Improving Matrix-vector Multiplication via Lossless Grammar-Compressed Matrices

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

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

Dominik Köppl  
M&D Data Science Center, Tokyo Medical and Dental University  
Chiyoda City, Tōkyō-to, Japan  
koeppl.dsc@tmd.ac.jp

Gonzalo Navarro  
Millennium Institute for Foundational Research on Data  
Department of Computer Science,  
University of Chile  
gnavarro@dcc.uchile.cl

Manuel Striani  
Department of Sciences and Technological Innovation,  
University of Piemonte Orientale  
Alessandria, Italy  
amuel.striani@uniupo.it

Francesco Tosoni  
Department of Computer Science,  
University of Pisa  
Pisa, Italy  
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 xc 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 xc 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:

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights are licensed to the VLDB Endowment.

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 \textit{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 getting 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 yield 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 get 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 (CSR/V) 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 we just mentioned, achieving simultaneously a saving in time and space is not new [12–14, 22], but to our knowledge our algorithmic approach is the first one 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 could be also used not only as stand-alone compression tools for matrices but also as a new powerful compression option within the CLA framework, or a similar system, in place of its simpler compressors.

Technically speaking, our starting point is the CSR representation of a matrix, which is a simple modification to the well-known compressed sparse row (CSR) representation [31]. The CSR 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 coming from real ML problems. In terms of attained compression ratios, the experiments show that 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 multiplication operations 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 multi-threaded 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; at a later stage, such groups get compressed independently 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 above papers 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 to work with our lossless grammar-based compressors for matrices. Then, we describe four new column-reordering algorithms that hinge on this score and, in order to boost compression, we apply them to blocks of rows 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 get a further reduction of up to 16% of the peak memory usage during matrix-vector multiplication.

As a final contribution of this paper, we compare our matrix compressor against the compressor used by the Compressed Linear Algebra (CLA) system, which constitutes the state-of-the-art in this setting [12, 13]. Experiments show that, in terms of compression, 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 the matrix-vector multiplication, being a factor between 3.14 and 19.12. In terms of running time, CLA is always at least twice 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 are indeed able to 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 that 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 that 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 $M \in \mathbb{R}^{n \times m}$ with $n$ rows and $m$ columns, the compressed sparse row (CSR) representation [31] is a classical 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 $\text{nz}$ listing the non-zero elements row-by-row; 2) a length-$t$ array $\text{idx}$ storing for each element in $\text{nz}$ 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 $\text{nz}$ 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 $\text{nz}$ not the actual non-zero values but their indices in $V$. If there are, say, fewer than $2^{15}$ distinct non-zero elements, then each entry in $\text{nz}$ only takes 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 (CSRVD), by making two minor modifications to the above scheme. Firstly, we combine the two length-$t$ arrays $\text{nz}$ and $\text{idx}$ in a single vector of pairs $S$, such that for $i = 1, \ldots, t$, entry $S[i]$ contains the pair of integers $(\text{nz}[i], \text{idx}[i])$. Secondly, instead of storing a separate array first we include its information in $S$ by storing a special symbol $\text{S}$ immediately after the last non-zero entry of each row. As a result, the array $S$ now has got length $t + n$ and 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 $(\ell, i, j)$, where $\ell$ 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 $\text{S}$. During the scanning, for each nonzero $M[i][j]$ we need to retrieve the index $\ell$ 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 CSRVD representation of a matrix $M \in \mathbb{R}^{n \times m}$ 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 CSRVD representation of matrix $M$ and a vector $x[1..m]$, it is straightforward to compute the matrix-vector multiplication $y = Mx$ 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 $(\ell, i)$ we add the value $V[\ell] \cdot x[j]$ to the entry $y[j]$. The occurrences of the symbol $\text{S}$ 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 $(\ell, i)$ we add the value $y[j] \cdot V[\ell]$ 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 CSRVD representation outlined above.

