Text Image Compression Using Soft Pattern Matching

PAUL G. HOWARD

AT&T Labs—Research, 100 Schultz Drive, Red Bank, NJ 07701-7033, USA
Email: pgh@research.att.com

We present a method for both lossless and lossy compression of bi-level images that consist mostly of printed or typed text. The key feature of the method is soft pattern matching, a way of making use of the information in previously encountered characters without risking the introduction of character substitution errors. We can obtain lossless compression which is about 20% better than that of the JBIG standard by direct application of this method. By allowing some loss based partly on the pattern matching using a technique called selective pixel reversal, we can obtain compression ratios about 2–4 times the compression ratios of JBIG and 3–8 times those of G3 facsimile with no visible loss of quality. If used in facsimile machines, these compression improvements would translate directly into communication cost reductions of the same factors, or into the capability of transmitting images at higher resolution with no increase in the number of bits sent.

Received July 11, 1996; revised June 3, 1997

1. INTRODUCTION

We address the problem of compression of bi-level images that consist mainly of printed or typed text. Current facsimile technology involves scanning and transmitting bi-level images. There is also a growing need for archiving document collections, often most efficiently done by scanning and storing bi-level images. For both of these applications, the material consists mainly of text, with occasional halftones and other types of images. Currently, images are scanned at a resolution of between 100 and 800 dots per inch (dpi). Since even at the comparatively low resolution of 200 dpi, a single page contains over 4 000 000 pixels (i.e. bits), the need for compression is obvious.

We address both lossless and lossy compression. In lossless compression, we insist that every pixel be correct in the reconstructed image. In lossy compression, we allow some pixels to be different, although we expect that the image will look no different to a human observer. Lossless compression can give compression ratios of between 10:1 and 40:1 for text documents at 200 dpi. Lossy compression has the potential for compression ratios about 2–3 times as large.

In this paper we describe a process which can be used for both lossless and lossy compression. For text documents at 200 dpi, our lossless compression ratios are between 20% and 65% better than those of the JBIG-1 standard [1]. Our lossy compression ratios are between 2.0 and 4.6 times the lossless ratios of JBIG-1, with only barely perceptible changes from the original. The lossless algorithm is similar to the method described by Mohiuddin *et al.* [2]; we extend the method to allow lossy compression by preprocessing each character in a way that reduces the number of bits output without noticeably distorting the character.

This paper is organized as follows. In Section 2 we offer a brief description of arithmetic coding; its use is required by most efficient bi-level image compression techniques, including ours. We show how the QM-Coder used in JBIG-1 handles the case of a two-symbol alphabet, and we show how we can extend the use of the QM-Coder to encode symbols from a multisymbol alphabet.

In Section 3 we discuss previous work, both on lossless statistical techniques (G3 and G4 facsimile standards and JBIG-1) and on lossy methods based on pattern matching and substitution. Section 4 describes soft pattern matching for lossless compression and selective pixel reversal for lossy preprocessing. We give implementation details, including parameter tuning. In Section 5 we give experimental results to compare our method with other available methods.

2. ARITHMETIC CODING

Arithmetic coding is a well-established coding technique that compresses a sequence of data optimally with respect to a probabilistic model. Its mechanism is to partition the interval [0, 1) of the real line into subintervals, the lengths of which are proportional to the probabilities of the sequences of events they represent. After the subinterval corresponding to the actual sequence of data is known, the coder outputs enough bits to distinguish that subinterval from all others. Methods are available that permit limited-precision arithmetic and incremental transmission. If probabilities are known for the possible events at a given point in the sequence, an arithmetic coder will use almost exactly $-\log_2 p$ bits to code an event which has a probability of p. We can think of the encoder and decoder as black boxes that use the probability information to produce and consume a

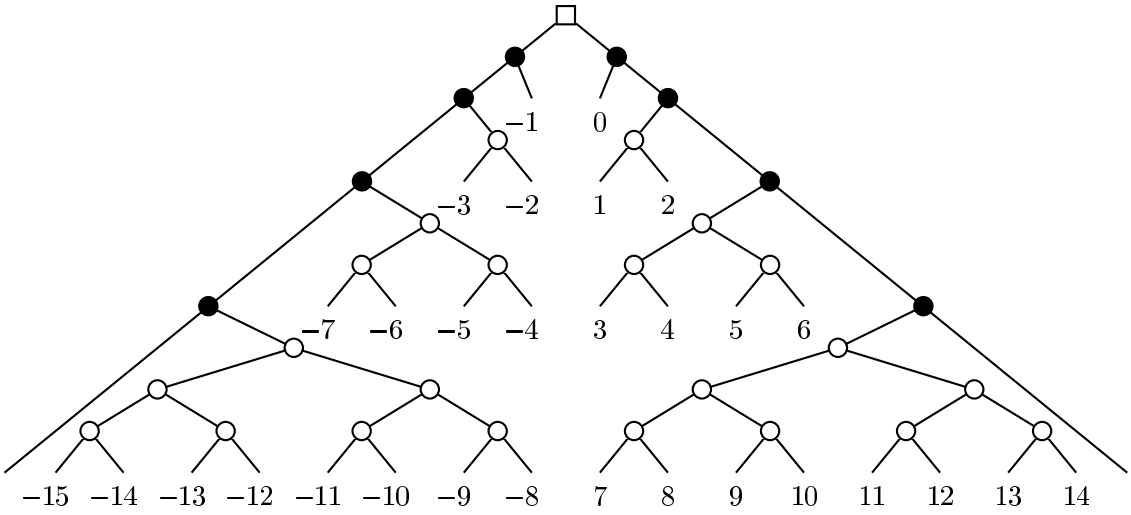

FIGURE 1. Part of the coding tree for multisymbol arithmetic coding. Each internal node represents one context with its own probability information, to be used by the QM-Coder. The square node at the root of the tree represents the first-phase decision, whether the integer n being coded is negative. The filled circles are the second-phase nodes, moving down the tree in ever-increasing ranges. The open circles represent third-phase decisions, traversing a complete binary subtree to reach the specific value of n.

bitstream. (See [3] or [4] for a detailed description of arithmetic coding.)

