On Self-Indexing Images — Image Compression with Added Value

Veli Mäkinen [*]
Dept. of Computer Science
University of Helsinki, Finland
vmakinen@cs.helsinki.fi

Gonzalo Navarro [†]
Dept. of Computer Science
University of Chile, Chile
gnavarro@dcc.uchile.cl

Abstract

Recent advances in compressed data structures have led to the new concept of *self-indexing*; it is possible to represent a sequence of symbols compressed in a form that enables fast queries on the content of the sequence. This paper studies different analogies of self-indexing on images. First, we show that a key ingredient of many self-indexes for sequences, namely the *wavelet tree*, can be used to obtain both lossless and lossy compression with random access to pixel values. Second, we show how to use self-indexes for sequences as a black-box to provide self-indexes for images with filtering-type query capabilities. Third, we develop a tailor-made self-index for images by showing how to compress two-dimensional suffix arrays. Experimental results are provided to compare the compressibility to standard compression methods.

1 Introduction

In the compression of one-dimensional data, a new intriguing possibility has emerged: One can compress the data into a representation that works as a flexible index structure for the content of the data [8]. The indexing functionalities range from the random access to data elements to pattern search cabablities. These compressed representations are often called *self-indexes* as they replace the original data, and can thus be considered as compression methods with some added value.

A natural question arises: Would it be possible to find analogous representations for two-dimensional data? This is not only of theoretical interest. Random access to pixel values can be valuable in hierarchical memories, where a large image can be stored compressed in a big but slow memory level (e.g. main memory) and its portions can be decompressed and copied into a faster memory level (e.g. graphics card cache); or in an image server that wishes to transmit portions of images.

Another appealing application is image retrieval by content. A typical query scenario in multimedia search engines is one where a sample image is pointed by the user and all similar ones in the database are retrieved. A self-index for the database of images would solve this problem in the special case of exact searching. A more

[*]Funded by the Academy of Finland under grant 119815.
[†]Funded by Yahoo! Research grant “Compact Data Structures”.

realistic scenario is to use self-indexes as filters; significant small features can be extracted from the query template and their rotated/scaled/altered occurrences can be sought for exactly using the self-indexes.

In this paper we show that a key ingredient of many self-indexes for sequences, namely the *wavelet tree*, can be used to obtain both lossless and lossy compression with random access to pixel values. The method can be seen as an improvement over the classical bit-plane encoding, as confirmed by our experiments. Connection to the popular *wavelet transform* is also discussed. We also show how to use a self-index for sequences as a black-box to provide self-indexing for images with filtering-type query capabilities. Finally, we develop a tailor-made self-index for images by showing how to compress *two-dimensional suffix arrays*. We report some encouraging preliminary experiments on the implementation of this method.

2 Wavelet trees for image compression

We develop an image representation that stores it in compressed form and allows random acces to its pixels. The representation is progressive, in the sense that a given detail level can be chosen when accessing the pixel values. Alternatively, one can ignore the least significant bits and achieve lossy compression.

The basic tool used in the representation is the bit-vector *rank* operation: query $rank(B,i) = rank_1(B,i)$ returns the number of bits set in the prefix $B[1,i]$ of a bit vector $B[1,n]$. Symmetrically, $rank_0(B,i) = i - rank_1(B,i)$. The dual query to $rank_1$ is $select(B,j)$, giving the position of the j-th bit set in B. All those can be supported in constant time and little space. We will use one [9] that in addition compresses B to $nH_0(B)+o(n)$ bits, where H_0 is the zero-order entropy of B: $H_0(B) = \frac{n_0}{n}\log\frac{n}{n_0} + \frac{n_1}{n}\log\frac{n}{n_1}$, being n_0 (n_1) the number of 0s (1s) in B ($\log = \log_2$ henceforth).

