Pattern-based compression of text images

Andrei Broder
Digital Syst. Res. Center
130 Lytton Avenue
Palo Alto, CA 94301
broder@pa.dec.com

Michael Mitzenmacher*
Dept. of Computer Science
UC Berkeley
Berkeley, CA 94720
mitzen@cs.berkeley.edu

Abstract

We suggest a novel approach for compressing images of text documents based on building up a simple derived font from patterns in the image, and present the results of a prototype implementation based on our approach. Our prototype achieves better compression than most alternative systems, and the decompression time appears substantially shorter than other methods with the same compression rate. The method has other advantages, such as a straightforward extension to a lossy scheme that allows one to control the lossiness introduced in a well-defined manner. We believe our approach will be applicable in other domains as well.

1 Introduction

We consider a compression scheme, GRID, designed for gray-scale images of documents consisting primarily of text. The GRID scheme has significant advantages over other methods in this domain, especially in terms of decompression time and extensions to lossy models. Moreover, the method underlying GRID is quite general and appears applicable to other similar problems.

GRID was developed as part of an ongoing research project at Digital Systems Research Center to make on-line reading from a computer screen an acceptable alternative to reading paper copy. The documents under consideration vary in size from a page to books of hundreds of pages. Textual documents are stored as gray pixel maps, with two bits per pixel (4 shades of gray) to enhance legibility via anti-aliasing. The dramatic improvement in reading quality by using gray-scale is an important feature of the system. Typically a 300 dpi black and white image of a page of text is displayed as 100 dpi gray scale. Compressing these pixel maps (or *pixmaps*) for storage and transmission is essential. While compression can be relatively slow, decompression must be very fast – to the user, turning a page should appear instantaneous.

Previous compression techniques for gray-scale or bilevel images of text generally come in two flavors: pixel-based or character based. Pixel-based schemes consider

*Supported by the Office of Naval Research. This work was done during an internship at Digital Systems Research Center.

and encode each pixel individually. Examples include the international standard JBIG [3], FELICS [2], and mgbilevel, part of the mg system [6, 7]. Character-based schemes attempt to derive the original text from the document by inferring or keeping information regarding character boundaries and using derived or provided font information. The mg system also includes such a scheme, called mgtic [6, 7], based on the work presented in [5].

We suggest a novel compression scheme based neither on pixels nor characters but on patterns determined from simple groupings of pixels. Our experiments indicate that our method achieves better compression than most alternative schemes, and the decompression time is substantially shorter than the time required by schemes achieving a similar compression rate. Our scheme extends naturally to a lossy version where the discrepancies introduced can be controlled in a straightforward way. In practice, the lossy scheme often corrects artifacts found in scanned documents, and it can be modified to yield a lossless scheme that improves compression at the expense of primarily compression time.

In Section 2, we will present the basic GRID algorithm, and explore some additions to the basic scheme, such as lossy encoding, that can improve performance. In Section 3, we shall present some results from our prototype implementation, including a comparison with other available methods.

2 The GRID algorithm

The GRID scheme is based on the following paradigm for encoding:

1. Break up the document pixmap into simple pieces
2. Get frequency counts on the patterns corresponding to the pieces
3. Use the frequency counts to construct a Huffman code table for the patterns
4. Encode the document using the Huffman table

The encoding thus consists of a Huffman table corresponding to the patterns, together with the encoded document.

The motivation for this scheme is similar to that of the character-based schemes. We assume that the uncompressed image contains frequently appearing patterns, determined by the underlying font. Rather than attempt to determine the font, however, we build up our own font in a first pass; we call this font the *derived font*. We keep our derived font simple to allow fast decompression and reasonably fast compression. We must also strive to keep the derived font relatively small, as the time the de-compressor spends reconstructing the derived font corresponds to latency before the first page can be decompressed, and the size of the derived font is directly related to the amount of memory utilized by GRID.