It does not matter to the encoder and decoder where the probabilities come from or how they change as long as the decoder has access to the same set of probabilities to decode an event that the encoder used to encode it. A different set of probabilities can be used for each event, allowing us to condition the coding of an event on the context in which it occurs. This requires the encoder and decoder to maintain a separate set of probabilities for each conditioning class.

Binary arithmetic coding

Arithmetic coding can deal with alphabets of any reasonable size, up to millions of symbols [5]. However, arithmetic coding tends to be slow except when the alphabet has just two symbols. In the binary case a number of simplifying approximations are possible. The QM-Coder used in the JBIG-1 standard [1] (and also the JPEG standard) provides both an approximate arithmetic coder and a table-driven technique for updating the probability estimate for each event's context after the event is coded; probability estimation is linked to coding both to increase coding speed and to use the coder as a source of pseudo-random bits.

Probability estimation for each context is done by a probabilistic finite-state machine. At each point in the coding, the probability estimate is in one of 226 states that represent both the probability of one of the two possible symbols in the given context and the confidence we have in the estimate. After the symbol has been coded using a close approximation to arithmetic coding, the probability estimate for its context may be changed depending on the actual symbol and possibly on an unrelated (i.e. pseudo-random) event internal to the coder. The combined mechanism of probability estimation and approximate arithmetic coding make up the QM-Coder. The QM-Coder is fast and easily implemented in software or hardware. Since there are only 226 probability states, each context's probability information requires only $\lceil \log_2 226 \rceil = 8$ bits of memory.

Multisymbol arithmetic coding

Arithmetic coding is more difficult when more than two symbols are possible at some point in the coding process. Here we describe an extension to the binary QM-Coder that can code any integer, positive, negative or zero. The extension involves repeatedly using the QM-Coder, narrowing the range of possible values at each step. In effect, we put the integers at the leaves of a binary tree and use the binary QM-Coder to code the sequence of branches needed to reach the desired integer. This is similar to the method used in the JPEG standard's lossless mode [6].

The tree described here codes any integer n using at most $2\lceil \log_2(|n| + 2) \rceil$ applications of the QM-Coder. Coding proceeds in three phases. The first phase consists of one decision, whether n is negative. In the second phase, we find an approximate range in which n lies by testing ranges of size 1, 2, 4, 8, 16, ... If n is non-negative, we test whether n is in $[0, 0]$, $[1, 2]$, $[3, 6]$, $[7, 14]$, $[15, 30]$, ..., the ranges doubling in size at each step. If n is negative, we test $[-1, -1]$, $[-3, -2]$, $[-7, -4]$, $[-15, -8]$, $[-31, -16]$, ... Finally, in the third phase we code the value of n within the range. Since the sizes of all ranges are powers of 2, we use a complete binary tree for this phase. Part of the tree is shown in Figure 1. The positive branch of the tree implements a code invented by Elias [7].

There are many other reasonable binary trees that we could use. Most of them give about the same amount of compression with about the same amount of computation; in particular, we will always need at least $O(\log n)$ binary choices for large values of n. Selecting a suitable tree is similar to the

well-studied problem of selecting an appropriate encoding for arbitrary integers.

A node of the tree consists of the 8-bit QM-Coder probability information and pointers to the left and right children of the node. To avoid excessive memory use, we build the tree in a lazy manner: we allocate memory only for nodes that are actually needed, at the time they are first needed.

Often the encoder and decoder know that a value to be encoded must lie within a certain range. When this leads to a decision being known with certainty when we are traversing the tree, neither the encoder nor the decoder codes the decision, although we may have to create new tree nodes.

We use this 'multisymbol-extended' QM-Coder in a number of places in the algorithm described here, to code image parameters, character identification numbers, relative and absolute character sizes, and character positions as offsets relative to other characters.

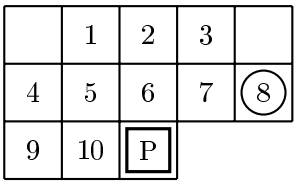

FIGURE 2. JBIG-1 template for lowest-resolution layer, also used in SPM for non-matched marks. The numbered pixels are used as the context for non-progressive coding of the pixel marked 'P'. This is the three-line template, used when JBIG-1 input parameter `LRLTWO` = 0. There is an alternative two-line template, but the three-line template generally works better for text images. Pixel '8' (circled) is the initial position of the adaptive template pixel. It can be moved during the encoding of an image, but for text images it is usually best to keep it fixed in its initial position.

3. PREVIOUS WORK

Previous work falls into two categories: (i) lossless compression of arbitrary bi-level images and (ii) lossy compression of textual bi-level images, often with an extension that allows lossless compression.

3.1. Lossless statistical compression techniques for arbitrary bi-level images

Most available techniques for compressing bi-level images are lossless. There are two basic approaches, run-length coding and arithmetic coding. Run-length-based techniques make use of the facts that large parts of most bi-level images consist of white space and that, for text images, the strokes that make up the characters are several pixels wide and many pixels long. ITU-T Group 3 (G3) and Group 4 (G4) facsimile standards call for coding a scan line as an alternating sequence of runs of white and black pixels, the lengths of the runs being coded using Huffman codes.

The ISO/IEC JBIG standard for lossless bi-level image compression [1] uses arithmetic coding. JBIG-1 has two modes, progressive and non-progressive. The progressive mode is not considered here. The following procedure describes non-progressive mode.

Non-progressive JBIG-1 algorithm

```
Code image dimensions
for each pixel in raster scan order do
    Find context consisting of neighbouring pixels
    Look up black probability for context
    Encode or decode pixel's colour using QM-Coder
    Update context probability using QM-Coder
end for
```

The pixels are coded in raster scan order. Each pixel is coded using a binary-arithmetic coder. The coder requires an estimate of the probability that the pixel being coded is black. In JBIG-1 this probability is estimated adaptively for each of a number of conditioning classes; the conditioning classes are based on the values of ten previously coded nearby pixels. The 10-pixel template relevant to text image compression is shown in Figure 2. The ten values taken together form a 10-bit vector that can take on $2^{10} = 1024$ different values; thus there are 1024 different conditioning classes, or contexts. Each pixel is coded by the QM-Coder, which uses the probability estimate for the pixel's context, then updates the estimate. Compression ratios are consistently better than those of the G3/G4 standards.