In the sequel, let us consider an $n \times n$ image I, where each $I[i,j] \in \{0,1,\ldots,\sigma-1\}$. For example, on an 8-bit grayscale image we would have $\sigma = 256$, and on a 24-bit RGB image we would have $\sigma = 16,777,216$. In the latter case, we assume that the RGB values are interleaved so that their most significant bits form the three most significant bits of the 24-bit value, and so on. For this section it is enough to consider the image row-by-row. Let $A = a_1 \cdots a_n$ be a row under consideration.

The representation uses a structure called *wavelet tree* [4], which is is a balanced binary tree whose leaves represent the symbols in the alphabet. The root is associated with the whole sequence $A = a_1 \cdots a_n$ (a row of the image), its left child with the subsequence of A obtained by concatenating all positions i having $a_i < \sigma/2$, and its right child with the complementary subsequence (symbols $a_i \geq \sigma/2$). This subdivision is continued recursively, until each leaf contains a repeat of one symbol. The sequence at each internal node is represented by a bit vector that tells which positions (those marked with 0) go to the left child, and which (marked with 1) go to the right child. Those bit vectors alone are enough to determine the original sequence: To recover a_i, start at the root and go left or right depending on the bit vector value B_i at the root. When going to the left child, replace $i \leftarrow rank_0(B,i)$, and similarly $i \leftarrow rank_1(B,i)$ when going right. When arriving at the leaf of symbol c it must hold that the original

a_i is c. This requires $O(\log \sigma)$ *rank* queries over bit vectors.

The spaces needed to represent each bit vector in compressed form to have constant-time *rank* queries [9] add up to $nH_0(A) + o(n \log \sigma)$ bits [4]. Added over the whole image of $n^2 = N$ cells, one achieves $NH_0(I) + o(N \log \sigma)$, where $H_0(I) = \sum_{0 \le c < \sigma} \frac{N_c}{N} \log \frac{N}{N_c}$, where I has N_c cells of color c. Note that $N \log \sigma$ is the space to store I in raw form.

We experimented with the wavelet tree without the sublinear *rank* structures to see how well the wavelet tree manages for plain compression. In practice adding *rank* structures allow for a flexible space/time tradeoff. The wavelet tree is easy to adjust to provide lossy compression; just stop the encoding at certain level of the tree.

Since the first levels of the tree are expected to contain long runs of 0s or 1s, we consider encoding those runs using Elias δ-encoding. Each level of the tree is encoded either with δ-encoding, with identifier coding [9], or uncompressed, depending on which takes less space. All encodings permit to attach the sublinear structures to support *rank*. We divided the colors in the wavelet trees so as to leave half of the different colors in each child. Table 2 shows the compression achieved.

Comparison to bit-plane encoding. Pruning the last wavelet tree levels is equivalent to dropping the least significant bit planes. Hence, the method can be seen as a new variant of the classical bit-plane encoding, which encodes each bit plane independently using methods for bit vector compression. The difference is that now the order of the bits in a bit plane is determined by its parent bit plane: We put together the second plane of the cells that share the same first plane, which should take advantage of the local homogeneity. Table 2 shows that compression actually improves.

Wavelet tree and wavelet transform. The popular wavelet transform used, e.g., in the JPEG-2000 standard can be seen as a generic variant of wavelet tree compression. In both, the original data is transformed into a data stream whose tail can be cut losing only the least significant bits of the message. They also share their recursive nature, where a child receives a shuffled message from its parent.

However, wavelet transforms represent a much richer family of compressors, taking into account more than just simple spatial features. Hence, the proposed method cannot compete with the best wavelet-transform-based methods in terms of compressibility. Yet, its added value is the random access to pixel values, and more notably, the possibility of adding search capabilities to the representation, explored next.

3 Self-indexing for filtering image searching

We will briefly sketch how to use the existing (one-dimensional) full-text self-indexes as filters for the two-dimensional case of images.

