A compression-boosting transform for two-dimensional data*

Qiaofeng Yang[1], Stefano Lonardi[1], and Avraham Melkman[2]

[1] Dept. of Computer Science & Engineering
University of California
Riverside, CA 92521

[2] Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel 84105

Abstract. We introduce a novel invertible transform for two-dimensional data which has the objective of reordering the matrix so it will improve its (lossless) compression at later stages. The transform requires to solve a computationally hard problem for which a randomized algorithm is used. The inverse transform is fast and can be implemented in linear time in the size of the matrix. Preliminary experimental results show that the reordering improves the compressibility of digital images.

1 Introduction

Every day massive quantities of two-dimensional data are produced, stored and transmitted. Digital images are the most prominent type of data in this category. However, matrices over finite alphabets are used to represent all sorts of information, like graphs, database tables, geometric objects, etc. From the compression standpoint, two-dimensional data has to be treated differently than one-dimensional data. In order to obtain good compression of 2D data, one has to exploit the dependencies (or equivalently, expose the redundancies) both between the rows and the columns of the matrix.

Lossless compression algorithms are typically composed by a pipeline of independent stages, usually ending with a statistical encoder. For example, the celebrated `bzip2` employs a pipeline composed by the Burrows-Wheeler transform (BWT), a move-to-front encoder, and finally an Huffman compressor. Each step somewhat reorder the data so that redundancies get exposed and removed by the subsequent stages. The objective of the BWT is exactly that of elucidating the dependencies between adjacent symbols in the original text string.

In this paper we propose an invertible transform for two-dimensional data over an alphabet Σ. For simplicity, we assume $\Sigma = \{0, 1\}$. The extension to larger

* This project was supported in part by NSF CAREER IIS-0447773, and NSF DBI-0321756. AM was supported in part by the Paul Ivanier Center for Robotics Research and Production Management. A one-page abstract about this work appeared in the *Proceedings of Data Compression Conference*, Snowbird, Utah, 2005.

alphabets is immediate and is not pursued here. The goal of the transform is to "boost" the compression achieved by later stages. The transform is described by a simple recursive algorithm, which can be outlined as follows.

Given the matrix to be transformed, first search for the largest *columnwise-constant* (resp., *rowwise-constant*) submatrix, that is, a submatrix identified by a subset of rows and the columns (which are not necessarily contiguous) whose columns (resp., rows) are constant (i.e., either all 0 or 1). Reorder the rows and the columns such that the columnwise-constant (or rowwise-constant) submatrix is moved to the left-upper corner of the matrix. Recursively apply the transform on the rest of the matrix. Stop the recursion when the partition produces a matrix which is smaller than a predetermined threshold (see Figure 3 for an illustration of this process).

The intriguing question is whether this somewhat simple matrix transformation helps compression. Arguments can be made in favor or against. On one hand, each columnwise-constant (or rowwise-constant) submatrix can be represented compactly in a canonical form (first all the 0-columns, then all the 1-columns) by the list of its rows and columns. If a matrix can be decomposed into a small number of large constant submatrices, one would expect an improvement in compressibility. On the other hand, while the transform groups together portions of the matrix which are similar, the reordering can also break the local dependencies that exist in the original matrix between adjacent rows and columns. Breaking these dependencies could increase the entropy and have a negative impact on the compressibility.

The contribution of this paper is twofold. First, we present a novel invertible transform for 2D data. The design of the transform went through a series of refinements, and here we present the result of such process (Section 5). We also studied the computational cost of the transform, which turns out to be unbalanced. The inverse transform is extremely fast and simple, whereas the direct transform is very expensive. Our compression-boosting phase is therefore suitable to applications in which the data is compressed once and decompressed many times, like large repositories of digital images where images are stored and rarely modified.

The computational cost of the forward transform depends on the complexity of finding the largest columnwise-constant/rowwise-constant submatrix. In [1] we studied theoretically the general version of this problem. Although the problem turns out to be **NP**-hard, a relatively simple randomized algorithm that has good performance in practice was introduced in [1]. For completeness of presentation, we will briefly outline the algorithm in Section 4. The interested reader can refer to the original paper for more details.