Mohiuddin *et al.* [2] describe a method similar to ours, in which they compress each character losslessly using a previously encoded mark as part of the context. They do not appear to distinguish between characters for which acceptable matches are found and those for which they are not. They also use a much smaller template (5 pixels instead of our 11), and of course they do not use the QM-Coder as their arithmetic coder since the QM-Coder had not yet been invented. Although they present their method as an extension of lossy pattern matching and substitution methods, they do not extend it back to lossy compression.

3.2. Lossy compression based on pattern matching and substitution

Some research has been done on lossy techniques for compressing text images using pattern matching and substitution [2, 8–15]. Conceptually, the idea is to segment the image into individual characters and to partition the set of characters into equivalence classes, each class ideally containing all the occurrences of a single letter, digit or punctuation symbol, and nothing else. Coding the image consists of coding the bitmap of a representative instance of each equivalence class, and coding the position of each character instance along with the index of its equivalence class. The equivalence classes and representatives can, of course, be constructed adaptively.

The following is a typical practical organization of the encoder for such a system. The various pattern matching and substitution (PMS) methods differ primarily in the matching technique employed.

Pattern matching and substitution algorithm

```
Segment image into marks
for each mark do
    Find 'acceptable match', if any
    if there is a match
        Code index of matching mark
    else
        Code bitmap directly
        Add new mark to library
    end if
    Code mark location as offset
end for
```

A significant drawback to PMS methods is the likelihood of substitution errors. To obtain good compression ratios, the pattern matcher must be aggressive in declaring matches, but as the matching criteria become less stringent, mismatches inevitably occur. A human reader can often detect and correct such errors (consciously or unconsciously), but sometimes this is not possible. Part numbers in images inherently have no context, and the figures '5' and '6' can be problematical even when context is available. We do not want to include typeface-specific or even alphabet-specific information in a general-purpose coder, but in most alphabets there are confusion pairs—different alphabet symbols that are similar according to most easily implemented matching criteria.

4. COMPRESSION BASED ON SOFT PATTERN MATCHING

The system described in this paper combines features of both the non-progressive JBIG-1 encoder and the PMS methods. The image is segmented into connected components of black pixels, which we call marks, following [15]. Each pixel of each mark is coded using a JBIG-like coder. We use pattern matching, but instead of directly substituting the matched character as in the PMS methods, we simply use its bitmap to improve the context used by the coder, a technique we call soft pattern matching.

Our system can be used in lossless or lossy mode. Lossless mode, described in detail in Subsection 4.1, is simpler, and is very similar to that of Mohiuddin *et al*. [2]. It is illustrated in the following procedure, which is similar to the generic lossy PMS method shown above; the only difference (shown in italics) is that lossy direct substitution of the matched character is replaced by a lossless encoding that uses the matched character in the coding context.

Soft pattern matching algorithm

```
Segment image into marks
for each mark do
    Find 'acceptable match', if any
    if there is a match
        Code index of matching mark
        Code bitmap using matching mark
        Conditionally add new mark to library
    else
        Code bitmap directly
        Add new mark to library
    end if
    Code mark location as offset
end for
```

The lossy mode follows the same outline, but requires the implementor to exercise some judgement in deciding on acceptable levels of distortion. The idea is to allow some pixels to be changed when doing so reduces the bit count without introducing any 'significant' changes to the image. In Subsection 4.2 we propose several ways in which loss can be safely introduced.

4.1. Lossless compression based on soft pattern matching

We now describe in detail a lossless compression method based on soft pattern matching. This method, called SPM (for soft pattern matching), applies only to images that consist mostly of printed or typed text. We expect that any necessary preprocessing has been performed to remove non-textual matter, such as halftone pictures, diagrams, handwriting and logos; this preprocessing step is not discussed here. For maximum compression, the non-textual material should be coded by some other method, such as JBIG-1, although our method can deal with any such material that remains in the image.

This method has a distinct advantage over PMS methods. In such methods a matching error leads directly to a character-substitution error. PMS methods can neither guarantee that there will be no mismatches nor detect them when they occur. In the soft pattern matching method, we use the matching mark only in the template, to improve our prediction of each pixel's colour. Even using a totally mismatched mark in the template leads only to reduced compression, not to any errors in the final reconstructed image. If the mark is well-matched, we take full advantage of our knowledge of it.

Segmenting the image into marks

As in PMS methods, we code the image by marks, or connected components of black pixels, so the first step is to segment the image into marks. Marks correspond roughly to letters, figures and punctuation symbols; the correspondence is not exact because of ligatures (ff fi fl ffi ffl), multipart letters (i j), multipart punctuation marks (: ; ! ? “ ” %), accented letters (é ö o̱), non-latin letters (Ξ), and multipart mathematical symbols ($= \equiv \leq \approx \vec{v}$), as well as because of

false ligatures and broken characters created by the printing and scanning processes. For this step we can use any standard segmentation technique. In our implementation we use a boundary-tracing method similar to the one described in [15, p. 263ff.]. We find four-connected black components rather than eight-connected components; this breaks up some false ligatures, which hurt compression more than broken characters. We do not attempt to separate the components of nested marks; for example, Θ, ⊙ and ⓒ are each treated as single marks.

To reduce the amount of memory required and to deal gracefully with vertical lines, very large letters, diagrams and other non-textual material, we use a form of image striping, arbitrarily limiting the vertical extent of extracted marks. Specifically, we maintain a buffer the width of which is the width of the image, but whose height is some fraction (5% in our implementation) of the larger of the image width and image height. (We also require that the buffer have at least 64 lines.) If we encounter an all-white scan line, we encode the entire buffer and flush it. Otherwise, when the buffer is full, we extract and code all marks that have black pixels in the top half of the buffer. This flushes at least the top half of the buffer, so we can continue. A connected component the height of which is more than the height of the buffer will be coded in pieces as more than one mark.