Full-text self-indexes. We start with a classical full-text index for a text string $T[1, n]$. The *suffix array* $\mathcal{A}[1, n]$ of T is an array of pointers to all the suffixes of T in lexicographic order [7]. Assume T is terminated by a unique endmarker "\$", so that lexicographic comparisons are well defined. $\mathcal{A}[i]$ points to text suffix $T[\mathcal{A}[i], n] = t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \ldots t_n$, and it holds $T[\mathcal{A}[i], n] < T[\mathcal{A}[i+1], n]$ in lexicographic order.

Given $\mathcal{A}$ and T, the occurrences of a pattern $P = p_1p_2 \ldots p_m$ can be counted in $O(m \log n)$ time, or even $O(m + \log n)$ using extra longest common prefix (lcp) information. The occurrences form an interval $\mathcal{A}[sp, ep]$ such that suffixes $t_{\mathcal{A}[i]}t_{\mathcal{A}[i]+1} \ldots t_n$, for all $sp \leq i \leq ep$, contain the pattern P as a prefix. This interval can be searched for using two binary searches. Once the interval is obtained, the occurrence positions are located by listing all its pointers in constant time each.

Self-indexes [8] replace the suffix array (and also the text) with a structure using $O(n \log \sigma)$ bits instead of the $O(n \log n)$ bits required by the suffix array. For example, the self-index in [6] consists of three parts: (1) Wavelet tree of the Burrows-Wheeler [2] transform (BWT) of text T. (2) Array C storing for each symbol c the number of occurrences of symbols smaller than c in T. (3) Some sub-linear data structures to support access to regularly sampled suffix array values.

This structure is shown to take space close to high-order entropy [6]. It can be used to find the suffix array interval $\mathcal{A}[sp, ep]$ containing pattern occurrences in time $O(m\lceil\frac{\log \sigma}{\log \log n}\rceil)$. Locating each occurrence takes $O(\log^{1+\epsilon} n)$ time, for any constant $\epsilon > 0$. Reproducing any text substring of length ℓ takes time $O(\ell\lceil\frac{\log \sigma}{\log \log n}\rceil + \log^{1+\epsilon} n)$.

Applying full-text self-index for images. We can apply the structure described above to an image I by concatenating all rows of I (appending the endmarker to each) into a sequence of length $N = n^2$. To search for an $m \times m$ pattern P, we search for each of its rows independently using the self-index of concatenated I. Let $occ(i)$ be the set of occurrence positions for pattern row i. The occurrence positions of the whole pattern P in T are then given by $occ(1) \cap (occ(2) - n) \cap (occ(3) - 2n) \cap \cdots \cap (occ(m) - (m-1)n)$, where $occ(i) - x = \{y - x \mid y \in occ(i)\}$.

The time requirement of the method depends on intermediate results, i.e., on the sizes of $occ(i)$. Typical heuristics can be used to improve the filtering efficiency, but in the worst case the overall size of the $occ(i)$'s can be mn. The scheme can obviously be extended to rectangular images and patterns, and to handle image collections.

Coping with large alphabets. A problem with this approach is that self-indexes have not been designed for extremely large alphabets. For example, parts (1) and (3) described above work well with alphabets of any size, but part (2) poses a problem. The array C (used in some form in all suffix-array-based self-indexes we know of [8]) requires $\sigma \log N$ bits, which on RGB images can be much more than $N \log \sigma$ (on a large image collection of total size N this might not be a problem).

A solution is to implement C as a bitmap $C_I[1, N]$ where only the bits at positions $C[c]$ are set; hence $C[c] = select(C_I, c)$. This bitmap requires $N + o(N)$ bits, compressible to $\sigma_I \log \frac{N}{\sigma_I} + o(N)$ [9], where $\sigma_I \leq \min(\sigma, N)$ is the number of different colors appearing in the images. We also need another bitmap $C_C[1, \sigma]$ to mark those σ_I colors. Hence, we work all the time on mapped colors (i.e., color c is represented as $c' = rank(C_C, c)$): search patterns must first be mapped using $rank$ on C_C, and any color c' displayed by the self-index must be converted back to the original value $c = select(C_C, c')$ in order to show it. The space for C_C is $\sigma + o(\sigma)$ bits. For the reduced functionality we need from it (i.e., only $rank(C_C, c)$ where $C_C[c] = 1$, and $select$) it can be represented using just $\sigma_I \log \frac{\sigma}{\sigma_I} + o(\sigma_I) + O(\log \log \sigma)$ bits [9].