3 GRAMMAR-COMPRESSED MATRICES

We show how to compress the CSRVD 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 $\Sigma$ is a context-free grammar that generates only $T$ [7]. For the sake of simplicity, we assume that the grammar is a so-called straight-line program [24] (SLP), that is, it consists of a set of rules of the form $L_i \rightarrow R_i R_i$, where $L_i$ is a nonterminal and each of $R_i$ and $R_{i*}$ can be either a terminal (i.e., an element of the base alphabet $\Sigma$), or a nonterminal. The grammar generates only $T$, implying that each nonterminal appears as the left-hand side of a single rule, so that one can 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 we obtain by repeatedly applying the substitution rules of the SLP grammar until we are left with a string over $\Sigma$. Thus one can leverage a SLP to represent $T$ as a succinct sequence $C$ of nonterminals; when needed, $T$ can be obtained from $C$ by evaluating the expansion of its nonterminals.

The output of the grammar compressor is a set of rules and a special nonterminal whose expansion generates only the input string $T$. If the grammar has $q$ rules, and therefore $q$ nonterminals $N_1, \ldots, N_q$, 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 in bounded by $|T|H_k(T) + o(|T|H_k(T))$ bits for any $k \in o(\log \sigma |T|)$,
Figure 1: A matrix and its CSRV representation. In the array \( S \) the symbol \( (\ell, j) \) 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 \( (3, 3) \). Only the same values in the same column are represented by the same pair \((i, j)\).

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

\[
V = [1.2, \ 1.7, \ 2.3, \ 3.4, \ 4.5, \ 5.6] \\
S = (1.1, \ (4.2), \ (6.3), \ (3.5), \ (3.1), \ (3.3), \ (5.4), \ (2.5), \ (1.1), \ (4.2), \ (3.3), \ (5.4), \ (4.1), \ (6.3), \ (3.5), \ (3.1), \ (3.3), \ (5.4), \ (4.5), \ (4.5), \ (4.5))
\]

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

where \( \sigma \) 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 very effective also 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 \((i, j)\). As a result, the output of the grammar compressor applied to \( S \) consists of a set of rules \( \mathcal{R} \) and a string

\[
C = N_{i_1}$N_{i_2}$\cdots N_{i_n}$
\]

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 \((S, V)\) 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 \((\ell, j) \in S\) we define

\[
eval_x((\ell, j)) = V[\ell] \cdot x[j];
\]

(recall that the pair \((\ell, j)\) represents the value \( V[\ell] \) stored in column \( j \) of matrix \( M \)). Similarly, for a nonterminal \( N_i \) whose expansion is \((\ell_{1, i_1}) \cdot (\ell_{2, i_2}) \cdots (\ell_{h, i_h}) \) we define

\[
eval_x(N_i) = \sum_{k=1}^{h} \eval_x((\ell_{k, i_k})) = \sum_{k=1}^{h} V[\ell_k] \cdot x[j_k]. \tag{2}
\]

From the above definition we immediately get

**Lemma 3.2.** If the grammar contains the rule \( N_i \to AB \), then \( \eval_x(N_i) = \eval_x(A) + \eval_x(B) \). \hfill \( \square \)

**Lemma 3.3.** Given the representation \((C, \mathcal{R}, V)\) of a matrix \( M \in \mathbb{R}^{n \times m} \) with \( C = N_{i_1}$N_{i_2}$\cdots N_{i_n}$, if \( y = Mx \) then it holds that \( y[r] = \eval_x(N_{i_r}) \) 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 \((\ell_{1, i_1}) \cdot (\ell_{2, i_2}) \cdots (\ell_{h, i_h}) \) representing all the non-zero elements of row \( r \) where, for \( k = 1, \ldots, h \), \( \ell_k \) denotes the position in \( V \) containing the value \( M[r][j_k] \). Thus

\[
y[r] = \sum_{k=1}^{h} M[r][j_k] \cdot x[j_k] = \sum_{k=1}^{h} V[\ell_k] \cdot x[j_k] = \eval_x(N_{i_r}). \tag{2}
\]

