

Electronic Alternatives to Micro-Fiches

Shmuel T. Klein¹ and Dana Shapira²

¹ Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

² Dept. of Computer Science, Ariel University, Ariel 40700, Israel
shapird@g.ariel.ac.il

Abstract. Micro-fiches have historically been used for distributing and storing large amounts of data, but data access was slow and cumbersome, and searching for a specific item was often involved with hour-long browsing. Today there are of course electronic alternatives, and we study the possibility of recoding *logically* a printed page, instead of *physically* reducing its size. Such coding allows subsequent lossless compression techniques to be applied. The paper describes the suggested process, including a novel approach to compress numerical data, and presents the details of a typical example.

1 Introduction and Background

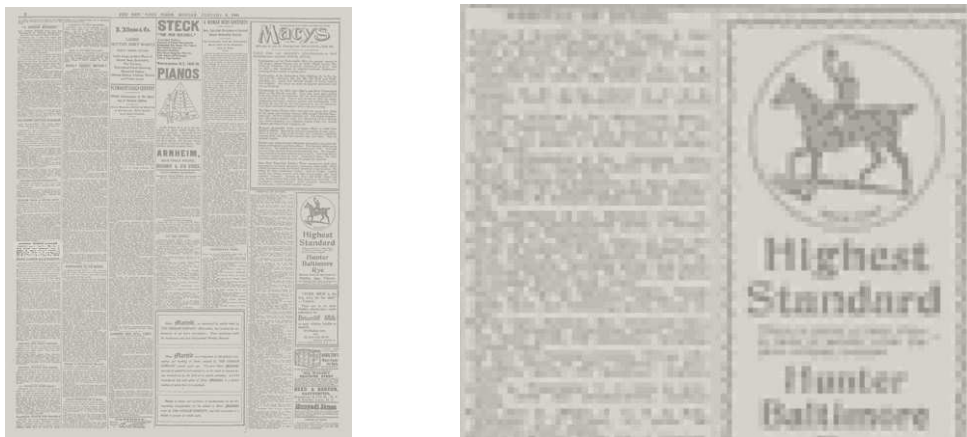
Long before the disk-on-key USB stick became the medium of choice for distributing and storing large amounts of data, and even before their predecessors, the CD-Roms, did micro-fiches serve this purpose. Newspapers and other publishers of primarily textual information used to archive their products by photographically reducing them to small cards, a small box of which could contain thousands of newspaper pages. The principal goal was indeed storage, and retrieval had much lower priority, requiring special micro-fiche viewing devices. Data access was therefore slow and cumbersome, and searching for a specific item often involved hour-long browsing.

The present suggestion deals with an electronic alternative to the classical micro-fiches. Instead of physically reducing the size of a printed page, it could be *recoded* logically. Such coding allows subsequent lossless compression techniques to be applied and moreover, the output may be stored on electronic or magnetic storage, which is by far superior with respect to the retrieval possibilities vs. the old micro-fiches.

Large information retrieval systems, in which the full texts are stored electronically and which allow direct access to any word or phrase, have been developed since the late sixties. Today, of course, practically all available data are produced electronically, and access is thereby facilitated. The exact layout and pagination is mostly not considered as an important factor, and the same information is often displayed differently on different devices, be it a small portable phone or a large computer screen. Our main concern here, however, is with retrieval systems in which the original layout of the text *is* of importance, as in the following applications:

- *Historical editions.* The study of historical texts may be of interest to humanists, historians, Bible experts and other scholars of classical texts. There are often well accepted standardized and widely referenced editions, which ought to be stored in their original form and pagination. A case in point is the Babylonian Talmud, whose 6000 pages of the Vilna Edition have become so widespread, that references to the Talmud are generally given by page number. Other examples could be original manuscripts of famous authors, poets and composers, and Holy Texts for several religions.

- *Journal pages.* Today, newspapers are designed, prepared and printed with the help of computers, but older journals have not been stored electronically. Nonetheless, reporters, anthropologists, lawyers and many others are often concerned with older editions, and much can be learned from the co-occurrence of certain data items on the same page or in proximity. As an example, consider in Figure 1 a page from the *New York Times* archive from 1900, and an enlarged detail of it; as can be seen, the text is not readable.
- *Technical design.* Manufacturers of complex machines such as cars, aircrafts, etc., used to store repair instructions in thousands of detailed design sheets. These included drawings and texts.