4 A self-index based on 2-dimensional suffix arrays

In this section we move forward towards a self-indexing technique that gives worst-case time guarantees on the search time. We build on a well-known technique to index images based on so-called L-suffixes (e.g. [3, 5]) The idea is that each position $I[i,j]$ of an image defines a *2-dimensional suffix*, whose successive characters are L-shaped bands of increasing size which start at (i,j) and grow towards larger i and j values. More formally, the ℓ-th "character" of the suffix starting at (i,j), for $\ell \geq 0$, is the sequence (taken as a single symbol) $I[i,j+\ell]\ I[i+1,j+\ell]\ I[i+2,j+\ell] \ldots I[i+\ell-1,j+\ell]\ I[i+\ell,j]\ I[i+\ell,j+1]\ I[i+\ell,j+2] \ldots I[i+\ell,j+\ell]$ (the reading order has been chosen at our convenience, see later). Figure 1 (left) illustrates. If we reach the border of an image, the suffix ends there, although one assumes that the image has an additional row and column of unique values, so as to ensure that any lexicographical comparison among different suffixes will finish before exceeding the image area.

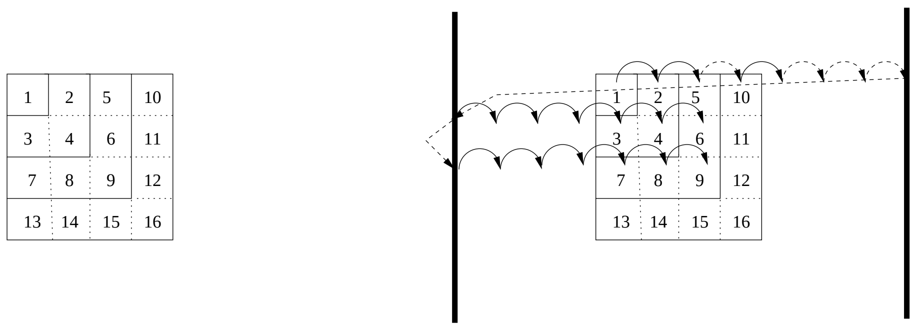

Figure 1: On the left, the reading order of the consecutive cells of an L-suffix. On the right, the mechanism to access them with our sampling. Curved arrows correspond to applications of Ψ to move rightwards in the (virtual) image. Dashed ones are those carried out to find out where we are in the image and be able to move to other rows. Solid ones are those carried out to obtain the cell values of interest. Thick vertical lines are the sampled columns, and dashed straight lines are the steps done with the sampling information. At this point the *front* is formed by the suffix array values pointing to the cells labeled 5,6,9. To obtain the new L-band (shown in thick lines) we obtain cells 10,11,12 by applying Ψ once to each cell in the current front, and then obtain the whole new bottom row.

By building a suffix tree over those L-suffixes (which can be done in linear time) it is possible to find all the positions where any $m \times m$ (square) pattern occurs in time $O(m^2)$, that is, linear in the pattern size.

It is not hard to derive a suffix array technique from the suffix tree. The suffix array just points to all the (i,j) image positions, in lexicographical order of the corresponding L-suffixes (seen just as a concatenation of the sequences of pixel values read in that L-order). It is also possible to index a collection of images using just one suffix array, so that each suffix array position points to some cell of some image. In this case, if N is the overall size of the collection (in cells), the search cost is $O(m^2 \log N)$, which can be reduced to $O(m^2 + \log N)$ by using some extra lcp information [7]. After this search, the position of each occurrence can be obtained in constant time. The suffix array (and extra lcp structures, if desired) can be built in time $O(N \log N)$ [5], and occupy $O(N \log N)$ bits of space. This is too much compared to the $N \log \sigma$ bits to store the raster images, which must also be maintained to permit searching.

