. Algorithmics on SLP-compressed strings: A survey. *Groups Complex. Cryptol.* 4, 2 (2012), 241–299.

[25] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press.

[26] Giovanni Manzini. 2021. A Collection of Some Machine Learning Matrices. https://www.kaggle.com/giovannimanzini/some-machine-learning-matrices. Version 3.

[27] Kurt Mehlhorn and Peter Sanders. 2008. *Algorithms and Data Structures: The Basic Toolbox*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter 11, 217–232. https://doi.org/10.1007/978-3-540-77978-0_11

[28] Alistair Moffat and Matthias Petri. 2020. Large-Alphabet Semi-Static Entropy Coding Via Asymmetric Numeral Systems. *ACM Trans. Inf. Syst.* 38, 4 (2020), 33:1–33:33.

[29] Gonzalo Navarro. 2021. Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Comput. Surv.* 54, 2 (2021), 29:1–29:31.

[30] Carlos Ochoa and Gonzalo Navarro. 2019. RePair and All Irreducible Grammars are Upper Bounded by High-Order Empirical Entropy. *IEEE Trans. Inf. Theory* 65, 5 (2019), 3160–3164.

[31] Yousef Saad. 2003. *Iterative methods for sparse linear systems.* SIAM.

[32] Jia Shi. 2020. Column Partition and Permutation for Run Length Encoding in Columnar Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2873–2874. https://doi.org/10.1145/3318464.3384413

[33] James A. Storer and Thomas G. Szymanski. 1982. Data compression via textual substitution. *J. ACM* 29, 4 (1982), 928–951.

[34] The Apache Software Foundation 2021. Apache SystemDS. https://systemds.apache.org/. [Online; accessed 14-Dec-2021].

[35] Binh Dao Vo and Kiem-Phong Vo. 2007. Compressing table data with column dependency. *Theoretical Computer Science* 387, 3 (2007), 273–283.