(a) NYT page from 1900

(b) Enlarged detail

Figure 1: Example of archived NYT page

2 Related Work

The usual approach to store such data electronically, was to scan the printed page as if it were a picture, and store it as a raster file. This always resulted in a dramatic reduction in picture quality, depending on the resolution chosen for the scanning process. Recently, scanned images, even in color, are of much better quality, but at a price: a color pixel may be coded in three bytes, covering a possible range of $2^{24} = 16$ million color shades; images are scanned at high resolutions, typically 600 dots per inch (dpi) or higher (i.e., 360,000 pixels per square inch), so that the scanned image of a standard $8.5'' \times 11''$ sheet would occupy about 96 MB! Even if modern compression methods may reduce this to merely 1% (using some standard like JPEG [28], or similar methods, all of which are *lossy*, i.e., discard a part of the information), we are still left with about 1 MB for a single sheet, which is much too large for many practical applications.

The state of the art today is the *Portable Document Format*, known by its acronym PDF, which is widely used. It includes many of the features we are interested in, but using PDF may be an overkill, and a typical page at reasonable resolution and containing a large amount of text may span about 3MB and even more per page. The use of the PDF format has recently been criticized in [7].

Our main concern here is with primarily textual data. On the one hand, this simplifies the problem, as there is no need for colors to be coded. A pixel can thus be stored in a single byte instead of three, giving a range of 256 grey levels, or ultimately even in a single bit, for strictly black and white text images. A standard 8.5" \times 11" sheet, still at 600 dpi, would thus need about 4 MB, or about 1 MB at the lower resolution of 300 dpi. On the other hand, lossy compression techniques are no more adequate, and the standard lossless compression methods, when applied to data of this type, rarely reduce their size to less than about 30%. We are thus still left with about 0.3 MB per page. In addition, a text image at 300 dpi is not always readable if small fonts are used.

Current text compression methods achieve a reduction of up to 70%, e.g., [32], but do generally not care about the original page layout. Keeping both the text in textual form as well as a scanned image is generally considered as redundant and is not supported by many available systems.

Many text compression methods have been published in the literature and as patents. Most fast on-the-fly methods are based on algorithms by Lempel and Ziv [33, 34]. A partial list of patents based on these includes [9, 11, 29, 30] and many others. A major problem of these algorithms is to find an efficient way to locate a recurring string in the text. Efficient solutions have been given in [8, 20]. When the speed of the encoding process is not critical, as in applications addressed in the current method, in which the compression may be done off-line in a preprocessing stage, better compression can be achieved by what is called a recompression method, as in [18]. As an alternative, one may use a compression method that does not require any decompression at all and allows the search for a pattern directly in the compressed text. This paradigm is known as *compressed pattern matching* and has been thoroughly investigated, e.g., in [22].

The compression of layouts has not been addressed as a topic in its own right. What might come closest is the compression of structured data such as lists of numbers or bit-vectors, as for example in [1, 6]. The system suggested herein does not claim to invent a new special purpose compression method, to be applied to images of textual data. Rather, it suggests a new approach, encompassing methods known as prior art, but unifying several such methods into a new coherent system.

As to related work, a similar system to the one suggested below, albeit with another application area, was suggested by Wong and Chan [31]. They teach how to store computer display information, which may contain both text and graphics. However, since the information there is supposed to be given already electronically, the basic assumptions are different from ours. In particular, [31] does not address the problem of scanning a page given in printed format, storing its layout and *simulating* its original form.

Reference to the reconstruction of layouts can be found in Morgan [25], who deals with generic forms. He defines the layout as a set of rectangles, similarly to what we do, but it is different in that it allows dynamic alterations of the general layout; for example, columns may grow or shrink, depending on the text they should contain, whereas for the applications we have in mind, the given page is supposed to be fixed and imposed as a part of the input, and its exact original form has to be approximated as closely as possible.

A number of publications that mainly appeared as patents deal with creating, for a given page, the layout which is best under some criterion, out of a set of possible layouts. What these methods have in common with our suggestion is that they scan the page and convert it into a sequence of images and text parts. But the purpose is not to reconstruct a page similar to the original, but to change it and produce some optimal tiling. Some of these works can be found, e.g., in the patents Hart et al. [12], Hayashi and Saito [13], Mason [24] and Kataoka et al. [15].

Many works, among others, Hernandez et al. [14], Fukui et al. [10], Borgendale and Dobkin [3], deal with text editors, in which the page can be changed. They include the treatment of graphics, but are not aimed at displaying a fixed printed page. Finally, Knuth's [23] \TeX typesetting software (that has been used for the preparation of this paper) produces a printed page in a Device Independent file format that shows many similarities to the proposed system, but lacks the requirement of producing a layout closely simulating a given original.