In order to reduce space, we use the concepts developed in Sadakane's Compressed Suffix Array (CSA) [10]. Let $\mathcal{A}[1,N]$ be the suffix array of the collection. We define an array $\Psi[1,N]$ as follows: Let $\mathcal{A}[p] = (n,i,j)$, meaning the cell (i,j) of image n. Then $\Psi[p] = p'$ such that $\mathcal{A}[p'] = (n,i,j+1)$ (or $(n,i,1)$ if image n has just j columns). In addition, let $C[c]$ be the number of cells in all the images whose pixel value is less than c, as before. Finally, we need to sample the images at regular intervals. Let s be a column sampling step. Then we will store, for each cell $I[i, j \cdot s]$ of each n-th image, value $ISA[n,i,j] = p$ such that $\mathcal{A}[p] = (n,i,j \cdot s)$ (the first and last columns of each image must be sampled as well). This is a sampled inverse suffix array. Those chosen p values are also marked in a bitmap $B[1,N]$. We also store a sampled suffix array $SA[rank(B,p)] = (n,i,j)$.

These structures are sufficient to replace the suffix array $\mathcal{A}$ and the images I of the collection. We describe next the relevant operations, and later analyze their space.

Searching for subimage patterns. We simulate the binary searches done over the suffix array, without having it. For this sake, we must be able to read the consecutive L-shaped sequences that correspond to some $\mathcal{A}[p] = (n,i,j)$ given just p, as we do not have $\mathcal{A}$ nor the image. Note that, because all the 2-dimensional suffixes are sorted first by their (i,j) cell, the color of cell (i,j) is the c such that $C[c] < p \leq C[c+1]$, which is simply $rank(C_I, p)$[1]. If we wished to obtain the color of $I[i,j+1]$, $I[i,j+2]$, etc. we simply have to repeat the procedure for $p' = \Psi[p]$, $p'' = \Psi[p']$, and so on. However, in order to read the L-shaped suffix we need also to be able to move from (i,j) to $(i+1,j)$. For this sake, we apply Ψ consecutive (at most s) times, virtually moving rightwards in some image, until we find a marked column that corresponds to p^*, i.e., $B[p^*] = 1$. We then find where we are with $SA[rank(p^*)] = (n,i,j^*)$, so we reached position $(i, j^* \cdot s)$ of image n. With this information we consult $ISA[n,i+1,j^*-1]$, which will give us the q^* such that $\mathcal{A}[q^*] = (n,i+1,(j^*-1)\cdot s)$; note that $(j^*-1)\cdot s \leq j < j^* \cdot s$. From that q^* we apply Ψ successive times (at most s) until reaching the q that corresponds to cell $(i+1,j)$ of the image, as desired. Note that we can use ISA to move directly to row $i+\ell$, by starting from $ISA[n,i+\ell,j^*-1]$.

As we read the L-suffix cells for increasing ℓ in order to carry out a comparison in our binary search for the pattern, we maintain a *front* of the values p_r such that $\mathcal{A}[p_r] = (n,i+r,j+\ell)$. When moving to the next ℓ value, we have just to take Ψ once on each such value, and use the method above to obtain the new $p_{\ell+1}$, first corresponding to cell $(i+\ell+1,j)$, then using Ψ to obtain all the bottom values of the new L-shape, and finally to reestablish the invariant with $\mathcal{A}[p_{\ell+1}] = (i+\ell+1, j+\ell+1)$. Note that the new bottom row of the L-shape is more expensive to obtain than the new rightmost column, hence our particular reading order. Figure 1 (right) illustrates.

Thus, the time cost to carry out the search for an $m \times m$ pattern, using sampling step s, is $O(m(m+s)\log N)$ accesses to Ψ. At the end we know that the occurrences

[1]If σ is small, it might be more practical to represent $C[1,\sigma]$ as a plain array and use binary search on it to obtain c.

of the pattern are pointed from the suffix array area $\mathcal{A}[sp, ep]$, thus there are $ep-sp+1$ occurrences. Note we have not accessed $\mathcal{A}$ nor the images.