2.1 Dividing up the image

The first step is to break up the pixmap, which can be pictured as a large two-dimensional array of pixels, into simple pieces. While the size and shape of these pieces should depend on the font size used in the underlying document, in practice, for 100 dpi gray scale images, we found that splitting the pixmap into 4 by 4 pixel blocks proved most effective for standard font sizes– conveniently, this block size corresponds to 32 bits, which is easily handled in software. One can imagine overlaying a grid on the pixmap that breaks it into these blocks, and hence we call the basic system the GRID system.

2.2 The Huffman table

Creating the Huffman table requires a preliminary pass through the document. Since white space is far and away the most frequent pattern, we expand our Huffman dictionary so that we may run-length encode white space blocks. Also note that one does not need to use all the patterns in the document in the Huffman code. Instead, one can use an escape code to explicitly send patterns that do not appear in the decoder's table. Although this costs in terms of overall compression, it allows one to control the size of the Huffman table, which as we have mentioned may be important for latency and memory utilization. We shall demonstrate that even relatively small Huffman tables still prove very effective. For most of our experiments, we adopt a table size of 20,000 entries, because this size balances good compression with the memory and latency requirements of the system under development.

The Huffman tables themselves can be compressed for additional savings. We suggest the following method, which in practice is both fast and effective:

1. Sort the patterns as 32-bit integers
2. Huffman encode the XOR differences (deltas) of consecutive patterns
3. Separately Huffman encode the code lengths of the patterns

The final encoding consists of the list of deltas with their associated Huffman code lengths. Note that if the de-compressor and compressor use the same scheme to establish the Huffman tree, it suffices to send the code lengths of each pattern instead of the actual codes.

This simple scheme is quite effective; in our experiments, for tables of 20,000 entries, we reduced the table size 50% over sending the patterns with no compression. We note that the problem of how best to arrange the list of entries to minimize the length after Huffman encoding the deltas seems similar to an NP-complete variant of the Travelling Salesman Problem, where the points are restricted to be vertices on a hypercube and the distance between points is the standard Hamming distance. More effective compression schemes may be possible[1]; this is still an open question.

[1]Indeed, we have found slightly better schemes using numerical differences and alternative prefix codes; however, the difference is small and the simple scheme is very fast.

Although one might hope that a single static Huffman table would prove effective over a large class of documents, particularly documents with the same font, in our experience this is ineffective when using small fixed size tables, as we shall show with results from our prototype.

Even with small Huffman tables, the document must be reasonably large for the amortized cost per page of the table to be small. Thus, GRID may prove less useful for small documents. As a general rule, we propose that GRID should be used on documents of ten or more pages.

2.3 White Space Improvements

By being more aggressive in encoding white space, we can more effectively utilize the Huffman table. In doing so, we relax our notion of overlaying a fixed grid on top of the document pixmap, and instead allow the grid to flex slightly. First, we make separate codes to run-length encode single blank horizontal lines. Second, we create a distinct "End of Line" code, to take advantage of the fact that most lines end with white blocks. Finally, we run-length encode white columns (4 by 1 columns) so that the first column of a pattern is always non-white. These enhancements lead to a significant improvement, particularly in the compression of images that represent Postscript documents. By ensuring that the patterns begin with non-white space, we remove redundancy from the pattern table, dramatically shrinking the number of patterns found in the document.

2.4 Lossy Compression

The frequency counts we find on the patterns can be used to develop a lossy compression scheme with good performance (especially on scanned images) at the cost of increased time for compression. The time for decompression, however, changes little with the lossy scheme, and in fact improves slightly. The idea of the lossy encoding scheme is also useful in designing more effective lossless compression schemes with only a small change in decompression time.

We suggest the following simple variation (others are possible): The compressor does a first pass to determine the frequency of the different patterns. On the second pass, when the compressor finds a pattern absent from the small Huffman table – that is, it finds a pattern that it cannot usefully compress – it attempts to find a pattern in the table that is "close" to the one to be compressed, and uses that pattern instead. The gain is substantial; in our experiments, a pattern not in the table requires more than four bytes to transmit, while patterns in the table take less than two bytes.