Coding marks in sequence

One step in processing each mark is to encode its position as an offset relative to a previously encoded mark. This can be done more efficiently if the offsets are mostly small. To keep the offsets small, we would like to code marks at least approximately line by line, and left to right within each line. Finding text lines is complicated by letters with descenders, by dots on 'i' and 'j', and by skewing introduced in the scanning process.

This is our procedure for putting marks in coding order, with some details omitted to avoid confusion. (The exact method used affects only the encoder, not the decoder.) We begin with marks sorted according to the position in raster scan order of the first pixel encountered. We find a tentative bottom scan line by finding the median of the bottom scan lines of the first 25 marks in the current list of unencoded marks. Then we select all marks the top scan line of which is at least as high as the tentative bottom scan line, and sort them according to the x-coordinate of their leftmost pixels. This method is designed for text oriented horizontally on the page, but experiments show that it can handle vertical text with a loss of compression efficiency of less than 10%.

Attempting to find a good matching mark

We attempt to find a previously coded mark that matches the current mark fairly closely. We do this by performing the following steps for each previously coded mark:

1. Prescreen the potential matching mark, skipping it if its size (or some other easily computed characteristic) is not close to that of the current mark. In our implementation, we reject potential matching marks that have a size in either dimension which differs by more than 2 pixels from that of the current mark.
2. Compute a match score, and call the potential matching character the best match if its score is better than that of any other potential matching mark tested so far. In our implementation, we compute the score by simply counting the number of mismatched pixels between the potential matching mark and the current mark when they are aligned according to the geometric centres of their bounding boxes.

After checking all previously coded marks, we decide whether the best match is 'acceptable' and code that decision using the QM-Coder with a single context, the acceptable match context. The best match is acceptable if its score is better than a prespecified threshold; the threshold may depend on characteristics of the current mark like its size. In our implementation, we accept a match if the number of mismatched pixels is at most 21% of the number of pixels enclosed in the bounding box of the current mark. The compression ratio is not particularly sensitive to the value of this threshold.

Coding a mark with no acceptable match

If no acceptable match is found, then we code the current mark using a fixed template of nearby pixels. This is exactly the method used in JBIG-1 and described in Subsection 3.1, except that we are applying it to a single mark, not to the entire image. We first code the height and width of the mark's rectangular bounding box using the multisymbol-extended QM-Coder. Then we code the pixels within the mark's bounding box in raster scan order exactly as in JBIG-1. For the context, we use the 10-pixel three-line template of Figure 2. (Other templates give slightly improved compression in some cases, but JBIG's template is natural and well-established, so we use it. The improvements are in the 1% range.) After coding the mark, we add it to the library of potential matching marks.

Coding a mark with an acceptable match

If an acceptable match is found, we code its index using the multisymbol-extended QM-Coder; then we code the bitmap of the current mark. To code the bitmap, first we code the numbers of rows and columns of the current mark relative to the corresponding numbers for the matching mark, using the multisymbol-extended QM-Coder. Second, we align the centre of the current mark with the centre of the matching mark. Finally, we code all the pixels within the bounding box of the current mark in raster scan order, using the QM-Coder as in JBIG-1 but with a different template. Each pixel's template consists of a combination of some pixels from the causal region of the current mark (pixels already seen and coded) and some more pixels from the matching mark in the neighbourhood of the pixel in the matching mark that corresponds to the current pixel. (We do not have to worry about causality in the matching mark since the matching mark is already entirely known by the decoder.) In our implementation we use 4 pixels from the current mark and 7

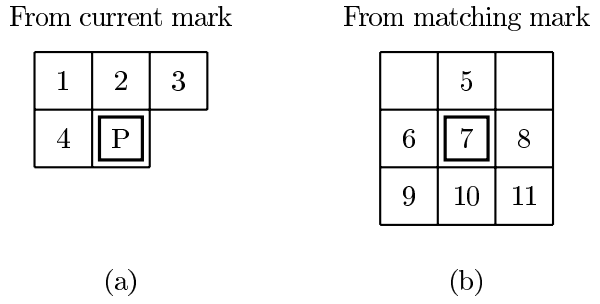

FIGURE 3. Template for matched marks. The numbered pixels are used as the context for coding the pixel marked 'P'. (**a**) Pixels taken from the causal part of the current mark. (**b**) Pixels taken from the matching mark. The pixel numbered '7' corresponds to pixel 'P' when the marks are aligned according to their centres.

more from the matching mark, so the template has 11 pixels, and there are $2^{11} = 2048$ different conditioning contexts. The pixels used in our implementation are shown in Figure 3. Since most of the pixels in the matching mark match those of the current mark (that is how we selected the matching mark), we can obtain better predictions and hence better compression by using some of the pixels in the matching mark, especially the pixel in the corresponding position, the pixel marked '7' in Figure 3.

After coding the mark, we add it to the library of potential matching marks. This can be done always, never or conditionally, based on the closeness of the match. In our current implementation we always add the new mark to the library. Adding the new mark conditionally would improve encoding speed by reducing the number of marks that must be tested as potential matches. The compression obtained might increase or decrease: reducing the number of possible matching marks reduces the number of bits required to identify the matching mark, but it often prevents using some very closely matching marks as matches in the coding process, increasing the number of bits required for encoding the bitmap.

Coding the mark's location

We specify the location of each mark by coding its offset relative to another previously coded mark, usually the most recently coded. For each mark after the first in a line, we code its horizontal offset as the signed distance from the right edge of the most recently coded mark to the left edge of the current mark. The vertical offset is the signed distance from the median of the bottom edges of the last three marks to the bottom of the current mark. For the first mark on a line, we code the horizontal offset as the signed distance from the left edge of the first mark on the previous line to the left edge of the current mark; we code the vertical offset as the distance from the bottom edge of the first mark on the previous line to the top edge of the current mark. (Using the distance to the bottom edge of the current mark instead of the top edge tends to give very slightly worse compression.) Before the offsets can be coded, the binary decision about the reference point is coded using the QM-Coder. The offsets themselves are coded using the multisymbol-extended QM-Coder, with separate contexts for each combination of offset direction (horizontal or vertical) and reference mark (most recent mark or first mark on previous line). The position of the first mark on a page is coded relative to the upper left-hand corner of the page, using the previous-line offset context.