Locating the occurrences. As we do not have direct access to $\mathcal{A}$, we cannot just output the occurrence positions $\mathcal{A}[p]$, $sp \leq p \leq ep$, each in constant time. Instead, for each such p, we must apply Ψ consecutive times until reaching a marked column, just as done for searching. If we applied Ψ r times until reaching cell $(n, i, j \cdot s)$, then the occurence is at cell $(i, j \cdot s - r)$ of the n-th image. This takes $O(s)$ accesses to Ψ to report each occurrence.

Displaying a subimage. The search mechanism already shows how can we obtain any subimage starting at $\mathcal{A}[p]$ provided we know p. In order to display an arbitrary subimage of size $m_r \times m_c$ starting at (n, i, j), we must start from some sampled position, that is, from $p' = ISA[n, i, \lfloor j/s \rfloor]$, and obtain from p' the slightly larger subimage of size $m_r \times (m_c + (j \; mod \; s))$ starting at $(i, \lfloor j/s \rfloor \cdot s)$. Hence the time required for displaying is $O(m_r(m_c + s))$ accesses to Ψ.

Space and compression. The sampling mechanism requires about $N+2(N/s)\log N$ bits for B, ISA, and SA, which can be made, for example, $O(N)$ bits by accepting a slowdown factor $s = O(\log N)$. The color table C has already been discussed in Section 3. The most space-consuming structure, however, is Ψ, as in principle it needs $N \log N$ bits, just like $\mathcal{A}$. In [10], it is shown that Ψ can be represented using NH_0 space on a linear text of length N and zero-order entropy H_0. However, the properties that permit proving those space bounds do not hold in the 2-dimensional case: Because L-suffixes are not suffixes of other L-suffixes, there is no guarantee that if (i, j) and (i', j') share the same cell, and the suffix starting at (i, j) is lexicographically smaller than that starting at (i', j'), then the suffix starting at $(i, j + 1)$ will be lexicographically smaller than that starting at $(i', j' + 1)$. We expect, however, that $\Psi[p] - \Psi[p - 1]$ will be a small positive number in a fair amount of cases for two reasons: (1) If the cells at (i, j) and (i', j') are the same but those at $(i, j + 1)$ and $(i', j' + 1)$ are different, then Ψ will be increasing, as the lexicographical comparison between (i, j) and (i', j') is actually decided depending on that between $(i, j+1)$ and $(i', j' + 1)$. (Note that our reading order is also designed for Ψ, which moves rightwards, so the second cell we read in an L-suffix is precisely the next to the right.) (2) Because of spatial homogeneity, the whole L-suffix at (i, j) should be similar to that at $(i, j + 1)$, and those should be close in $\mathcal{A}$. Hence $\Psi[p]$ could be close to p, so consecutive values should not differ by much.

Therefore, we encode Ψ in differential form using an encoding technique that favors small numbers (see Section 5). Absolute samples are inserted every s' positions in Ψ, to permit fast random access. The sampling introduces $N + (N/s') \log N$ additional bits of space (compressible to $(N/s') \log(Ns')$) and $O(s')$ cost to access Ψ. For example, with $s' = \log N$ we have $o(N)$ extra space and $O(\log N)$ slowdown factor.

Note that, obviously, we can also remove some of the least significant planes of the image before indexing it, and hence those will also be removed from the pattern before searching for it. This gives some space gains that are shown in the experiments. Any

Collection	Type	Images	Cells	Plain size	JPG size	Suffix array
Micro	Gray	13	56,583,672	8	2.935	26
Art	RGB	12	6,551,318	24	1.396	23
Maps	RGB	10	4,365,440	24	0.842	23
Fonts	Gray	9	1,119,744	8	5.515	21

Table 1: Characteristics of the four image collections used. We show cell type, number of images, total number of cells, plain raster size (bpc), lossless JPG size (bpc), and size of a plain suffix array (bpc). To enable classical searching one needs to add the plain and suffix array sizes. Collection *Font* is almost black&white.

image shape can be handled, but to search for rectangular patterns we must use the search algorithm as a filter for maximal squares of the pattern, just as in Section 3.