Second, we study empirically the effects of the transform on the compressibility of two-dimensional data by comparing the performance of `gzip` and `bzip2` before and after the application of the transform on synthetic data and digital images. The preliminary results in Section 6 show that the transform boosts compression.

In closing, we want to point out that since our transform simply changes the representation of the data and it does not deal with the encoding problem (i.e., assigning bits to symbols), here we are not proposing a complete data compression software tool. Also, since we our transform is not optimized for digital images, the transform is not an image compression tool either. As said above, the primary use of our transform primary is as a preprocessing step to reorder the data so that the downstream compression with standard lossless encoder would be more efficient.

2 Related works

Since we are not proposing a new compression method, we will not delve into the vast literature on lossless image compression. There are however, a few related works on the idea of reordering the columns and/or the rows of a matrix with the objective of reducing the storage space or the access time to the element of the matrix.

In [3–5], the main concern is to compress database tables by exploiting the dependencies between the columns. In [3], Buchsbaum *et al.* observe that partitioning the table into blocks of columns and compressing each block of columns individually could improve compression. The key problem is to find the optimal partition of columns. In the follow-up paper [4], the authors add the possibility of rearranging the columns. Their tool, called `pzip` outperforms `gzip` by a factor of two on database tables. Along the same line of research, the authors of [5] introduce the k*-transform* which captures the dependencies between $k+1$ columns of a matrix. Although the problem of computing the k-transform for $k \geq 2$ is **NP**-hard, the proposed polynomial-time heuristic for the 2-transform performs remarkably well compared to `pzip`, `bzip2` and `gzip`.

The task of compressing boolean (sparse) matrices is also addressed in [6–10]. For example, in [9] the objective is to reorder the columns of a matrix such that the 1's in each row appear consecutively. Again, the problem of finding a reordering which minimizes the number of runs of 1's is **NP**-hard. This problem reduces to solving a traveling salesman problem for which the authors propose an heuristic algorithm. In [10] the objective is to find a reordering of both rows and columns of a boolean matrix so that the matrix can be broken into homogeneous rectangles and the description complexity involved in describing those rectangles (called *cross-association*) is minimized. The problem is defined in an information theoretical context and a two-stage heuristics algorithm is proposed.

3 Notations and problem definition

The input to the transform is a two-dimensional $n \times m$ matrix $X \in \{0,1\}^{n \times m}$. The element (i,j) of X is denoted by $X_{[i,j]}$. A contiguous submatrix from row i_1 to row i_2 and from column j_1 to column j_2 is denoted by $X_{[i_1:i_2,j_1:j_2]}$.

A *row selection* of size k of X is defined as a subset of the rows $R = \{i_1, i_2, \ldots, i_k\}$, where $1 \le i_s \le n$ for all $1 \le s \le k$. Similarly, a *column selection* of size l of X is defined as a subset of the columns $C = \{j_1, j_2, \ldots, j_l\}$, where $1 \le j_t \le m$ for all $1 \le t \le l$. Given a row selection R, we say that a column j, $1 \le j \le m$, is *constant* with respect to R if the symbols in the j-th column of X restricted to the rows in R, are identical.

The submatrix $X_{[R,C]}$ *induced* by the pair (R, C) is defined as the matrix

$$X_{[R,C]} = \begin{vmatrix} X_{[i_1,j_1]} & X_{[i_1,j_2]} & \cdots & X_{[i_1,j_l]} \\ X_{[i_2,j_1]} & X_{[i_2,j_2]} & \cdots & X_{[i_2,j_l]} \\ \cdots & \cdots & \cdots & \cdots \\ X_{[i_k,j_1]} & X_{[i_k,j_2]} & \cdots & X_{[i_k,j_l]} \end{vmatrix}$$

A submatrix induced by a pair (R, C) is called *columnwise-constant* (resp., *rowwise-constant*) if all its columns (resp., rows) are constant. Hereafter, for brevity we will use the term *constant* submatrix to denote either a columnwise-constant or a rowwise-constant submatrix.