**Theorem 3.4.** Given the grammar-compressed CSRV representation \((C, \mathcal{R}, V)\) of a matrix \( M \in \mathbb{R}^{n \times m} \) and a vector \( x \in \mathbb{R}^{m} \), 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 \( W[i] = \eval_x(N_i) \). 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 \( W[i] = \eval_x(N_i) \) is the sum of two terms that can be either of the form \( \eval_x((h, k)) \) or \( \eval_x(N_i) \) with \( j < i \). In the former case \( \eval_x((h, k)) = V[h] \cdot x[k] \); in the latter case \( \eval_x(N_i) = W[j] \) for some already-computed entry, since \( j < i \). One may indeed observe that the pairs \( N_i \)’s are ranked by the time when they get computed. After filling \( W \), we use Lemma 3.3 to get 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 \((i,j) \in S\) we define \(\text{rows}((i,j))\) as the set of rows whose CSRV representation contains \((i,j)\). Note that \(k \in \text{rows}((i,j))\) if, and only if, the expansion of the nonterminal \(N_k \in C\) contains the pair \((i,j)\) or, equivalently, \(M[k][j] = V[i]\).

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

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

\[
\text{sum}_y((i,j)) = \sum_{k \in \text{rows}((i,j))} y[k]
\]

Lemma 3.7. Given the CSRV representation \((S,V)\) of matrix \(M \in \mathbb{R}^{n \times m}\), let \(S'\) be the set of distinct symbols in \(S\) (i.e., without duplicates). If \(x' = y'M\) then, for \(j = 1, \ldots, m\), it holds

\[
x[j] = \sum_{(i,j) \in S'} V[i] \cdot \text{sum}_y((i,j))
\]

(note the summation involves only pairs in \(S'\) with second component \(j\).

Proof. Since

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

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

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

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

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

Lemma 3.9. For any symbol \(\alpha\) (terminal or nonterminal), let \(R_\alpha\) denote the set of nonterminals \(N_j\)'s such that their defining rule \(N_j \rightarrow AB\) contains \(\alpha\) in their right-hand side (i.e., \(A = \alpha\) or \(B = \alpha\)), and let \(I_\alpha\) denote the set of row indices \(\ell\) such that \(N_i \rightarrow \alpha\) (hence \(\ell \in I_\alpha\) iff the expansion of \(\alpha\) coincides with the \(\ell\)-th row). Then,

\[
\text{sum}_y(\alpha) = \sum_{N_j \in R_\alpha} \text{sum}_y(N_j) + \sum_{\ell \in I_\alpha} y[\ell]. \quad (3)
\]

Proof. Since each occurrence of \(\alpha\) is either the right-hand side of a single rule, or coincides with some \(N_i\), we have

\[
\text{rows}(\alpha) = \left( \bigcup_{N_j \in R_\alpha} \text{rows}(N_j) \right) \bigcup I_\alpha
\]

and the lemma follows by induction on the number of steps in the derivation of \(\alpha\).

In view of Lemma 3.7, to compute \(x' = y'M\), we need to compute \(V[1] \cdot \text{sum}_y((i,j))\) for all \((i,j) \in S'\). To this end we first compute \(\text{sum}_y\) for nonterminals and then we use Lemma 3.9 to derive the values \(\text{sum}_y((i,j))\).

In our implementation we introduce an auxiliary array \(W[1,q]\), where \(q = |R|\), such that at the end of the computation \(W[1] \) contains \(\text{sum}_y(N_1)\). More in detail, we initially set \(x[1,m] = 0\) to zero, and we set \(W[1, q] = 0\) to zero as well, except for the entries \(W[i]\) that we initialise to \(y[i]\) for every nonterminal \(N_i\) 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 \(N_j \rightarrow AB\) 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 \((h,k)\) we increase \(x[k]\) by \((V[h] \cdot W[j])\).