5 Experiments and Discussion

We used four small image collections intended to be representative of different applications: *Micro* (microscopic images from NCI Visuals Online), *Art* (gothic paintings from WebMuseum Paris), *Maps* (ancient maps of Japan from NYPL Digital Gallery), and *Fonts* (some font images from Identifont). Table 1 shows some of their characteristics[2] (bpc stands for bits per cell).

Table 2 shows the space required for the Ψ function in Section 4 using different encoding methods and number of planes (#, 8 is lossless). "Gap" is just the sum of exact bits required by the numbers; it cannot be decoded without help but serves as a lower bound to other gap encodings: Elias γ- and δ-encodings, Rice codes, Dense codes [1], and Fibonacci codes. Huffman coding is excluded as its symbol table is too large to make it competitive. Negative numbers are interleaved with the positive ones. The table also shows the bpc needed by the wavelet tree, wavelet tree over BWT, and bit plane coding (Sections 2 and 3).

To these numbers, we must add the space for colors and samples. That for color is insignificant for gray images. For RGB images, using uncompressed bitmaps, it would add 1.05 bpc, plus about 2 megabytes for the whole collection (the latter must be divided by 8 for each plane we remove in the lossy case). For sampling the suffix array, again without compression on the bitmaps, we have 1.05 bpc extra plus a space/time tradeoff, for example 0.8 bpc if we insert a sample every 64 positions (which yields reasonable performance). To sample the absolute Ψ values we have other 1.05 bpc plus another tradeoff, for example 1 bpc if we sample one out of 32 values of Ψ. Note, on the other hand, that C is not needed on WT and Plane, as no searching is provided, and sampling is needed only for BWT if random access is to be provided.

On RGB images, the compression ratios achieved by wavelet trees is comparable

[2]Sources: *Micro*: `http://visualsonline.cancer.gov`, by browsing topic "Pathology – Electron Microscopy (EM)" and choosing 300 DPI. *Arts:* `http://www.ibiblio.org/wm`, collection "Les très riches heures du Duc de Berry" in subdirectory `rh`. *Maps:* `http://digitalgallery.nypl.org`, browsing term "Maps – Japan". *Fonts:* `http://www.identifont.com/free-fonts.html`, all the fonts in the category "Text serif".

Coll.	#	Ψ encoding (bpc)						Wavelet tree (bpc)		
		Gap	γ-code	δ-code	Rice	Dense	Fib.	WT	BWT	Planes
Micro	8	12.60	24.20	20.43	17.24	17.00	17.97	5.98	5.42	6.45
	4	7.19	13.39	13.03	15.56	10.30	11.49	2.16	1.94	2.54
	2	5.18	9.36	9.76	15.52	7.32	8.78	0.75	0.79	0.86
	1	3.95	6.90	7.61	15.20	5.34	6.67	0.27	0.33	0.27
Art	8	11.91	22.82	19.66	19.70	16.23	17.21	17.00	15.97	20.47
	4	8.91	16.82	15.58	14.84	12.67	13.69	6.04	5.48	8.47
	2	6.40	11.81	11.79	14.56	9.17	10.46	2.09	2.24	2.87
	1	4.77	8.54	9.03	14.57	6.65	8.14	0.78	0.90	0.88
Maps	8	7.31	13.62	12.88	18.01	10.24	11.43	8.47	7.34	14.75
	4	5.77	10.53	10.53	15.68	8.05	9.45	2.43	2.34	4.58
	2	4.68	8.36	8.79	15.18	6.43	7.90	0.99	0.95	1.47
	1	3.52	6.03	6.76	15.32	4.57	5.82	0.28	0.25	0.40
Fonts	8	4.27	7.54	8.01	14.35	5.73	7.12	1.53	1.07	3.99
	4	4.27	7.54	8.01	14.35	5.73	7.12	1.53	1.06	1.99
	2	3.90	6.81	7.54	12.97	5.29	6.60	0.90	0.72	0.95
	1	3.60	6.19	7.08	11.59	4.86	6.11	0.45	0.47	0.45