3 General Description of the Suggested System

The main idea of our work is a shift of focus: instead of taking high quality pictures of a text page and approximating the original by trying to keep the loss of information due to the application of some lossy compression technique to a minimum, we try to approximate the building blocks at the lowest level, i.e., the images of the alphabet characters in various fonts, record the general layout of the page to be printed, and thereby *simulate* the image of the text page. The following advantages may thus be achieved:

- Even if the simulation is not an absolutely true copy of the original, the difference might be very hard to notice, if at all. Anyway, a scanned image is not a true replica either, because of the limited resolution.
- The picture quality will be by far superior to that of a scanned image.
- The necessary space will at the same time be considerably reduced, actually so far as to enable to storage of millions of images on a single USB stick.
- Contrarily to a scanned image, the simulated one can be the basis of a location sensitive software package, allowing access to and processing of the *words* on the page, not just their image. This opens new opportunities, such as grammatical analysis, automatic translation, various Hypertext functions, etc.

In a first phase, the underlying text has to be stored as such, rather than as a picture. For many applications, this incurs no additional overhead at all, since the text is anyway available. This is true for literary works like those collected in the *Bibliothèque de la Pléiade* [26], which is a part of the French full text retrieval system known as *Trésor de la Langue Française*, described in [4]; other examples might be famous texts such as the Bible or the Talmud. In case the text has not been recorded previously, this may be done with the help of optical character recognition (OCR) packages, which nowadays operate with very high accuracy, especially if printed and not handwritten pages are scanned. The overhead, even in this case, will not be significant. To return to the above example of a standard 8.5" \times 11" sheet, it rarely contains more than about 1000 words or 5000 characters. Using even simple compression methods, this could be stored in less than 3K, and more typically, for less dense

pages and with better compression, in about 1K — just 0.1% of our earlier estimate of the size of a stored image of such a sheet.

The second phase consists of recording the general layout of the page. Practically all printed pages may be decomposed into a set of non-overlapping rectangles. For newspapers these consist usually of several columns. For various literary works, the layout may be more sophisticated, with the main text in several rectangles in the center, surrounded by rectangles of various shapes, holding commentaries written in smaller fonts. This layout can be stored effectively, since all we need for exactly locating a rectangle is the coordinates of two of its diagonally opposed corners, as in [25], or equivalently the coordinates of one of the corners and the height and width of the rectangle, as in [12]; in any case, two number pairs suffice. If two bytes are used for each of these numbers, a granularity of 10^{-3} inches may be achieved, by far high enough so that any deviation below it may not be detected by the human eye. Even if up to a hundred rectangles have to be stored for a single page (and the actual number is usually much smaller), the storage requirements for the layout are still below 1K.

The next step consists of “reconstructing” the original form of the printed page, using the text and the layout. This might be complicated for handwritten manuscripts, for which the form of the lines, as well as the spaces between lines, words and characters, may fluctuate. But for printed pages, the inter-line space is usually constant, and within a line, the words are generally spread out as uniformly as possible.

The amount of space between characters within a word, as well as the height of the characters influencing the space between the lines, are a part of the font specification [23]. In older printed texts, the typesetter may have physically arranged the letters and words and adjusted the spaces manually, yielding unavoidable fluctuations. One of the assumptions of this work is, that it is not worth investing the effort (and the additional bits) allowing the faithful reconstruction of these rather arbitrary deviations from equally distributed inter-word spacing. So even if in the simulated page, the spaces are not exactly matching those of the original one, the differences will mostly remain undetected, but even when they are visible, no important data has been lost.

All one needs to store to enable the reconstruction of the page is therefore: for each rectangle, a pointer to the starting word in the text, the number of lines within the rectangle, and for each line, the number of words it contains. Such lists of numbers may be stored very efficiently (as e.g., in [6]), and the amount of space needed for it per page will generally be of the order of 1–2K.

In a final touch, thought has also to be given to pictorial and any other data that is not textual in nature. Though we assume here that the bulk of the information is text, there might be small ornaments, pictures, drawings, etc., or even characters in special fonts (like an overdimensional starting character of a chapter), that have to be embedded. These will be kept in image form, yet it will not have the disadvantages of storing the full page as a raster file:

1. for mainly textual data, for which this method is intended, the overall proportion of pictorial data should not exceed a few percent; furthermore, lossless techniques can subsequently be applied.
2. for the data that will be kept in image form, standard lossy compression techniques are suitable, since the images will not represent small type characters.

The main purpose of displaying a page is to let the user browse through it. This is the only function supported by conventional systems based on micro-fiches (where this

requires special hardware) or on the representation of the text page as an image. But here, there is also a possibility of using sophisticated search mechanisms, as supported by text retrieval systems. One can therefore easily look for words or phrases, using well-known pattern matching techniques such as [5], or use more advanced tools, including grammatical variants and metrical constraints, requiring dictionaries [21], concordances or signature files [16, Section 8.1.1], that can be compressed by various techniques [17, Section 6.5].