Example. Given the 6×6 matrix $X = \begin{vmatrix} \mathbf{001011} \\ \mathbf{101101} \\ \mathbf{100110} \\ \mathbf{101001} \\ \mathbf{111101} \\ \mathbf{110110} \end{vmatrix}$ over the alphabet $\Sigma = \{\mathbf{0}, \mathbf{1}\}$, a selection $(R, C) = (\{2, 4, 5\}, \{1, 3, 5, 6\})$ results in the columnwise-constant submatrix $X_{[R,C]} = \begin{vmatrix} \mathbf{1101} \\ \mathbf{1101} \\ \mathbf{1101} \end{vmatrix}$. $X_{[R,C]}$ is the largest area columnwise-constant submatrix in X.

The main computational problem is the following. Given a matrix $X \in \{0, 1\}^{n \times m}$, find a constant submatrix with the largest area. This problem is strongly related to the Maximum Edge Biclique problem since a $n \times m$ binary matrix can also be interpreted as the adjacency matrix of a bipartite graph. The biclique problem requires to find the biclique which has the maximum number of edges which corresponds to the largest constant submatrix composed only of 1's. This problem was proved to be **NP**-hard in [11] by reduction to 3Sat. The weighted version of this problem was shown to be **NP**-hard by Dawande *et al.* [12]. A 2-approximation algorithm based on LP-relaxation was given in [13].

4 Finding the largest columnwise-constant submatrix

Given that the problem of finding the largest constant submatrix of 1's is **NP**-hard, it is unlikely that a polynomial time algorithm could be found. In [1] we introduced a randomized algorithm which is able to find the optimal solution with probability $1 - \epsilon$, where $0 < \epsilon < 1$. For completeness of presentation,

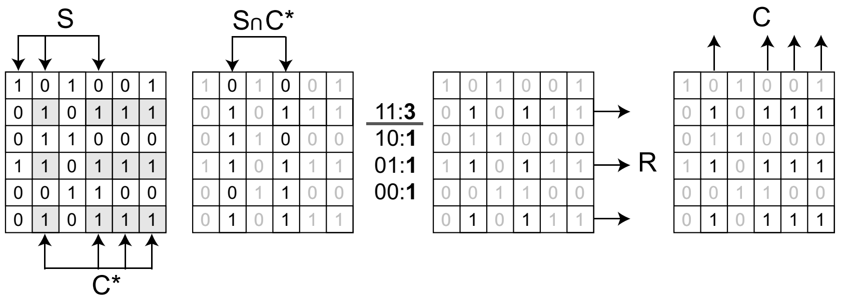

Fig. 1. An illustration of a recovery of a constant submatrix (shaded boxes), assuming $r^* = 3$

next we give a brief outline of the algorithm for the largest columnwise-constant submatrix. Rowwise-constant submatrices can be found along the same lines.

Recall that we are given a matrix $X \in \{0,1\}^{n \times m}$ and the objective of the algorithm is to discover a columnwise-constant submatrix $X_{(R^*,C^*)}$. Let us assume that the submatrix $X_{(R^*,C^*)}$ is maximal. To simplify the notation, let us call $r^* \equiv |R^*|$ and $c^* \equiv |C^*|$.

The key idea is the following. Observe that if we knew R^*, then C^* could be determined by selecting the constant columns with respect to R^*. If instead we knew C^*, then R^* could be obtained by taking the maximal set of rows which read the same symbol on the columns C^*. Unfortunately, neither R^* nor C^* is known. Our approach is to "sample" the matrix by randomly selecting subsets of columns (or rows), expecting that eventually one of the subsets will overlap with the solution (R^*, C^*).

In the following we describe how to retrieve the solution by sampling columns (one has also the choice to sample the rows). First, select a subset S of size k uniformly at random from the set of columns $\{1, 2, \ldots, m\}$. Assume for the time being that $S \cap C^* \neq \emptyset$. If we knew $S \cap C^*$, then (R^*, C^*) could be determined by the following three steps (1) select the string(s) w that appear exactly r^* times in the rows of $X_{[1:n, S\cap C^*]}$, (2) set R^* to be the set of rows in which w appears and (3) set C^* to be the set of constant columns corresponding to R^*. An example is illustrated in Figure 1.