Initialization

We obtain slightly better results by initializing the bitmap encoding contexts (the 1024 contexts for coding with no acceptable match and the 2048 contexts for coding with an acceptable match). The values we use are based on those that have been developed by the adaptive QM-Coder at the end of coding a certain 200 dpi image similar to test document #7; then we modify these values so that they represent less extreme probabilities, and so that they are more easily changed by the adaptation process.

4.2. Extension to lossy compression

There are several modifications that can be made to the method described in Subsection 4.1 to increase the compression ratio by making the compression lossy. All the variations involve preprocessing the original image followed by lossless coding of the modified image. This implies that the time for lossy encoding is greater than the time for lossless encoding, but the difference is not large.

The first three modifications are standard techniques that enhance the image while reducing the bit count as a side effect, while the other two involve preprocessing the marks in a way designed specifically to reduce the bit count while not degrading their appearance appreciably.

We can obtain some improvement in compression ratio by quantizing the offsets. For English text on a portrait-oriented page, character positions can be safely quantized to about 0.015 inch in the horizontal dimension and to about 0.01 inch in the vertical dimension; any more quantization causes noticeable distortion, but does not seriously affect legibility. Unfortunately, the increase in compression ratio is small, on the order of 1%, and the restored images do not look as good, so this is not a useful procedure.

We get some improvement by eliminating very small marks that represent noise on the page. In our implementation we ignore only marks consisting of single black pixels surrounded by eight white pixels, but we can safely ignore somewhat larger marks, up to about 0.01 inch in the larger dimension. The size in pixels of the ignored marks would thus depend on the scanning resolution.

We get more improvement by smoothing each mark before compressing it and thus before entering it into the list of potential matching marks. One simple smoothing that can be done is to remove single protruding pixels (white or black) along mark edges. Smoothing has the effect of standardizing local edge shapes, which improves prediction accuracy and increases the number of matching pixels between the current mark and potential matching marks. The increase in compression ratio is about 10%.

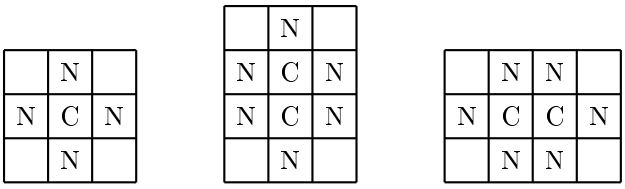

FIGURE 4. Patterns for determining whether potential reversed pixels are isolated enough to be reversed. The candidate pixels (i.e. poorly predicted pixels) are marked 'C'. Pixels marked 'C' are reversed if they are surrounded by one of the three patterns of non-candidates (marked 'N').

We can get a considerable improvement in compression by taking advantage of the best matching mark described in Subsection 4.1 and applying selective pixel reversal. For every pixel, we obtain a code length of less than 1 bit if we can predict its colour well (i.e. with probability greater than $\frac{1}{2}$), and greater than 1 bit if we cannot. One mechanism to reduce the code length is to reverse the colour of poorly predicted bits before coding them. To avoid seriously distorting the mark as a whole, we should limit the reversal to places where it will be imperceptible. We can generally reverse isolated poorly predicted pixels, and in fact reversing groups of two isolated pixels causes only a minor perceived difference. In our implementation we use 4-connectivity to decide whether a pixel or group of pixels is isolated.

Unfortunately, the exact colour probabilities are not readily available to the encoder during the preprocessing phase, before the actual coding begins. First, the QM-Coder changes the probabilities adaptively during coding. Second, there is a cascade effect: changing the colour of one pixel changes the context (and thus the colour probability) of up to four other pixels in the current mark. Evaluating all combinations of potential pixel reversals would be prohibitively slow, and would prevent the loss from being introduced during a preprocessing step, so instead we use a simple and fast approximation of each pixel's colour prediction: we approximate it by the colour of the aligned pixel in the best matching mark, that is, pixel '7' in Figure 3. Since the match score favours marks most of the corresponding pixels of which match, this is a good approximation.

We initially consider all mismatched pixels in the mark as candidates for reversal. Then we actually reverse those that fall into one of the patterns of Figure 4, that is, the 'isolated' mismatched pixels. After reversing the colour of the isolated mismatched pixels, we code the mark as in the lossless case, using the 11-pixel template of Figure 3.

Finally, we can also apply selective pixel reversal to improve the compression of marks with no acceptable matching mark. In this case there is no convenient corresponding pixel to serve as an approximation for the prediction of the current pixel. Instead, we choose as candidates (poorly predicted pixels) those pixels that differ from both of their two nearest causal neighbours, the one above and the one to the left. Other pixels are not necessarily well predicted; we just do not know that they are predicted badly enough that flipping them would be likely to reduce the bit count.

TABLE 1. Descriptions of the CCITT bi-level test images

Image	Description
1	Typed business letter (English)
2	Hand-drawn circuit diagram
3	Printed and typed invoice (French)
4	Dense typed report (French)
5	Printed technical article with figures and equations (French)
6	Graph with printed captions (French)
7	Dense Kanji document
8	Handwritten memo with additional large white-on-black letters

Resolution	Image no.	Image size
200 dpi	1–8	1728×2339
300 dpi	1–8	2592×3508
400 dpi	1–8	3456×4677
600 dpi	1–6, 8	5184×7016
600 dpi	7	5184×3035

Again, candidates are actually reversed only if they are 'isolated' in the sense of Figure 4. After reversing the colour of the isolated poorly predicted pixels, we code the mark as in the lossless case, using the 10-pixel template of Figure 2.