In addition, the system allows also interaction with the page. With the help of a pointing device, which could be a mouse, a lightpen, or anything else supported by touch-sensitive screens like, ultimately, our bare fingers, certain parts of the text image can be “marked” as selected. Given that the system has created the display and the layout, basing itself on the text, which is stored separately, it is possible to evaluate exactly the characters displayed in the region on the screen that has been pointed to. In other words, the system can “understand” which words or signs have been selected. One can therefore enlarge selected parts of the image, as we are all used by handling our smartphones, zooming in and out on request, which is especially useful in the presence of small type fonts.

A zooming function may also be available if the full page has been stored as an image. But in that case, enlarging the display cannot add information that has not been stored beforehand, so that the resolution at which the page has originally been scanned essentially limits the usefulness of such enlargements. If certain characters of the scanned font are of the size of a small number of pixels, they might be impossible to read regardless of the enlargement used, as can be seen in Figure 1(b).

Some of the possible interactions may include, among others: using dictionaries for the automatic translation of selected words into another language; for proper nouns, automatic access to historical, geographical or biographical data; for any word or even character string, giving all possibilities to interpret it (very important for languages with high frequency of homonyms), adding global statistical information, such as the number of times the string occurs in this and other texts; if a full phrase, rather than a single word is selected, access to parsers, taggers or other data of interest to linguists.

4 Compression methods

One of the possible partitions of the vast area of Data Compression is by the nature of the data to be handled. A large amount of the data considered as important enough to be stored is *textual*, generally written in some natural language. Another significant part of the data is pictorial: *images* of all kinds and sizes, often collected into sequences forming videos and films. We wish to concentrate here on a third kind, which is *numerical* data. This differs from the textual form, of which it is a subset, in several ways:

1. There is no inherent redundancy, which is exploited by many of the text compression techniques;
2. The underlying alphabet, generally consisting of just 2, 10 or 16 digits, plus some whitespace, is much smaller than for text, especially if the elements to be encoded are not restricted to be just simple characters, ASCII for example, but may be chosen as the different words in a textual database. If numbers, rather than digits, are to be encoded, the alphabet is potentially infinite;

3. Contrarily to text, errors will be impossible, or much harder, to detect;
4. Depending on the origin of the data, lossless compression may be required, or various degrees of lossy methods could be tolerated.

More specifically, most of the data that needs to be encoded for the layout consists of the description of rectangular bounding boxes into which the various texts have to be placed. Practically all of these rectangles have their edges parallel to the borders, so that two points in the plane, corresponding to diagonally opposed corners, suffice to obtain a well-defined rectangle. The rare cases of rectangles or other forms deviating from this assumption about the orientation are dealt with as *additional images* in the third phase of the suggested system. Figure 2 shows a small sample of the 4-tuples defining the rectangles on our test file to be described below. The measurements are given in points (pt), where 1pt is $\frac{1}{72}$ of an inch, about a third of a millimeter or more precisely 0.353mm. Obviously, the precision given by the software of up to 15 digits after the decimal point is a senseless overkill, measuring distances of the order of attometers (a billionth of a nanometer).

107.94351196289062	28.557645797729492	527.6393432617188	581.769775390625
107.93960571289062	58.182525634765625	527.402099609375	717.7293701171875
255.5399169921875	125.14797973632812	379.80010986328125	193.12799072265625
278.819091796875	133.1363067626953	356.4554138133594	167.07965087890625
232.61822509765625	200.4618682861328	527.4641723632812	615.6199951171875

Figure 2: Sample of 4-tuples defining rectangles

We suggest the following compression method that is especially adapted to deal with such layout data. It could be qualified as being *semi-lossless*, as on the one hand, it is lossy by not restoring accurately the original file bit per bit, yet it is lossless in the sense that no useful information has been removed. This new paradigm could find applications beyond those discussed here in the context of numerical data. In an application to X-rays, for instance, one could tolerate a certain degree of lossy compression, as long as the reconstructed image could be judged as “diagnostically equivalent” to the original.

There are several freely available tools helping to extract information about bounding boxes from PDF pages, and we used PyMuPDF [27] on our test data. Every quadruple of numbers is given in decimal form in about 70–80 bytes, representing four 64-bit floating point numbers.

We shall store each number x by encoding separately the integer and fractional parts, I_x and F_x , respectively. Using just 4 bits for F_x partitions the unit into 16 equi-sized parts, so by choosing the integer k , $0 \leq k < 16$, such that $\frac{k}{16}$ is closest to F_x , the potential error is bounded by $\frac{1}{32}$ pt, about one hundredth of a millimeter. This is hardly noticeable to human eyes. Taking the last number in Figure 2 as example, $F_x = 0.619995 \dots$ and will be represented by the integer $k = 10$, standing for $\frac{10}{16} = 0.625$ pt; the error is less than 0.002mm.