The algorithm would work, but there are a few problems that need to be solved. First, the set $S \cap C^*$ could be empty. The solution is to try several different sets S, relying on the argument that the probability that $S \cap C^* \neq \emptyset$ *at least once* will approach one with more and more selections. The second problem is that we do not really know $S \cap C^*$. But, certainly $S \cap C^* \subseteq S$, so our approach is to check all possible subsets of S. The final problem is that we assumed that we knew r^*, but we do not. The solution is to introduce a *row threshold* parameter, called $\hat{r}$, that replaces r^*.

As it turns out, we need another parameter to avoid producing submatrices with small area which could potentially degrade the compressibility at later

```
LARGEST_COLUMNWISE_CONSTANT_SUBMATRIX($X, t, k, \hat{r}, \hat{c}$)
INPUT: $X$ is a $n \times m$ matrix over $\{0,1\}$
          $t$ is the number of iterations
          $k$ is the selection size
          $\hat{r}, \hat{c}$ are the "thresholds" on the number of rows and columns, resp.
1   repeat $t$ times
2       select randomly a subset $S$ of columns such that $|S| = k$
3       for all subsets $U \subseteq S$ do
4           $D \leftarrow$ all strings composed of either 0 or 1 induced by $X_{[1:n,U]}$ that
                appear at least $\hat{r}$ times
5           for each string $w$ in $D$
6               $V \leftarrow$ rows corresponding to $w$
7               $Z \leftarrow$ all constant columns corresponding to $V$
8               if $|Z| \geq \hat{c}$ then save $(V, Z)$
9   return the $(V, Z)$ that maximizes $|V| \times |Z|$
```

Fig. 2. A sketch of the algorithm that discovers large columnwise-constant submatrices

stages as discussed in Section 5. The *column threshold* parameter $\hat{c}$ is used to discard submatrices whose number of columns is smaller than $\hat{c}$. The algorithm considers all the submatrices which satisfy the user-defined row and column thresholds as candidates. Among all candidate submatrices, only the ones that maximize the total area are kept.

A sketch of the algorithm is shown in Figure 2. The algorithm depends on four key parameters, namely the selection size k, the row threshold $\hat{r}$, the column threshold $\hat{c}$, and the number of iterations t. A detailed discussion on how to choose each of these can be found in [1]. The worst case time complexity of the algorithm LARGEST_COLUMNWISE_CONSTANT_SUBMATRIX is $O\left(tk2^k(kn+nm)\right)$.

Because of the randomized nature of the approach, there is no guarantee that the algorithm will find the solution after a fixed number of iterations. We therefore need to choose t so that the probability that the algorithm will recover the solution in at least one of the t trials is $1-\epsilon$, where $0 < \epsilon < 1$. In [1], we proved that the algorithm is able to find the maximal solution with probability $1-\epsilon$ when the number of random selections t satisfies

$$t \geq \frac{\log \epsilon}{\log\left(\binom{m-c^*}{k} + \sum_{i=1}^{k}\left(1-\left(1-\frac{1}{|\Sigma|^i}\right)^{n-r^*}\right)\binom{c^*}{i}\binom{m-c^*}{k-i}\right) - \log\binom{m}{k}} \qquad (1)$$

5 Forward transform

As mentioned in the introduction, our strategy to boost the compressibility of two-dimensional data is to recursively decompose the input matrix based on the presence of large columnwise-constant or rowwise-constant submatrices found by the randomized search described above. The input to the recursive decomposition algorithm is the original matrix X along with user-defined thresholds ($\hat{r}$ and $\hat{c}$) and the number of iterations t. If one fixes ϵ, then the number of iterations t can be computed using equation (1).

The recursive decomposition is carried out as follows. First, the procedure LARGEST_COLUMNWISE_CONSTANT_SUBMATRIX (and possibly also the procedure LARGEST_ROWWISE_CONSTANT_SUBMATRIX) is ran on X. If a constant submatrix is found, the rest of the matrix is partitioned into two submatrices depending on the size of the constant submatrices discovered at the next recursion level, as illustrated in Figure 3.