To make this scheme practical, one must adopt a definition of "close," and establish a data structure so that close patterns can be found quickly during compression. The definition should be chosen so that replacing the actual pattern by the close pattern should have negligible effects on the readability of the document, and therefore should correspond to some visually-based metric. In practice, for our experiments we chose to regard the pattern as a vector with 16 coordinates (corresponding to the

16 pixels in the pattern). Each coordinate can take four values, 0 to 3. We used as distance the 1-norm which corresponds to the total difference in shades between patterns.

The problem of finding a close pattern in this context has been addressed in the data structures literature: for example, see [1] and references therein. We adopted the following solution, based on the work of [1]. Suppose we wish to find the closest pattern in the table within some distance d. We hash each pattern several times, each time based on a subset of the pixels constituting the pattern. We choose these subsets in such a way to try to minimize the number of collisions in the hash table, while guaranteeing that any two patterns that differ in only d pixels will both be mapped to the same bucket at least once. The time to find a close pattern depends on the number of hash functions and the expected number of collisions; this time affects compression, but not the decompression.

Note that we can use the lossy model to derive an improved lossless scheme. We use the method above to find a pattern within some distance d of the one to be sent, and then represent modifications to the pattern using additional escape codes. Because of the large difference in the space required to represent patterns in the table and those not in the table, often two or three shade differences can be corrected while still gaining in overall space. This method can be extremely effective, especially on scanned documents. The gain comes at the price of additional complexity and time for both compression and decompression, although the penalty for decompression is small.

We note that this lossy approach can be applied with other compression algorithms and is independent of the GRID scheme. However, because the lossy scheme is based on GRID, it proves highly effective for GRID encoding.

3 Implementation Results

A preliminary implementation to gauge the efficacy of GRID has been developed. The primary goal was to determine the compression to be gained by the GRID scheme, and it has not yet been tuned for speed. Estimates of the number of operations necessary per pixel, however, suggest that decompression time will substantially outperform alternative methods based on arithmetic coding.

We developed our own test bed based on documents currently available in the underlying system. We included both scanned documents and Postscript documents of varying lengths; Table 1 provides brief descriptions of the sample documents. Here Spaper and Paper represent the same document; the underlying image from Paper was generated with Postscript, while the image for Spaper was obtained by printing the document and scanning it back in.

In the rest of this section, we highlight results from our implementation. The data we present represents typical performance. Note that compressed sizes for GRID do not include the size of the Huffman table; we chose not to include this because in our proposed application, the time per page is more significant than the total size, and because these tables may compress even more than we have found. With our scheme,

Reference	Article	P / S	Pages	Type
WinSock	Microsoft: Windows Sockets	P	137	Text
SRC #79	DEC SRC Report #79	P	80	Text/Math
Paper	Short CS Paper	P	14	Text/Math
Spaper	Short CS Paper	S	14	Text/Math
SRC #78	DEC SRC Report #78	S	44	Text/Figs.
C.Letter	Computer Letter, Vol.7 Num 2	S	12	Text

Table 1: Document index [P = Postscript generated, S = Scanned]

compressed Huffman tables require roughly 2 bytes per entry. Therefore the total size with a 20K table is about 0.04MB higher than the size given in the table.

3.1 Compression vs. Huffman Table Size

Table 2 demonstrates how GRID compression varies with the number of patterns. In the table, the n in GRID(n) refers to the number of non-white patterns allowed in the Huffman table, and the entries give the size in megabytes after compression. As one would expect, the compression improves with the size of the Huffman table; however, even very small tables allow significant compression.

In the table, we use the symbol for infinity to refer to the number of distinct patterns found in the document, or the table size one would have if allowed arbitrary table space. Note that the number of patterns does not grow linearly with the size of the document; instead it depends on the complexity of the document. In particular, scanning noise dramatically increases the number of patterns found; the number of patterns in Paper almost doubles in the scanned version Spaper.