As to the integer part I_x , we first note that for the application at hand, the distribution of the occurring values is not uniform. This is due to the fact that the rectangles are not placed arbitrarily within a page, but that the typesetter tried to impose some alignment, so that certain values of the x part in the (x, y) coordinates, indicating the offset from the left edge of the page, show a tendency to re-occur. Of course, the high precision floats will never be matching exactly, but when quantization to the integer part level is used, repetitions will appear. This suggests that applying Huffman coding to the set of possible values might yield some savings.

Instead of preparing codewords for all the possible values (about 800 for a page of height 11”), requiring 10 bits, we evaluated the average length of a Huffman codeword for each of the four components of the two pairs (x_0, y_0) and (x_1, y_1) separately. On our test data, this yielded averages of 3.93, 5.49, 4.59 and 5.92 bits, respectively. As can be seen, the averages for the x components are smaller than for the y components, which can be explained by the fact that there is apparently more regularity in the *width* of the rectangles, depending on the horizontal offset from the left edge of the page, than in their *height*, measured as vertical offset from the top edge. Indeed, many different pages will display rectangles with very similar widths, but the heights are adjusted according to the desired balance between the amounts of the various texts that have to be included.

In a final touch, we mentioned already that a rectangle can be encoded equivalently by (x_0, y_0) and $(width, height)$, instead of (x_0, y_0) and (x_1, y_1) , where $width = x_1 - x_0$ and $height = y_1 - y_0$. This could be advantageous, because width and height are translation invariant, so one may expect more repetitions, and thus a more skewed distribution, implying better performance of the Huffman code. On our test data, the average number of bits for *height* and *width* are 5.82 and 4.16 bits, instead of 5.92 and 4.59 for y_1 and x_1 , a further improvement of 2–10%. The expected total number of bits required for a quadruple is thus 36.4, or 9.1 bits per number.

Figure 3 gives a schematical view of the compressed file, in which variable length color-coded Huffman codewords, standing for the integer parts of the four different components, alternate with lossy fixed length encodings of the fractional parts. Decoding is possible because each of the Huffman codes is prefix-free, and so is therefore also a sequence of alternating elements from different Huffman codes, even if interspersed with fixed-length codewords. A detailed example for this compression method is given below.



Figure 3: Schematic view of the compressed form of the coordinate quadruples

5 Typical Examples

A first example of the usefulness of the new system can be found in Figure 4, displaying the first few verses of the famous fable *Le Corbeau et le Renard* about the Raven and the Fox by *Jean de La Fontaine* (1621–1695). The left part is from a historical edition printed in 1890, and spans more than 303K in its original raster form of 554×187 pixels, which is reduced to about 31.3K by JPEG. Though readable, the quality is very low. The version on the right can be magnified as much as desired, is almost visually identical to the original, yet requires, even in uncompressed form, less than 0.4K bytes. This evaluation includes the \LaTeX formatting commands, but excludes the definition of the characters in the various fonts, which are considered negligible overhead when amortized over a large enough text corpus.

The following additional example has already been mentioned in the introduction and will be brought here in more detail. The Babylonian Talmud is one of the major works of Jewish culture. It includes texts relating to all aspects of Jewish life, and describes the discussions about various possible interpretations of the Bible, held by more than three thousand scholars, mainly from Babylon and Israel, who lived

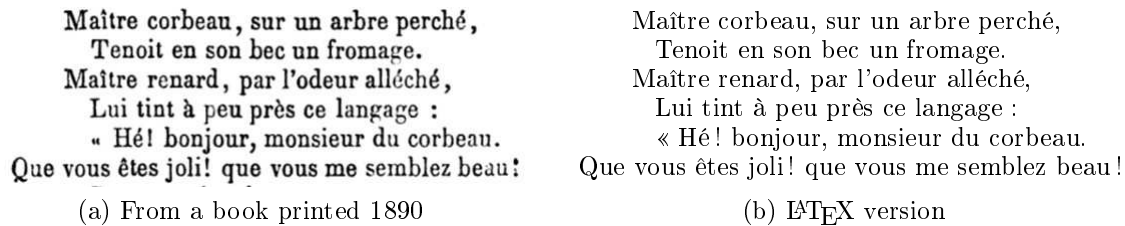


Figure 4: Example of a Fable by La Fontaine

between the first and fifth centuries. The Talmud consists of 36 tractates, and was first printed by Daniel Bomberg in Venice in 1520–23. This edition had a total of 5894 pages, and most posterior editions kept this partition. The talmudic text consists of about 2.5 million words, or about 13 MB.