The crucial observation is that when we reach the rule \(N_j \rightarrow AB\) we have already computed in \(W[j]\) the correct value \(\text{sum}_y(N_j)\) 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 \(\text{sum}_y(N_j)\) is added to \(\text{sum}_y(A)\) and \(\text{sum}_y(B)\), affecting their corresponding values in \(W\) if they are nonterminal, or being accumulated in the proper entry of \(y\) if they are terminals.

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

We point out that we do not require that in the array \(S\), which is compressed to \(C\) and \(R\), the pairs relative to the same row are ordered according to column index, as we arranged them in Figure 1. For helping 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 improvement in compression obtained by reordering the columns of \(M\) globally: namely, reordering the elements in each row according to 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 get a family of grammar-compression algorithms.

Given a matrix \(M \in \mathbb{R}^{n \times m}\) 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 $S$ is encoded by the integer 0, while the pair $(i,j)$ is encoded by the integer $1 + im + j$ (recall that $0 \leq j < m$ 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 csvr and 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 $A B$, replacing all their occurrences by a new nonterminal $N_i$, and appending the rule $N_i \rightarrow A B$ to the current rule set. We modified RePair so that it never builds a rule involving the symbol $S$, 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 $N_0 S N_1 S \cdots N_n S$ discussed in the previous section; instead $C$ is usually longer and may even include terminals $(i,j)$. 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 pair of symbols. In its naive representation, RePair outputs $|C| + 2|\mathcal{R}|$ 32-bit integers overall.\footnote{Notice that for a rule $N_i \rightarrow A B$, we have to encode only $A$ and $B$ because the nonterminals $N_i$ get increasing ids.} However, this is quite a wasteful representation: if the largest nonterminal is represented by the integer $N_{\text{max}}$, we can represent $C$ and $\mathcal{R}$ using packed arrays with entries of $w = 1 + \lceil \log_2 N_{\text{max}} \rceil$ bits. What is 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. Therefore we have experimented 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 $1 + \lceil \log_2 N_{\text{max}} \rceil$ bits as described 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 gets partitioned into blocks of rows. Given a parameter $b > 1$, a $r \times c$ matrix $M$ is partitioned into $b$ blocks of size $\lfloor r/b \rfloor \times c$ (except for the last block which might get 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 $x' = y' M$ are independent as well; 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 $S_j$ contains the encoding of the non-zero elements of the $i$-th row block. We thereby grammar-compress each subvector $S_j$ using RePair; the resulting string $C_j$ and rule set $\mathcal{R}_j$ 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 $\times$ cols $\times$ 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 csvr representation, whilst the last three columns report the sizes of the three variants of our RePair compressor described above. All sizes are given as percentage of the ratio between the size of the compressed and the uncompressed matrix representations (rows $\times$ cols $\times$ 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 csvr representation, that takes advantage of repeated values, is already obtaining some compression; in particular for Optical, which has fewer distinct nonzeros, csvr 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 csvr and re\_32 output sizes is of interest to get some indication of the effectiveness of grammar
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 nonzero reports the percentage of non-zero elements over the total, while column #nonzero reports the number of distinct non-zero values.

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

As expected for both algorithms the peak memory usage of the single-threaded version of re_32 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 $|R|$ 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 blocks of consecutive rows as mentioned above.

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 be still a small fraction of the uncompressed size. Indeed, the peak memory usage of the grammar compressors...
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.