The decision whether to choose the partition ($\mathsf{a}+\mathsf{c}, \mathsf{b}$) or the partition ($\mathsf{a}, \mathsf{b}+\mathsf{c}$) depends on the size of the constant submatrices found in the resulting matrices $\mathsf{a}+\mathsf{c}$, b, $\mathsf{b}+\mathsf{c}$, and a. Let us call A_1, A_2, B_1, B_2 the areas of the constant submatrices found in $\mathsf{a}+\mathsf{c}$, b, a, $\mathsf{b}+\mathsf{c}$, respectively. Based on the values of A_1, A_2, B_1, and B_2 we studied three distinct criteria to determine the partition. The first is based on the condition $A_1 + A_2 > B_1 + B_2$ (hereafter called `sum`). The second and the third tests are $\max\{A_1, A_2\} > \max\{B_1, B_2\}$ (called `max`) and $A_1 > B_2$ (called `indiv`[3]). In all cases, if the test is true the algorithm chooses the partition ($\mathsf{a}+\mathsf{c}, \mathsf{b}$). Otherwise, the algorithm chooses the partition ($\mathsf{a}, \mathsf{b}+\mathsf{c}$). A discussion on how the test type affects the final compressed size is reported in Section 6.

Once the partition is determined, the randomized search is performed recursively on the newly formed matrices in the same manner. The recursion stops when the matrix becomes non-decomposable. We say that a matrix is *non-decomposable* if either it has less than $\hat{r}$ rows or less than $\hat{c}$ columns, or if the largest constant submatrix contained in it is smaller than $\hat{r} \times \hat{c}$.

The reason behind our choice of splitting in ($\mathsf{a}+\mathsf{c}, \mathsf{b}$) or ($\mathsf{a}, \mathsf{b}+\mathsf{c}$), instead of ($\mathsf{a}, \mathsf{b}, \mathsf{c}$) is the following. Each time the algorithms partitions the matrix, we risk to split large constant submatrices that we could have potentially found later. The smaller is the number of matrices we split, the higher are the chances of finding large constant submatrices. Experimental results (not shown) confirmed our choice.

It should be noted that the user-defined thresholds ($\hat{r}$ and $\hat{c}$) play an important role in the transform. If the thresholds are too low, there is a danger of having a deep recursion tree and potentially finding a large number of tiny constant submatrices. If the thresholds are too high, there will be just a few constant submatrices. Both cases will have a negative impact on the compression. An experimental study regarding the choice of these thresholds is reported in Section 6.

6 Implementation, Experiments and Results

We now describe how the transformed data is represented. Clearly, each constant submatrix can be represented very succinctly. The column indices of columnwise-constant submatrices are reordered so that each row reads `00...0011...11`. Thus,

[3] note that in this latter case we do not need to search in `b` and `a`.

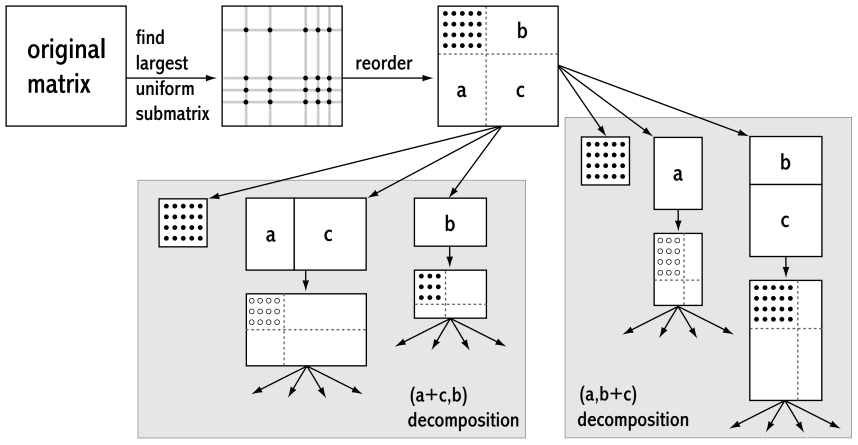

Fig. 3. Illustration of one step of the forward transform. Depending on the size of the constant submatrices in a+c, b, a, b+c either the decomposition $(\mathsf{a}+\mathsf{c},\mathsf{b})$ or $(\mathsf{a},\mathsf{b}+\mathsf{c})$ is chosen.

each constant submatrix can be represented by the list of rows and column indices, and where the transition from `0` to `1` takes place. Non-decomposable submatrices are saved contiguously in row-major order. The content of non-decomposable matrices is saved in a file called `string`.