Figure 5(a) displays a typical page (71b of tractate *Kiddushin*). The text of the Talmud can be found in the center, surrounded by various commentaries: the most important ones are those put immediately adjacent: on the left the commentary by Rashi (Rabbi Shlomo Yitzhaki, 1040–1105), and on the right by the Tossafists (13th century). More annotations are further away from the center, and appear generally in smaller print. The top line is a running title, giving the name of the tractate, and name and number of the current chapter. It is important to note that this layout, corresponding to the edition printed in Vilna by the Romm family in 1880–86, has been practically universally accepted, and thousands of Talmud scholars all over the world read, study and refer to these pages in exactly this form. It is thus not rare that one remembers certain passages not so much by their exact word sequence, but by their location (e.g., close to the upper right corner) on the page. Hence the need to store not only the text, but to store it specifically in this form.

Figure 5(b) brings the general layout of the page of Figure 5(a). Note that a small number of rectangles is sufficient to cover the bulk of the text. The small squares missing in the upper right corner and closely below in the layout correspond to words written in the original page in a special larger font. Note also that not all the text close to the borders has been put into rectangles in this example, to emphasize the possibility of leaving even text parts, like short comments, as additional pictures. In the given example, only texts spanning two or more full lines are recorded. Figure 5(c) then gives whatever has not been covered. This includes here the title line, small geometric ornaments, words in special fonts and shorter comments, but could be significantly reduced by also storing the rest of the text, and leaving in image form only non-textual data, which is rare in the Talmud.

In addition, cross-references are extremely frequent in the Talmud, which therefore is ideally suited to become a Hypertext environment [2]. Having this possibility directly on the displayed page that looks identical to the page the user is familiar with from his books, is a great advantage over other display systems without Hypertext support.

We now proceed to a rough calculation of the required space by the different methods. Even if we assume that one can store a typical page of the Talmud in 200K in compressed form (which corresponds to low resolution and reduced quality), one would need a total of 1.2 GB for the entire Talmud. A simple scanning application like those available on our mobile phones produces low-quality PDF images of about 0.5 MB per page, totaling 3GB for the full text. Moreover, this would only include the pictures of the pages, no text, dictionary or concordance. On the other hand,

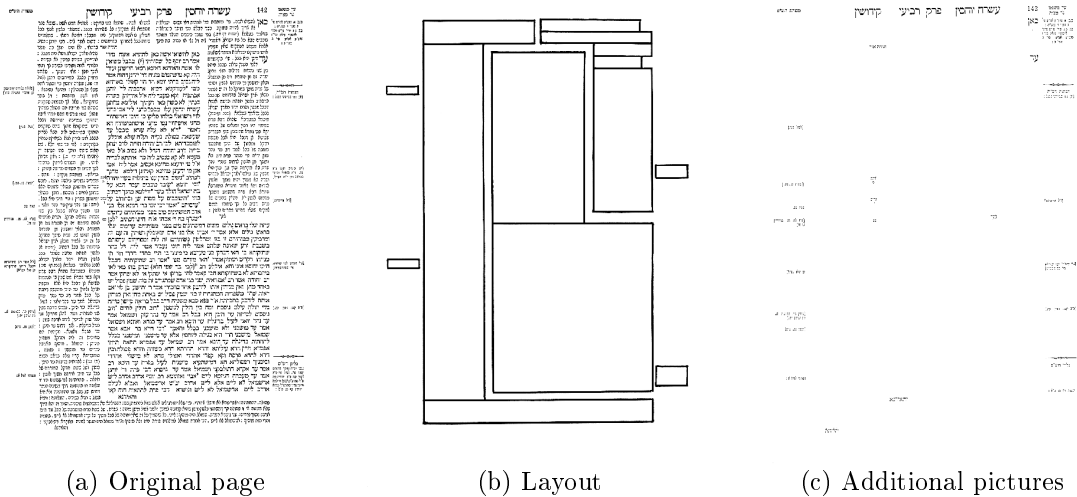


Figure 5: Example of typical text page

the text alone of the full Talmud, together with the commentaries by Rashi and the Tossafists, span less than 50 MB, and can easily be compressed to 25 MB. To this we have to add the exact pictures of all the characters in all the fonts that are used in the text, but this is done only once for the full Talmud, and will overall occupy less than 1 MB. If we count 1K per page of layout (6 MB in total), another 1K for storing the number of words in each line (6 MB in total) and even 4K for the compressed form of the additional pictures per page (24 MB in total), we end up with about 60 MB, roughly 2% of the low resolution parallel in PDF. One can thus add other, related, texts, and all the auxiliary files mentioned above for gaining access to the text with the help of an information retrieval system [19].