<table>
<thead>
<tr>
<th>matrix</th>
<th>re_iv 1 thread</th>
<th>re_ans 1 thread</th>
<th>csrv 16 threads</th>
<th>re_32 16 threads</th>
<th>re_iv 16 threads</th>
<th>re_ans 16 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>peak mem</td>
<td>time</td>
<td>peak mem</td>
<td>time</td>
<td>peak mem</td>
<td>time</td>
</tr>
<tr>
<td>Susy</td>
<td>76.15%</td>
<td>3.89</td>
<td>73.40%</td>
<td>4.88</td>
<td>80.66%</td>
<td>0.26</td>
</tr>
<tr>
<td>Higgs</td>
<td>50.30%</td>
<td>8.28</td>
<td>47.12%</td>
<td>11.03</td>
<td>54.12%</td>
<td>0.36</td>
</tr>
<tr>
<td>Airline78</td>
<td>17.16%</td>
<td>2.88</td>
<td>15.40%</td>
<td>3.94</td>
<td>41.57%</td>
<td>0.17</td>
</tr>
<tr>
<td>Covtype</td>
<td>9.42%</td>
<td>0.05</td>
<td>10.16%</td>
<td>0.07</td>
<td>14.60%</td>
<td>0.01</td>
</tr>
<tr>
<td>Census</td>
<td>4.37%</td>
<td>0.12</td>
<td>4.11%</td>
<td>0.19</td>
<td>23.88%</td>
<td>0.05</td>
</tr>
<tr>
<td>Optical</td>
<td>39.83%</td>
<td>0.73</td>
<td>39.23%</td>
<td>1.08</td>
<td>51.70%</td>
<td>0.04</td>
</tr>
<tr>
<td>Mnist2m</td>
<td>7.33%</td>
<td>7.09</td>
<td>6.85%</td>
<td>9.87</td>
<td>12.83%</td>
<td>0.20</td>
</tr>
<tr>
<td>ImageNet</td>
<td>5.21%</td>
<td>4.56</td>
<td>5.21%</td>
<td>4.58</td>
<td>6.95%</td>
<td>0.38</td>
</tr>
</tbody>
</table>

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.

is up to $3 \times$ 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 $R$ are represented with 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 $O(|C| + |R|)$ time; re_iv and re_ans use compressed representations of $C$ and $R$: 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 get more 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 the computation of the CSRV representation is more costly than the computation of the grammar. The explanation is that the former takes \(O(mn)\) time, while the latter takes time proportional to the number of non-zero 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, non surprisingly, computing the packed-array representation of \(C\) and \(R\) 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 works by replacing pairs of symbols appearing adjacent and in many rows with a single nonterminal. Therefore, we aim at reordering matrix columns so that correlated columns appear adjacent to each other. To this end, we define the similarity between two columns by counting the identical pairs they form (cf. the formal definition in the next subsection). In a sense, this similarity score estimates the compressible fraction of every column pair. This models the compression performance of a tool like RePair when the 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. After defining a notion of similarity between pairs of columns, we use it with 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 \(M \in \mathbb{R}^{n \times m}\) we define the \(m \times m\) column-column similarity matrix \(\text{CSM}\) as follows. For each pair of column indices \(i\) and \(j\), with \(1 \leq i \neq j \leq m\), we build the sequence of pairs

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

and we define \(RP_{\text{NZ}}^{ij}\) as the number of repetitions of pairs of non-zero 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 \(RP_{\text{NZ}}^{12} = 2\) because \(P_{12}\) contains only one non-zero pair, i.e. \((1.2, 3.4)\), which has two repetitions; and \(RP_{\text{NZ}}^{13} = 1\) because \(P_{24}\) contains two non-zero pairs, i.e. \((1.2, 5.6)\) and \((1.2, 2.3)\), but only one repetition of \((1.2, 2.3)\).

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

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

From the previous example we have \(\text{CSM}[1][2] = 2/6 = 0.3\), and \(\text{CSM}[1][3] = 1/6 = 0.16\).

The computation of \(\text{CSM}[j][i]\) 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 was the method of choice for our experiments.