Table 2: Bits per cell to code the different structures proposed.

to Ψ-based compression, although wavelet trees take much more advantage from lossy compression. On gray levels, wavelet trees are much more efficient, taking even less space than JPG (see Table 1). In all cases, the compression is superior than that of bit plane encoding (sometimes even with Ψ). On the other hand, the wavelet tree on the BWT does not significantly improve compression as it does on text documents (and might do even worse in the lossy case). This is not totally surprising: the wavelet tree alone exploits local homogeneity; whereas after the transform it exploits row-wise repetitiveness. The latter is the key to text compression, whereas the former is much more relevant in image compression. We recall that this BWT offers some subimage searching capabilities (in which case we have to store structure C, as explained). Note that the wavelet tree encoder does not need to know the exact color depth in order to succeed: With *Fonts* converted to 24-bit RGB images, the wavelet tree encoder obtained nearly the same compression as before (1.53 bpc became 1.55 bpc), but bit-plane encoding became much worse (3.99 bpc became 11.97 bpc).

As for Ψ, Dense codes are the best coding choice, as well as easy to program and fast to decode. Overall, our search-capable lossless image representation takes 6-17 bpc (depending on the image compressibility) plus 4-5 bpc for sampling and colors. Although not competitive with lossless JPG, it is remarkable that we support efficient searches using much less spaces than that needed by the suffix array plus the image, and usually less than just the raster image. Our preliminary timings (which are optimistic as we still do not compress Ψ; this could make them around 20 times slower except for construction) on 54MB collection *Micro*, on a commodity desktop PC (2-processor Intel Pentium IV, 3 GHz each, 1 MB cache, 4 GB RAM, Linux, C code, gcc compiler, full optimization) are as follows: we build the index (with a very rudimentary suffix array construction algorithm) in less than 7 sec/MB; we do a binary search for an existing 10×10 pattern in 0.23 msec and 100×100 in 5.13 msec;

we locate each occurrence in 4.67 μsec; we display any 10×10 subimage in 0.05 msec and 100×100 in 2.51 msec. This is for a sampling rate of 64 for the suffix array.

We plan to complete our preliminary prototypes in order to have actual time performance figures, as well as trying out improvements like coding several image rows as a single sequence using wavelet trees (to take advantage of vertical spatial homogeneity, not only horizontal). Besides, we are leaving open some fundamental questions. One is: is it possible to search with variable precision, which can be chosen at search, not indexing, time? Another regards the duality between Ψ and the BWT, well-known in one-dimensional data [8]. It would possibly lead to a self-index with guaranteed worst-case search time just as that based on Ψ, yet based on the wavelet tree, which seems to compress better. However, the duality seems to be absent in the 2-dimensional case. Can it be recovered under a suitable definition of suffix?

References

[1] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.

[2] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report Technical Report 124, Digital Equipment Corporation, 1994.

[3] R. Giancarlo and R. Grossi. *Suffix tree data structures for matrices*, pages 293–340. Oxford University Press, 1997. Chapter 11.

[4] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850, 2003.

[5] D. Kim, Y. Kim, and K. Park. Generalizations of suffix arrays to multi-dimensional matrices. *Theoretical Computer Science*, 302:401–416, 2003.

[6] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. SPIRE'07*, LNCS 4726, pages 229–241, 2007.

[7] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Computing*, 22(5):935–948, 1993.

[8] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[9] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA'02*, pages 233–242, 2002.

[10] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. of Algorithms*, 48(2):294–313, 2003.