To show an example of the compression of the layout data, we took the ten first pages of the Talmud, Tractate *Brachot*, pages 2a to 6b, and applied PyMuPDF, which extracted 135 bounding boxes, each defined by 4 high precision floating point numbers, (x_0, y_0) and (x_1, y_1) , the first five of which are depicted in Figure 2. As explained above, the actual encoding is performed on

$$(x_1 - x_0, y_1 - y_0) = (419.69583129882818, 553.212129592895508),$$

instead of (x_1, y_1) . The four integer parts are therefore 107, 28, 419 and 553, and the corresponding fractional parts, starting with 0.9435, 0.5567, 0.6958 and 0.2121 are encoded by the 4-bit fixed length standard binary representations of the integers 15, 9, 11 and 3, respectively.

Four different Huffman trees are then constructed, one for each of the distributions of the possible values of x_0 , y_0 , $x_1 - x_0$ and $y_1 - y_0$. For our example, 135 values with repetitions are sampled for each of the trees, yielding the trees shown in Table 1. We here use the notation $\langle n_1, n_2, \dots, n_k \rangle$, known as a *quantized source*, where n_i is the number of codewords of length i in a given Huffman tree. If *canonical* Huffman codes are used, in which the depths of the leaves are non-decreasing from left to right, the quantized source suffices to uniquely define the layout of the Huffman tree and accordingly, the corresponding codewords, after having ordered the elements by non-increasing frequencies. Once we know the optimal length l_i of the i -th codeword, the codeword itself can be defined as the first l_i bits following the binary point in the infinite expansion of the binary number $\sum_{j=1}^{i-1} 2^{-l_j}$.

Table 1 shows, for each distribution, the corresponding quantized source, the integer to be encoded, its rank in the ordered list of different elements, the depth of the corresponding leaf in the Huffman tree, which is the length of the codeword used to encode the element, and finally the codeword itself.

distribution	quantized source	integer	rank	depth	codeword
x_0	$\langle 0, 2, 0, 2, 6, 9, 6 \rangle$	107	8	5	10111
y_0	$\langle 0, 0, 0, 4, 10, 8, 40 \rangle$	28	5	5	01000
$x_1 - x_0$	$\langle 0, 0, 4, 4, 2, 5, 14 \rangle$	419	4	3	011
$y_1 - y_0$	$\langle 0, 0, 0, 4, 4, 6, 68 \rangle$	553	64	7	1101101

Table 1: *Huffman trees and encoding of the integer parts.*

Figure 6 summarizes the example, showing the encoding of the first 4-tuple and using a color-code to match that of Figure 3. The decimal value of the grey binary fixed-length parts appear above the encoding, the Huffman encoded values are listed below their variable-length blue, green, pink or yellow codewords. In this example, four floating point numbers are encoded by $5 + 5 + 3 + 7 + 4 \times 4 = 36$ bits, exactly 9 bits per number. Extrapolating from this (not necessarily representative) example, we get an estimate of less than 3MB for the layout data of the entire Talmud.

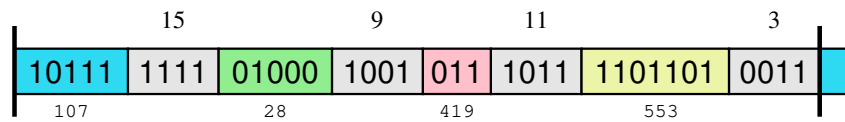


Figure 6: *Encoding of the first 4-tuple*

The Talmud is today commercially available in picture form, with varying degrees of accessibility. The suggested system would make the full collection available on significantly reduced storage space, in much higher quality, and allow many functions to be performed that are not supported by picture-based systems. There are also Information Retrieval Systems on the market that support full-text access but lack the ability of showing the retrieved text in its customary form. By increasing the total size of the system only slightly, the suggested method could also add this desirable feature.

Summarizing, the contribution of this paper is the suggestion of replacing the modern alternatives to micro-fiches, in which we assume that not only the content, but also the exact original layout is of importance, by a system that *simulates* the output page rather than reproducing it as a picture. The loss should be hardly noticeable by the bare eye, but the advantages in terms of compression and possible further processing may be significant. In addition, we suggested also a novel semi-lossless compression scheme for numerical data appearing in the description of layouts.

References

1. ASRAF S., GROSS Y., KLEIN S.T., REVIVO R., SHAPIRA D., New compression schemes for natural number sequences, *Discrete Applied Mathematics*, **327** (2023) 18–27.