	Postscript			Scanned		
	WinSock	SRC #79	Paper	SRC #78	C.Letter	Spaper
Pages	137	80	14	44	12	14
Size (MB)	31.78	18.56	3.248	10.21	2.784	3.248
GRID (10K)	2.534	1.218	0.230	1.041	0.583	0.291
GRID (20K)	2.323	1.132	0.205	0.982	0.547	0.268
GRID (30K)	2.220	1.095	0.198	0.945	0.528	0.248
GRID (∞)	2.015	1.047	0.198	0.738	0.406	0.214
$\infty =$	92312	51998	23106	133930	90802	45165

Table 2: Basic GRID performance (GRID(n) means n Huffman table entries)

3.2 Scanned Documents vs. Postscript Documents

The performance of GRID on scanned documents degrades noticeably when small Huffman tables are used. While some performance loss is to be expected, the significant discrepancy can be easily explained. Whenever a pattern is not found in our Huffman table, it must be explicitly encoded, essentially inducing a penalty of several bytes. In scanned documents, the noise leads to a significantly greater number of patterns that do not fall within the small table, as we have seen in Table 2.

Using the enhancements described previously leads to significant gains, especially in Postscript generated images. EGRID refers to an enhanced version of GRID that takes more care in compressing white space and uses special codes to encode patterns that are one shade (Hamming distance 1) away from some pattern in the table. The effects of EGRID are demonstrated in Table 3.

Allowing lossy encoding greatly improves compression on scanned documents, as can be seen in Table 3. The lossy version LGRID allowed up to three shades of difference between an observed pattern and the encoded pattern. Surprisingly, preliminary results suggest that lossy encoding actually improves the quality of scanned images by removing noise. The use of a lossy compression scheme to remove noise was investigated in [4], in a different setting. Quantitative results in this setting require some model of both scanning noise and the distribution of patterns in text documents, which we have not developed yet. Here we merely note that in a small number of perceptual tests users preferred the lossy version of the scanned document to the original.

	Postscript			Scanned		
	WinSock	SRC #79	Paper	SRC #78	C.Letter	Spaper
Pages	137	80	14	44	12	14
Size (MB)	31.78	18.56	3.248	10.21	2.784	3.248
GRID (20K)	2.323	1.132	0.205	0.982	0.547	0.268
EGRID (20K)	1.966	0.860	0.173	0.862	0.508	0.225
LGRID (20K)	1.861	0.843	0.173	0.739	0.432	0.199

Table 3: Improving GRID performance

3.3 Comparisons with Other Methods

We tested our system against a variety of alternatives (all run at their default settings). The best, in terms of compression, proved to be the mgbilevel system. However, the arithmetic coding used in this system makes it unsuitable given the time requirements of our application. We also tested the character-based mgtic program. The current implementation of mgtic proved unsuitable, for two reasons. First, many of the tested documents contained small pictures. Second, and more importantly, in documents with anti-aliased fonts, characters overlap regularly. The mgtic system attempts to recognize individual characters based on character boundaries, and hence fails on anti-aliased fonts.

	Postscript			Scanned		
	WinSock	SRC #79	Paper	SRC #78	C.Letter	Spaper
Pages	137	80	14	44	12	14
Size (MB)	31.78	18.56	3.248	10.21	2.784	3.248
GRID (20K)	2.323	1.132	0.205	0.982	0.547	0.268
EGRID (20K)	1.966	0.860	0.173	0.862	0.508	0.225
LGRID (20K)	1.861	0.843	0.173	0.739	0.432	0.199
JBIG	2.304	1.207	0.239	0.838	0.457	0.260
GZIP	2.174	1.219	0.237	1.017	0.547	0.302
MGBI	1.486	0.749	0.159	0.709	0.413	0.215

Table 4: Comparing GRID performance

The JBIG software of [3] provided comparable performance, both in terms of time and overall compression.[2] The decompression time for JBIG appears to be several times larger than the decompression time for GRID[3], although JBIG performs better in terms of compression time. Using lossy encoding, GRID outperforms JBIG in terms of overall size and decompression time, at the expense of compression time and latency before the first page can be shown.