4.3. Lossless compression after lossy compression

Even after the image has been preprocessed and coded in a lossy fashion, it is possible to obtain a lossless coding with an acceptable compression ratio. The principle is to use the soft pattern matching method to encode the entire image, using the 11-pixel template of Figure 3 and treating the entire lossily encoded image as the 'matching mark'. This method is recommended in [15], with a larger template consisting of 17 pixels.

Both encoding and decoding are considerably slower since every pixel in the image is coded, and the compression ratios (including both the lossy and lossless components) are not as good as those of the lossless SPM method, but they are better than those of JBIG-1, and we can thus obtain a simple two-level progressive encoding. (Results using this method are not included in Section 5.)

5. EXPERIMENTAL RESULTS

We have tested our method on the eight bi-level images in the ISO/ITU test set at 200, 300, 400 and 600 dpi; the images are described in Table 1. We use the versions in the 'Standard Image Set (CD-03) Bi-Level/Gray Scale/Color Image Test Suite' of the National Communications System[1].

[1] Although these images are used within the International Organization for Standardization (ISO) for evaluation of image compression proposals, they are copyrighted. The CD-ROM is available at nominal cost from the International Telecommunications Union (`www.itu.ch`) as ITU-T Recommendation T.24 Encl. (11/94).

TABLE 2. Detailed results. Sizes of compressed files (in bytes) and compression ratios for various algorithms. For each combination of image number, resolution and algorithm, two numbers are shown. The top number (roman type) is the byte count, including overhead. The bottom number (italic type) is the compression ratio, that is, the ratio of the original file size (with no overhead) to the compressed file size (including overhead). `mgtic` was unable to compress some images

		Lossless					Lossy	
Image	Resolution	SPM	G3	G4	JBIG	`mgtic`	SPM	`mgtic`
1	200	9 299	23 714	16 619	12 858	10 814	5 125	4 532
		54.3	*21.3*	*30.4*	*39.3*	*46.7*	*98.6*	*111.5*
	300	14 351	38 212	25 338	19 459	15 792	6 631	6 348
		79.2	*29.7*	*44.9*	*58.4*	*72.0*	*171.4*	*179.0*
	400	19 418	55 143	34 516	26 077	21 710	8 255	9 361
		104.1	*36.6*	*58.5*	*77.5*	*93.1*	*244.8*	*215.8*
	600	46 293	108 143	72 874	52 339	52 404	15 692	31 956
		98.2	*42.0*	*62.4*	*86.9*	*86.8*	*289.7*	*142.3*
2	200	8 441	19 095	10 465	7 995	8 123	6 656	7 746
		59.9	*26.5*	*48.3*	*63.2*	*62.2*	*75.9*	*65.2*
	300	12 468	30 377	15 972	12 033	11 797	9 934	11 271
		91.2	*37.4*	*71.2*	*94.5*	*96.3*	*114.4*	*100.8*
	400	17 041	43 741	22 391	16 570	15 808	13 215	15 254
		118.6	*46.2*	*90.2*	*121.9*	*127.8*	*152.9*	*132.5*
	600	25 565	71 490	35 044	25 488	23 936	19 764	23 070
		177.8	*63.6*	*129.7*	*178.4*	*189.9*	*230.0*	*197.1*
3	200	15 971	36 640	25 591	19 987	–	8 749	–
		31.6	*13.8*	*19.7*	*25.3*	–	*57.7*	–
	300	24 334	58 906	38 444	30 138	27 060	11 762	14 450
		46.7	*19.3*	*29.6*	*37.7*	*42.0*	*96.6*	*78.7*
	400	34 370	84 866	53 224	41 677	38 935	16 864	23 092
		58.8	*23.8*	*38.0*	*48.5*	*51.9*	*119.8*	*87.5*
	600	61 269	147 670	90 003	69 859	73 802	29 446	44 384
		74.2	*30.8*	*50.5*	*65.1*	*61.6*	*154.4*	*102.4*
4	200	29 562	74 552	64 059	48 974	35 063	15 309	9 501
		17.1	*6.8*	*7.9*	*10.3*	*14.4*	*33.0*	*53.2*
	300	44 652	120 005	95 879	73 674	–	17 749	–
		25.5	*9.5*	*11.9*	*15.4*	–	*64.0*	–
	400	62 494	174 156	131 424	99 972	71 214	21 584	21 247
		32.3	*11.6*	*15.4*	*20.2*	*28.4*	*93.6*	*95.1*
	600	134 196	320 945	237 911	179 187	–	44 035	–
		33.9	*14.2*	*19.1*	*25.4*	–	*103.2*	–
5	200	17 149	40 091	29 286	23 244	18 921	9 808	8 197
		29.5	*12.6*	*17.3*	*21.7*	*26.7*	*51.5*	*61.6*
	300	25 583	64 369	44 024	35 005	27 863	12 448	12 190
		44.4	*17.7*	*25.8*	*32.5*	*40.8*	*91.3*	*93.2*
	400	35 721	93 039	60 784	47 926	–	16 283	–
		56.6	*21.7*	*33.2*	*42.2*	–	*124.1*	–
	600	65 251	161 331	102 700	80 323	74 925	27 227	42 458
		69.7	*28.2*	*44.3*	*56.6*	*60.7*	*167.0*	*107.1*
6	200	11 755	27 096	15 775	11 731	12 348	8 330	9 994
		43.0	*18.6*	*32.0*	*43.1*	*40.9*	*60.7*	*50.6*
	300	17 575	42 877	24 032	17 917	17 903	12 016	14 720
		64.7	*26.5*	*47.3*	*63.4*	*63.5*	*94.6*	*77.2*
	400	24 092	61 996	33 145	24 932	–	16 452	–
		83.9	*32.6*	*61.0*	*81.0*	–	*122.8*	–
	600	46 110	110 569	61 727	45 993	–	26 201	–
		98.6	*41.1*	*73.7*	*98.8*	–	*173.5*	–

Images 1, 3, 4, 5 and 7 contain substantial amounts of text; the other three images contain mostly line art, handwriting and graphical material. In all cases we have compressed the entire image, not just the text part; the non-text parts