2. AVIAD A., HyperTalmud: a hypertext system for the Babylonian Talmud and its commentaries, MSc. Thesis, Department of Mathematics and Computer Science, Bar-Ilan University, Israel (1993).
3. BORGENDALE K.W., DOBKIN D.J., System and method for editing a structured document to preserve the intended appearance of document elements, U.S. Patent 5,276,793, Jan. 4, 1994.
4. BOOKSTEIN A., KLEIN S.T., ZIFF D.A., A Systematic Approach to Compressing a Full-Text Retrieval System, *Information Processing and Management* **28**(6) (1992) 795–806.
5. BOYER R.S., MOORE J.S., A fast string searching algorithm, *Comm. ACM* **20** (1977) 762–772.
6. CHOUÉKA Y., FRAENKEL A.S., KLEIN S.T., Compression of Concordances in Full-Text Retrieval Systems, *Proc. 11-th ACM-SIGIR Conf.*, Grenoble (1988) 597–612.
7. ERTL, A.M., What is the PDF format good for?, Vienna University of Technology, <https://www.complang.tuwien.ac.at/anton/why-not-pdf.html>, 2024.
8. FIALA E.R., GREENE D.H., Data Compression with Finite Windows, *Communications of the ACM* **32**(4) (1989) 490–505.
9. FRUCHTMAN A., GROSS Y., SHAPIRA D., KLEIN S.T., Systems and methods of data compression, U.S. Patent 11,722,148, Aug. 8, 2023.
10. FUKUI M., IWAI I., DOI M., TAKEBAYASHI Y., Method and apparatus for editing documents, U.S. Patent 5,179,650, Jan. 12, 1993.
11. GIBSON D.K., GRAYBILL M.D., Apparatus and method for very high data rate compression incorporating lossless data compression and expansion utilizing a hashing technique, U.S. Patent 5,049,881, Sep. 17, 1991.
12. HART P.E., CHILTON J.K., PEAIRS M., Document layout using tiling, U.S. Patent 5,553,217, Sep. 3, 1996.
13. HAYASHI N., SAITO K., Document layout processing method and device for carrying out the same, U.S. Patent 5,379,373, Jan. 3, 1995.
14. HERNANDEZ I.H., BARKER B.A., MACHART B.H., Flow attribute for text objects, U.S. Patent 4,723,209, Feb. 2, 1988.
15. KATAOKA M., USAMI Y., YAMADA M., HARADA K., HORI C., Layout displaying apparatus for a word processor, U.S. Patent 5,287,445, Feb. 15, 1994.
16. KLEIN S.T., *Basic Concepts in Data Structures*, Cambridge University Press, Cambridge, 2016.
17. KLEIN S.T., *Basic Concepts in Algorithms*, World Scientific Publishers, New Jersey, 2021.
18. KLEIN S.T., Efficient Optimal Recompression, *The Computer Journal* **40**(2/3) (1997) 117–126. Efficient optimal data recompression method and apparatus, U.S. Patent 5,392,036, Feb. 21, 1995.
19. KLEIN S.T., BOOKSTEIN A., DEERWESTER S., Storing Text Retrieval Systems on CD-ROM: Compression and Encryption Considerations, *ACM Trans. on Information Systems* **7** (1989) 230–245.
20. KLEIN S.T., SHAPIRA D., A New Compression Method for Compressed Matching, *Proc. Data Compression Conference (DCC)* (2000) 400–409.
21. KLEIN S.T., SHAPIRA D., Compressed Matching in Dictionaries, *Algorithms*, **4**(1) (2011) 61–74.
22. KLEIN S.T., SHAPIRA D., Pattern matching in Huffman encoded texts, *Information processing & management*, **41**(4) (2005) 829–841.
23. KNUTH D.E., *TEX and METAFONT: New directions in typesetting*, American Mathematical Society, Providence, RI (1979).
24. MASON C.A., Document processing method and system, U.S. Patent 5,214,755, May 25, 1993.
25. MORGAN M.W., Method and apparatus for representing bordered areas of a generic form with records, U.S. Patent 5,208,906, May 4, 1993.
26. https://en.wikipedia.org/wiki/Bibliothèque_de_la_Pléiade
27. <https://pymupdf.readthedocs.io/en/latest/>
28. WALLACE G.K., The JPEG still picture compression standard, *Comm. of the ACM* **34**,4 (April 1991) 31–44.
29. WELCH T.A., A Technique for High-Performance Data Compression, *Computer* **17**(6) (1984) 8–19.
30. WHITING D.L., GEORGE G.A., IVEY G.E., Data compression apparatus and method, U.S. Patent 5,016,009, May 14, 1991.
31. WONG H.H., CHAN V.W., Compact memory for mixed text in graphics, U.S. Patent 5,457,776, Oct. 10, 1995.

32. ZAVADSKYI I., KLEIN S.T., SHAPIRA D., Word-Based Forward Coding, *Proc. Data Compression Conf. (DCC)*, Snowbird, Utah (2024) 351–361.
33. ZIV J., LEMPEL A., A universal algorithm for sequential data compression, *IEEE Trans. on Inf. Th.* **IT-23** (1977) 337-343.
34. ZIV J., LEMPEL A., Compression of individual sequences via variable rate coding, *IEEE Trans. on Inf. Th.* **IT-24** (1978) 530–536.