A variation on GRID involves splitting the document into four by one columns and applying Lempel-Ziv type coding (often available on Unix systems as gzip). This variation follows the GRID paradigm of splitting the document into simple pieces, but adopts an alternative to Huffman coding. This method performs nearly as well as GRID, while avoiding the initial latency. Because Lempel-Ziv encoders are a mature technology, this variation may also prove suitable in some applications.

Table 4 summarizes the difference in overall compression among the techniques on our test bed of documents. Recall that the figures for GRID based systems do not include the approximately 40K necessary for the Huffman table.

3.4 Fixed tables

As mentioned previously, attempting to use a fixed table dramatically reduces overall compression. Here in Table 5 we briefly demonstrate the effect of using the table

[2]We note that we have found in the tested documents that JBIG performs better when the document is not split into planes, but the pixmap is simply treated as a bitmap. This may be true more generally in sparse documents, and may be worth further investigation. Moreover, it does not require re-integrating the planes after decompression.

[3]Because the GRID system has not yet been optimized, we do not have reliable timing statistics at this time.

from one document on another. In particular, note that even if we attempt to use the table for the scanned version of a document (Spaper) on the original (Paper) or vice versa, the compression is dramatically adversely affected. A direct comparison of the frequency tables of the patterns of the two documents demonstrates dramatic differences, and therefore this result is not as surprising at it might first seem.

File	Table	Size (MB)
WinSock	WinSock	2.323
	Paper	3.869
	SRC # 79	3.606
Paper	Paper	0.205
	SPaper	0.340
	WinSock	0.353
SPaper	SPaper	0.268
	Paper	0.401

Table 5: Fixed tables don't work! (Size after lossless compression)

4 Conclusion

Block-based approaches have become the norm in color image compression, but not in textual document compression. We have demonstrated that a pattern-based approach to document compression can be quite effective, even in comparison to pixel or character based approaches, and offers significant advantages in terms of decompression speed and lossy semantics. We believe that modifications of our general approach may be useful in a variety of arenas.

Some interesting theoretical questions have arisen as part of work. In particular, the questions of how best to compress an unordered list of numbers and how to find a close pattern in a small dictionary of patterns arise naturally, and we believe answers to these questions would have widely general applicability.

5 Acknowledgments

The authors would like to thank the several people at Digital Systems Research Center who have contributed to this research, and especially Michael Burrows and Paul McJones for their excellent suggestions.

References

[1] D. Greene, M. Parnas, and F. Yao. Multi-index hashing for information retrieval. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 722–731, 1994.

[2] P. G. Howard. The design and analysis of efficient lossless data compression systems. Technical Report CS-93-28, Brown University, 1993. (Ph. D. thesis).

[3] M. Kuhn. Software kit for jbigkit. Available via anonymous ftp from ftp.uni-erlangen.de as /pub/doc/ISO/JBIG/jbigkit-0.7.tar.gz.

[4] B. K. Natarajan. Filtering random noise via data compression. In *Proceedings of the 3rd IEEE Data Compression Conference*, pages 60–68. IEEE Computer Society Press, 1993.

[5] I. H. Witten, T. C. Bell, M. E. Harrison, M. L. James, and A. Moffat. Textual image compression. In *Proceedings of the 2nd IEEE Data Compression Conference*, pages 42–51. IEEE Computer Society Press, 1992.

[6] I. H. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, 1994.

[7] I. H. Witten, A. Moffat, T. Bell, et al. Software kit for mg. Available via anonymous ftp from munnari.oz.au in the directory /pub/mg.