The storage of \(\text{CSM}\) takes \(\Theta(m^2)\) words if we use a full-sized representation. We also experimented with two heuristics as for reducing that space bound to \(O(mk)\), where \(k\) is a user-defined sparsity parameter. The first heuristic consists of building a sparse
Covtype16.35% over Airline78, and up to 10.26% over Covtye, 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 in 16 blocks of rows
Table 4: Compression achieved by our column-reordering algorithms, with the locally-pruned CSM matrix, followed by the algorithm re_ans. Compression ratios can be compared against those in Table 1, which are also reported in the column “n.r.” (no reordering) to facilitate reading; red values represent the best results. The last column reports the relative space reduction compared to the version without reordering.

<table>
<thead>
<tr>
<th>matrix</th>
<th>n.r.</th>
<th>LKH</th>
<th>PathCover</th>
<th>MWM</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susy k=4</td>
<td></td>
<td>38.03%</td>
<td>38.00%</td>
<td>37.99%</td>
<td>0.36%</td>
</tr>
<tr>
<td>Susy k=8</td>
<td></td>
<td>38.02%</td>
<td>38.04%</td>
<td>37.92%</td>
<td>0.36%</td>
</tr>
<tr>
<td>Susy k=16</td>
<td></td>
<td>38.05%</td>
<td>38.02%</td>
<td>37.99%</td>
<td>0.36%</td>
</tr>
<tr>
<td>Higgs k=4</td>
<td></td>
<td>9.63%</td>
<td>9.21%</td>
<td>10.17%</td>
<td>10.26%</td>
</tr>
<tr>
<td>Higgs k=8</td>
<td></td>
<td>8.65%</td>
<td>9.52%</td>
<td>8.32%</td>
<td>8.32%</td>
</tr>
<tr>
<td>Higgs k=16</td>
<td></td>
<td>9.43%</td>
<td>8.34%</td>
<td>9.63%</td>
<td>9.63%</td>
</tr>
<tr>
<td>Airline78 k=4</td>
<td></td>
<td>3.74%</td>
<td>3.30%</td>
<td>4.19%</td>
<td>4.19%</td>
</tr>
<tr>
<td>Airline78 k=8</td>
<td></td>
<td>3.51%</td>
<td>3.24%</td>
<td>3.72%</td>
<td>3.72%</td>
</tr>
<tr>
<td>Airline78 k=16</td>
<td></td>
<td>3.25%</td>
<td>3.26%</td>
<td>3.72%</td>
<td>3.72%</td>
</tr>
<tr>
<td>Census k=4</td>
<td></td>
<td>1.37%</td>
<td>1.39%</td>
<td>1.37%</td>
<td>1.37%</td>
</tr>
<tr>
<td>Census k=8</td>
<td></td>
<td>1.33%</td>
<td>1.37%</td>
<td>1.41%</td>
<td>1.41%</td>
</tr>
<tr>
<td>Census k=16</td>
<td></td>
<td>1.31%</td>
<td>1.30%</td>
<td>1.39%</td>
<td>1.39%</td>
</tr>
<tr>
<td>Optical k=4</td>
<td></td>
<td>33.23%</td>
<td>32.60%</td>
<td>33.19%</td>
<td>4.99%</td>
</tr>
<tr>
<td>Optical k=8</td>
<td></td>
<td>32.68%</td>
<td>33.03%</td>
<td>33.26%</td>
<td>4.99%</td>
</tr>
<tr>
<td>Optical k=16</td>
<td></td>
<td>33.22%</td>
<td>32.89%</td>
<td>32.95%</td>
<td>4.99%</td>
</tr>
<tr>
<td>Mn.2m k=4</td>
<td></td>
<td>5.29%</td>
<td>5.31%</td>
<td>5.32%</td>
<td>0.73%</td>
</tr>
<tr>
<td>Mn.2m k=8</td>
<td></td>
<td>5.29%</td>
<td>5.31%</td>
<td>5.29%</td>
<td>0.73%</td>
</tr>
<tr>
<td>Mn.2m k=16</td>
<td></td>
<td>5.30%</td>
<td>5.31%</td>
<td>5.30%</td>
<td>0.73%</td>
</tr>
<tr>
<td>Im.Net k=4</td>
<td></td>
<td>3.84%</td>
<td>3.87%</td>
<td>3.90%</td>
<td>2.14%</td>
</tr>
<tr>
<td>Im.Net k=8</td>
<td></td>
<td>3.82%</td>
<td>3.84%</td>
<td>3.88%</td>
<td>2.14%</td>
</tr>
<tr>
<td>Im.Net k=16</td>
<td></td>
<td>3.78%</td>
<td>3.81%</td>
<td>3.86%</td>
<td>2.14%</td>
</tr>
</tbody>
</table>