Row and column indices of constant and non-decomposable submatrices are saved in another file called `index`. For each set of row and column indices, the first index is saved as it is, while the rest is saved as differences between adjacent indices. The `length` file is used to record the number of rows and the number of columns for constant and non-decomposable submatrices, along with a binary flag to indicate whether the submatrix is constant or non-decomposable.

The information contained in the files `string`, `index` and `length` allows one to invert the transform and reconstruct the original matrix. The inverse transform is simple and extremely fast. Basically, the matrix is reconstructed element by element in the order of the indices stored in `index`. The inverse transform was implemented and tested to make sure that we had all the information necessary to recover the original matrix. The time complexity of the inverse transform is linear in the size of input.

In order to determine whether the transform improves compression, we compared the size of the file obtained by compressing the original matrix against the overall size of the files `string`, `index` and `length` compressed with the same program. We employed two popular lossless compression algorithms, namely `gzip` and `bzip2`.

We tested the three criteria (`sum`, `max`, `indiv`) discussed in Section 5 on several images and simulated data. The result on the image `bird` is reported in Figure 4 for different choices of the thresholds. In the majority of our experi-

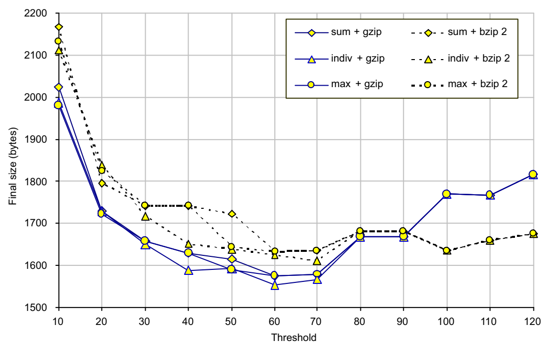

Fig. 4. The performance of the algorithm on the image `bird` for different strategies (`sum`, `max`, `indiv`) in choosing how to partition the matrix

filename	`gzip`	transform+`gzip`	`bzip2`	transform+`bzip2`
$\texttt{matrix}_1$	11,121	**10,041**	11,014	**10,197**
$\texttt{matrix}_2$	11,111	**9,536**	11,051	**9,712**
$\texttt{matrix}_3$	11,094	**8,989**	10,951	**9,194**
$\texttt{matrix}_4$	11,061	**8,395**	10,919	**8,530**

Table 1. Results on 256 × 256 synthetic data. File $\texttt{matrix}_i$ contains i embedded columnwise-constant submatrices of size 64×64. Parameters: $\hat{r} = 10, \hat{c} = 10, t = 10,000$

ments, the strategy `indiv` appeared to be the best. Therefore, all experimental tests that follow employ the `indiv` test.

6.1 Simulations on synthetic data

We generated several datasets, each composed of four random matrices of size 256 × 256 over a binary alphabet. In each of the four matrices we embedded one, two, three, and four columnwise-constant submatrices of size 64×64, respectively. The position and the content of each embedded submatrix were randomly chosen under the condition that the submatrices did not overlap with each other.

For each matrix we compared the compression size obtained with `gzip` and `bzip2` before and after the transform. The performance of the transform was measured on several datasets. Table 1 shows the results averaged on all datasets. The results are very stable with respect to the choices of $\hat{r}$ and $\hat{c}$. Any choice in the range 10 to 60 produces almost identical results. The number of iterations t was set to $10,000$.

The goal of these simulations was twofold. First, it allowed us to test the ability of our randomized search to recover the embedded submatrices. Failures in recovering all the submatrices are typically due to the recursive partitioning. If the partitioning process happens to split an embedded submatrix, there is no

filename	*size*	gzip	transform+gzip	bzip2	transform+bzip2
bird	65,792	1,978	**1521**	1,778	**1581**
camera	65,792	4,330	**3693**	3,839	**3664**
lena	65,792	3,450	**3186**	3,026	**2930**
peppers	65,792	3,186	**2941**	2,757	**2671**
tulips	65,792	5,133	**4695**	4,483	**4329**