TABLE 2. Continued

		Lossless					Lossy	
Image	Resolution	SPM	G3	G4	JBIG	mgtic	SPM	mgtic
7	200	38 798	78 563	66 434	52 261	45 368	25 967	24 794
		13.0	*6.4*	*7.6*	*9.7*	*11.1*	*19.5*	*20.4*
	300	55 857	121 966	97 359	79 542	61 580	32 471	29 174
		20.3	*9.3*	*11.7*	*14.3*	*18.5*	*35.0*	*39.0*
	400	75 016	172 329	131 015	107 529	82 304	38 857	39 073
		26.9	*11.7*	*15.4*	*18.8*	*24.5*	*52.0*	*51.7*
	600	61 812	128 289	93 195	75 096	68 805	30 186	44 130
		31.8	*15.3*	*21.1*	*26.2*	*28.6*	*65.2*	*44.6*
8	200	13 847	31 819	17 985	13 252	14 159	11 219	13 202
		36.5	*15.9*	*28.1*	*38.1*	*35.7*	*45.0*	*38.3*
	300	20 305	51 549	27 749	19 817	–	16 404	–
		56.0	*22.0*	*41.0*	*57.4*	–	*69.3*	–
	400	27 490	75 322	38 463	27 246	–	22 142	–
		73.5	*26.8*	*52.5*	*74.2*	–	*91.3*	–
	600	48 534	125 523	66 427	47 201	46 563	33 857	45 327
		93.7	*36.2*	*68.4*	*96.3*	*97.6*	*134.3*	*100.3*

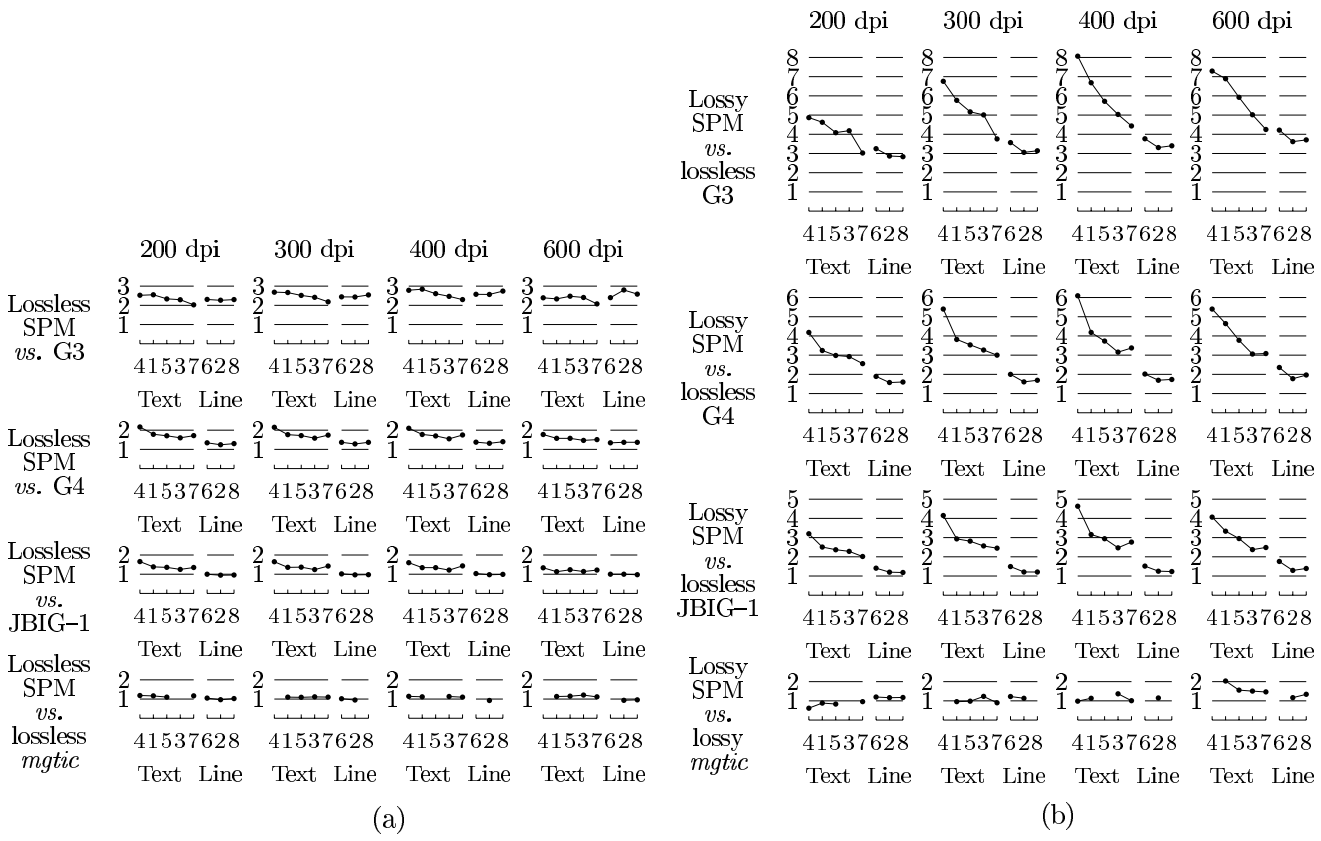

FIGURE 5. SPM compression compared with that of other methods. (**a**) Lossless SPM. (**b**) Lossy SPM. Each of the small graphs shows the ratio between the SPM compression ratio and that of the other method for each of the five mostly textual CCITT test images (numbers 4, 1, 5, 3 and 7) and the three line art images (numbers 6, 2 and 8). The larger the ratio, the better SPM does relative to the other method; a ratio of 1 means that the methods give the same compression.

are naïvely treated as though they were text. The non-text images are included for completeness, to show that SPM is able to compress them competitively even though it is designed strictly for text.