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). 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 with 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 $(p_0 - p_r) / p_0$ where $p_0$ 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 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 CLA was set 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 size of the compressed matrix and the peak memory usage during the multiplication algorithm expressed as 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
<table>
<thead>
<tr>
<th>matrix</th>
<th>size</th>
<th>PM</th>
<th>time</th>
<th>size</th>
<th>PM</th>
<th>time</th>
<th>size</th>
<th>PM</th>
<th>time</th>
<th>PM</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Susy</td>
<td>68.99</td>
<td>77.53</td>
<td>0.35</td>
<td>65.99</td>
<td>82.77</td>
<td>0.45</td>
<td>76.14</td>
<td>—</td>
<td>—</td>
<td>53.27</td>
<td>63.09</td>
</tr>
<tr>
<td>Higgs</td>
<td>41.63</td>
<td>46.68</td>
<td>0.58</td>
<td>37.44</td>
<td>44.63</td>
<td>0.71</td>
<td>32.74</td>
<td>146.68</td>
<td>2.09</td>
<td>48.38</td>
<td>54.56</td>
</tr>
<tr>
<td>Airline78</td>
<td>9.35</td>
<td>16.06</td>
<td>0.17</td>
<td>8.13</td>
<td>16.43</td>
<td>0.23</td>
<td>12.34</td>
<td>120.27</td>
<td>1.17</td>
<td>13.27</td>
<td>17.53</td>
</tr>
<tr>
<td>Covtype</td>
<td>4.78</td>
<td>16.25</td>
<td>0.01</td>
<td>4.17</td>
<td>16.11</td>
<td>0.01</td>
<td>4.55</td>
<td>70.15</td>
<td>0.05</td>
<td>6.25</td>
<td>10.26</td>
</tr>
<tr>
<td>Census</td>
<td>2.00</td>
<td>5.70</td>
<td>0.01</td>
<td>1.55</td>
<td>7.25</td>
<td>0.02</td>
<td>3.77</td>
<td>108.96</td>
<td>0.16</td>
<td>5.54</td>
<td>7.92</td>
</tr>
<tr>
<td>Optical</td>
<td>36.05</td>
<td>44.50</td>
<td>0.06</td>
<td>34.93</td>
<td>56.39</td>
<td>0.09</td>
<td>40.44</td>
<td>176.90</td>
<td>0.20</td>
<td>53.55</td>
<td>57.26</td>
</tr>
<tr>
<td>Mnist2m</td>
<td>6.24</td>
<td>8.19</td>
<td>0.64</td>
<td>5.88</td>
<td>8.30</td>
<td>0.82</td>
<td>6.22</td>
<td>47.09</td>
<td>1.98</td>
<td>6.46</td>
<td>6.76</td>
</tr>
<tr>
<td>ImageNet</td>
<td>4.70</td>
<td>6.59</td>
<td>0.48</td>
<td>4.28</td>
<td>6.59</td>
<td>0.48</td>
<td>6.67</td>
<td>56.80</td>
<td>0.97</td>
<td>5.52</td>
<td>5.89</td>
</tr>
<tr>
<td>gzip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>106.14</td>
<td>0.17</td>
</tr>
</tbody>
</table>

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 in 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 achieve 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. 2017WT7SSH), 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 “HumanAI Network” (grant #952826); 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).