Table 2. Comparing the compressibility of 256×256 binary images before and after the transform. Threshold parameters $\hat{r} = 60, \hat{c} = 60$. Iterations $t = 10,000$

hope of recovering it as a single piece. This observation is behind the idea of partitioning into $(\mathsf{a}+\mathsf{c}, \mathsf{b})$ or $(\mathsf{a}, \mathsf{b}+\mathsf{c})$ instead of partitioning into. $(\mathsf{a}, \mathsf{b}, \mathsf{c})$. That may avoid splitting a potentially large constant submatrix that lies in $\mathsf{a}+\mathsf{c}$ or $\mathsf{b}+\mathsf{c}$

Second, this synthetic data is arguably the most favorable type of data for our transform. The "background" of the matrix is random, and therefore there are very few dependencies between rows and columns. The large majority of the dependencies are the ones created by the embedding of the constant submatrices. In some sense, this data represents the best case scenario. This is shown by a considerable improvement in the file size after the transform is applied.

6.2 Experiments on digital binary images

In order to determine whether the transform can boost the compressibility of general data, we tested the transform on five 256×256 images downloaded from the Internet (namely `bird`, `camera`, `lena`, `peppers`, and `tulips`, all of which are commonly used in the data compression community). Each 8 bpp greyscale image was converted to binary by setting to black each pixel whose brightness was below 128 and to white each pixel above 128. Table 2 shows the compression results before and after the transform (for $\hat{r} = 60, \hat{c} = 60, t = 10,000$). In all the images we tested, the transform improves the lossless compression downstream.

In these experiments, we considered only columnwise-constant matrices. We tested an implementation of the transform that also searches for rowwise-constant matrices, but it did not boost the compressibility further.

Although a comparison of the results in Table 2 against a specialized lossless image tool, say JBIG, would appear appropriate, it is not. Our transform is (1) general purposed (i.e., not optimized for digital images), and (2) not a complete compression tool. As said in the introduction, we do not deal with the encoding problem (we are in fact relying on `gzip` and `bzip2`), nor we are necessarily bound to process digital images. If we were to compare the results of the table against JBIG, the performance of `gzip`/`bzip2` would also part of the equation, and these tools have not been designed specifically to compress digital images.

Next, we tested how sensitive is the transform to the choice of the parameters $\hat{r}$ and $\hat{c}$, and to the number of iteration t. We selected the image `bird`,

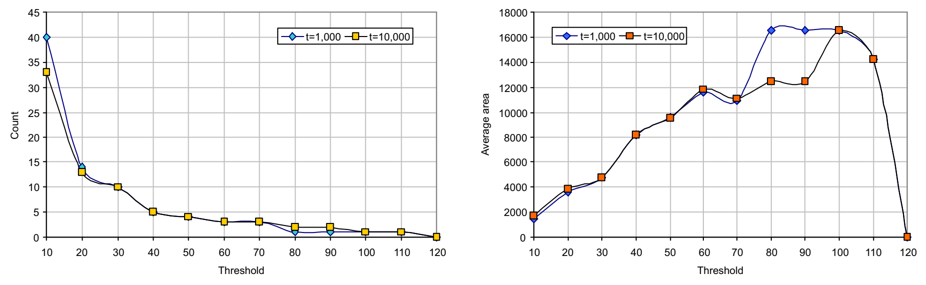

Fig. 5. LEFT: Number of columnwise-constant submatrices in image `bird`. RIGHT: Average area of columnwise-constant submatrices in image `bird`

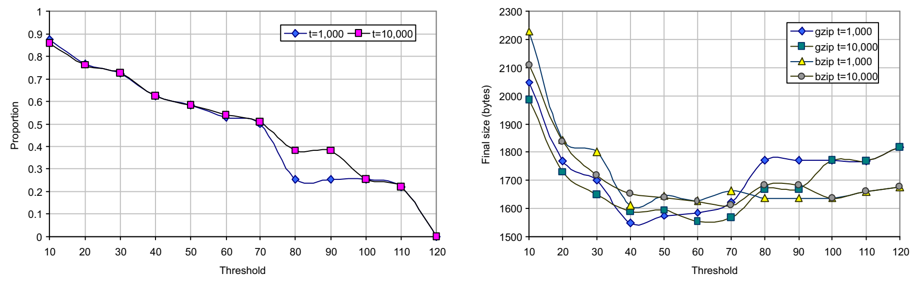

Fig. 6. LEFT: Proportion of columnwise-constant submatrices in image `bird`. RIGHT: Comparing the final compression size for the image `bird` for different choices of the threshold parameters