The 600 dpi images contain more isolated noisy pixels than those at the other resolutions. In addition, in the process of producing the test suite the 600 dpi version of image #7 was truncated to 43% of its correct size; to avoid confusion in the standards community this was never corrected. We compare our method with three lossless methods: two-dimensional Group 3 facsimile with $K = 4$, Group 4 facsimile and JBIG-1, and with one lossy/lossless method, the `mgtic` program described in [15]. (`mgtic` could not compress 8 of the 32 images on our hardware.)

The results are given in detailed form in Table 2, showing byte counts and compression ratios for all images using all methods, lossy and lossless. The results are also displayed graphically in Figure 5. In the graphs the images are divided into the five textual images, arranged so that the images with a more textual nature (with respect to the Latin character set) tend to be to the left, and the three non-textual images, arranged on a similar basis. The graphs show the factor by which SPM improves compression compared with the various other methods.

For each of the five textual images at all four resolutions, lossless SPM compresses better than all four other lossless methods tested. It is 2–3 times better than G3, and 21–66% better than JBIG-1 (except for the noisy 600 dpi images).

Lossy SPM is 3–8 times better than lossless G3, and 2–4.6 times better than lossless JBIG-1. The advantage to lossy SPM increases somewhat at higher resolutions. Lossless SPM is 10–20% better than lossless `mgtic`. Lossy SPM is somewhat worse than lossy `mgtic` at 200 dpi, but over 50% better at 600 dpi. For all the textual images, lossy SPM outputs fewer bits at 600 dpi than lossless G3 outputs at 200 dpi; the saving ranges from 11% for image #7 to 41% for image #4.

At low resolutions lossy `mgtic` produces higher compression ratios than lossy SPM, but this comparison takes no account of distortion; it should be tempered by the realization that `mgtic` makes occasional substitution errors. At higher resolutions lossy SPM outperforms lossy `mgtic`.

Our implementation was not tuned either for speed or for efficient use of memory, but we can make a few general comments regarding resource usage. Both the encoder and decoder use large amounts of memory—several megabytes for most of the test images. (Since we store bitmaps in the library using a run-length form, the library size grows only linearly with the resolution.) Any pattern matching algorithm has similar memory requirements and, like ours, can be tuned to use less memory. However, pattern matching algorithms will never come close to the small constant amount of memory required by G3, G4 or JBIG-1. In our method, again as in all pattern matching algorithms, encoding is considerably slower than decoding because of the need to search for library marks matching each mark. Depending on the number of marks in the image, the time for encoding can range from a few seconds (on a SPARCstation 10) to a few minutes. In our implementation the time increases approximately as the square of the number of marks. Various mechanisms can be used to speed the search. The time used for arithmetic encoding and decoding is actually smaller than that of JBIG-1, since far few binary decisions need to be coded. For example, SPM requires only 1 132 730 binary decisions to encode the 9 092 736-pixel image #1 at 300 dpi.

6. CONCLUSION

We have presented a method for highly efficient compression of bi-level images consisting mostly of printed or typed text. The basic soft pattern matching method is lossless, but it can be made lossy by preprocessing the image; selective pixel reversal is especially effective. The lossless method compresses more than JBIG-1, the best existing standard. The lossy extension gives compression ratios up to more than 4 times those of lossless JBIG-1. Compared with G3, the method commonly used in fax machines, our lossy method gives 3–5 times the compression (and thus requires only roughly 20–35% as many bits) at 200 dpi; for normal type sizes (down to about 8 point) there is virtually no visible degradation of quality. Alternatively, we can apply our lossy method to higher resolution images (600 dpi); this uses fewer bits than 200 dpi G3 and gives considerably higher quality.

REFERENCES

[1] JBIG (1993) Progressive bi-level image compression. ISO/IEC International Standard 11544, ITU Recommendation T.82.

[2] Mohiuddin, K., Rissanen, J. J. and Arps, R. (1984) Lossless binary image compression based on pattern matching. *Proc. Int. Conf. on Computers, Systems, and Signal Processing*, Bangalore, India, pp. 447–451.

[3] Howard, P. G. and Vitter, J. S. (1994) Arithmetic coding for data compression. *Proc. IEEE*, **82**, 857–865.

[4] Witten, I. H., Neal, R. M. and Cleary, J. G. (1987) Arithmetic coding for data compression. *Commun. ACM*, **30**, 520–540.

[5] Moffat, A., Neal, R. and Witten, I. H. (1995) Arithmetic coding revisited. *Proc. Data Compression Conf.*, Snowbird, UT, pp. 202–211.

[6] Pennebaker, W. B. and Mitchell, J. L. (1993) *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York.

[7] Elias, P. (1975) Universal codeword sets and representations of integers. *IEEE Trans. Inform. Theory*, **IT–21**, 194–203.

[8] Ascher, R. N. and Nagy, G. (1974) A means for achieving a high degree of compaction on scan-digitized printed text. *IEEE Trans. Comput.*, **C-23**, 1174–1179.

[9] Pratt, W. K., Capitant, P. J., Chen, W. H., Hamilton, E. R. and Wallis, R. H. (1980) Combining symbol matching facsimile data compression system. *Proc. IEEE*, **68**, 786–796.

[10] Wong, K. Y., Casey, R. G. and Wahl, F. M. (1982) Document analysis system. *IBM J. Res. Develop.*, **26**, 647–656.

[11] Johnsen, O., Segen, J. and Cash, G. L. (1983) Coding of two-level images by pattern matching and substitution. *Bell Syst. Tech. J.* **62**, 2513–2545.

[12] Holt, M. J. J. and Xydeas, C. S. (1986) Recent developments in image data compression for digital facsimile. *ICL Tech. J.*, 123–146.

[13] Holt, M. J. (1988) A fast binary template matching algorithm for document image compression. *Proc. 4th Int. Conf. on Pattern Recognition*, Cambridge, UK, pp. 230–239.

[14] Witten, I. H., Bell, T. C., Harrison, M. E., James, M. L. and Moffat, A. (1992) Textual image compression. *Proc. Data Compression Conf.*, Snowbird, UT, pp. 42–51.

[15] Witten, I. H., Moffat, A. and Bell, T. C. (1994) *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York.