and we ran the transform on different parameter choices. We computed the total number and the average area of the columnwise-constant submatrices found (Figure 5), for several choices of $\hat{r} = \hat{c}$ and for two values for t. We also recorded the total proportion of the matrix which was covered by columnwise-constant submatrices (the rest is non-decomposable), and the final size of the files after compression (Figure 6). Observe that when the thresholds are low, the proportion of the matrix covered by columnwise-constant submatrices is quite high. However with low thresholds, the transform finds a large number of columnwise-constant submatrices which average area is low (Figure 5), which in turn results in large file sizes for `index` and `length`. Compared to the file `string`, files `index` and `length` are considerably harder to compress. Therefore, the consequence of choosing thresholds too low is poor compression boosting. Good compression relies on finding a balance between the gain of representing a portion of the matrix a single bit and the cost of adding the extra information necessary to reconstruct the original matrix.

The optimal value of the thresholds $\hat{r} = \hat{c}$ for the image `bird` is around 40, but other values in the range 40 to 70 achieve very similar results. We carried out the same analysis on other 256×256 images, and the same general considerations apply. With respect to the final compression, in most cases the larger is the number of iterations t, the better is the compression.

References

1. Lonardi, S., Szpankowski, W., Yang, Q.: Finding biclusters by random projections. In: Proceedings of Symposium on Combinatorial Pattern Matching (CPM'04). Volume 3109 of LNCS., Istanbul, Turkey, Springer (2004) 74–88
2. Storer, J.A., Helfgott, H.: Lossless image compression by block matching. Comput. J. **40**(2/3) (1997) 137–145
3. Buchsbaum, A.L., Caldwell, D.F., Church, K.W., Fowler, G.S., Muthukrishnan, S.: Engineering the compression of massive tables: an experimental approach. In: Proceedings of the ACM-SIAM Annual Symposium on Discrete Algorithms, San Francisco, CA (2000) 213–222
4. Buchsbaum, A.L., Fowler, G.S., Giancarlo, R.: Improving table compression with combinatorial optimization. In: Proceedings of the ACM-SIAM Annual Symposium on Discrete Algorithms, San Francisco, CA (2002) 175–184
5. Vo, B.D., Vo, K.P.: Using column dependency to compress tables. In Storer, J.A., Cohn, M., eds.: Data Compression Conference, Snowbird, Utah, IEEE Computer Society Press, TCC (2004) 92–101
6. Galli, N., Seybold, B., Simon, K.: Compression of sparse matrices: Achieving almost minimal table sizes. In: Proceedings of Conference on Algorithms and Experiments (ALEX98), Trento, Italy (1998) 27–33
7. Bell, T., McKenzie, B.: Compression of sparse matrices by arithmetic coding. In Storer, J.A., Cohn, M., eds.: Data Compression Conference, Snowbird, Utah, IEEE Computer Society Press, TCC (1998) 23–32
8. McKenzie, B., Bell, T.: Compression of sparse matrices by blocked Rice coding. IEEE Trans. Inf. Theory **47**(3) (2001) 1223 – 1230
9. Johnson, D.S., Krishnan, S., Chhugani, J., Kumar, S., Venkatasubramanian, S.: Compressing large boolean matrices using reordering techniques. In: To appear in Proceedings of International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada (2004)
10. Chakrabarti, D., Papadimitriou, S., Modha, D., Faloutsos, C.: Fully automatic cross-assocations. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-04), ACM Press (2004) 89–98
11. Peeters, R.: The maximum-edge biclique problem is NP-complete. Technical Report 789, Tilberg University: Faculty of Economics and Business Adminstration (2000)
12. Dawande, M., Keskinocak, P., Swaminathan, J.M., Tayur, S.: On bipartite and multipartite clique problems. Journal of Algorithms **41** (2001) 388–403
13. Hochbaum, D.S.: Approximating clique and biclique problems. Journal of Algorithms **29**(1) (1998) 174–200