Symbol Dictionary Design for the JBIG2 Standard

Yan Ye, Dirck Schilling, Pamela Cosman, Hyung Hwa Ko†

ECE Dept., Univ. of California at San Diego, La Jolla, CA 92093-0407
{yye,dschilli,pcosman}@code.ucsd.edu (858)822-1250
†Dept. of Elec. Comm. Eng., Kwangwoon Univ., Seoul, Korea 139-701
hhkoh@daisy.kwangwoon.ac.kr +82-2-940-5137

Abstract: The JBIG2 standard for lossy and lossless bi-level image coding is a very flexible encoding strategy based on pattern matching techniques. The encoder collects a set of symbols in a dictionary and encodes a page by reference to the dictionary symbols. JBIG2 allows the encoder to view all symbols and choose a good set for the dictionary. We propose a two-pass technique for choosing the dictionary entries, and discuss the trade-offs in bit allocation that these choices entail. The proposed dictionary design technique outperforms simpler dictionary formation techniques by up to 7-9% and 10-18% for lossless and lossy compression, respectively. In the lossy case, we further point out one way to significantly speed up the symbol bitmap compression procedure.

1 Introduction

The technical specifications of JBIG2 have been established [1, 2]. It will be the first international standard that provides for both *lossless* and *lossy* compression of bi-level images. It is meant for both text and halftone data; a JBIG2 encoder is expected to segment an image into different regions [3] and use different coding mechanisms for text and for halftones. In this paper, we are only concerned with the coding of text images. In JBIG2, the coding of text is based on pattern matching techniques [1].

On a typical page of text, there are many repeated characters. Rather than coding all the pixels of each occurrence of each character, we code the bitmap of one representative instance of the character and put it into a dictionary. Such a bitmap is called a "symbol" in text regions. Two instances of the same character often do not match pixel for pixel. One can measure the mismatch between one instance of a character and another instance of the same or another character. For example, the Hamming distance is a measure of the mismatch and is the measure we use throughout this paper.

In *pattern matching and substitution* (PM&S) [4], a criterion for deciding the mismatch between two symbols is chosen. For coding a new symbol, we look for the dictionary entry which has the smallest mismatch with the symbol to be coded. If that smallest mismatch is less than the preset threshold, we code the symbol by using a pointer to the dictionary entry. If the closest dictionary entry is too far away, we code the new symbol bitmap directly and then add it to the dictionary. In either case, we have to encode also the position of the symbol on the page, usually relative to the previously encoded symbol.

Use of PM&S allows high lossy compression ratios for bi-level images that have many repeated symbols. However, there are inevitable substitution errors. If these errors are not acceptable, JBIG2 allows the use of a residual coder. The difference between the lossy version and the original image can be coded by a JBIG1-style lossless coder, resulting in a final lossless image that has better compression than JBIG1.

JBIG2 also allows an alternative to PM&S called *soft pattern matching* (SPM) [5]. In SPM, the dictionary entry that is referenced is not directly substituted for the new symbol. Rather, it merely provides the context for arithmetic encoding of the new symbol. Therefore, even using a totally mismatched reference symbol merely leads to reduced compression efficiency, not to any substitution errors. One can also choose to do lossy compression with SPM. There are different ways to introduce the loss and they will result in different levels of lossiness. In [5] several approaches for lossy compression are proposed. These approaches are essentially free of substitution errors except for very tiny fonts. We will present the details in Section 3.

In this paper we are only concerned with SPM-based JBIG2 encoder design. In particular we investigate the problem of symbol dictionary design. Choosing cleverly a proper subset from all the extracted symbols as the symbol dictionary is essential for the encoder to achieve good compression ratios. This paper is arranged as follows. In Section 2, we present the proposed dictionary design technique together with two existing simpler dictionary formation techniques. In Section 3, three approaches for lossy compression from [5] are explained. Finally in Section 4 we will compare both lossless and lossy compression performance of all three techniques on two CCITT standard images.

2 Dictionary structure in JBIG2

In JBIG2, information to be transmitted is organized in segments. As discussed in the JBIG2 final committee draft [1], for a typical page image with mostly text and maybe some general graphics, a group of symbol dictionary entries will be transmitted first in the *symbol dictionary segments*. There are two types of symbol dictionaries, *direct dictionary* and *refinement/aggregate dictionary*. Entries in the direct dictionary are symbols that do not refer to any other symbols, and their bitmaps are arithmetically or Huffman coded using only causal neighbors from themselves as context. This type of bitmap coding is called *direct coding*. Entries in the refinement/aggregate dictionary are symbols that refer to some other dictionary symbol (either in the direct or in the refinement/aggregate dictionary), and thus their bitmaps are arithmetically coded using causal neighbors from themselves, as well as unrestricted neighbors from the referenced symbol. This type of bitmap coding is called *refinement coding*. Figure 1 shows two JBIG2-compliant context templates for arithmetic direct coding and refinement coding. After transmitting the dictionaries, next are encoded the *text region segments*, in which the makeup of the input page is described to the decoder. Text region segments describe locations and dictionary indices of the extracted symbols, possibly followed by some refinement coded bitmaps. We call the refinement coding of bitmaps that happens in the text region segments *embedded coding*. Finally, the *generic* *region segments* which describe the remaining general graphics in the page are encoded.

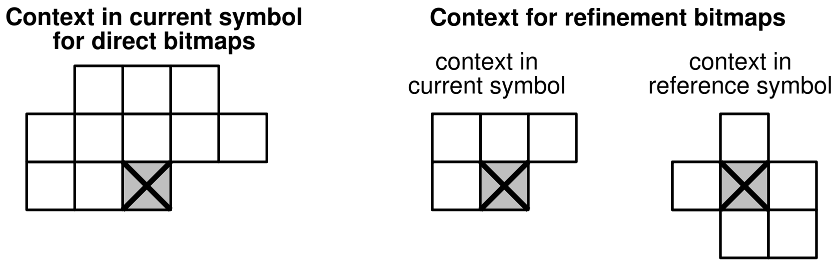

Figure 1: Contexts used for encoding direct and reference bitmaps.

A design question for a JBIG2 encoder is: how many symbols and what symbols should be put into the dictionary? The basic trade-off is as follows. If one puts more symbols into the dictionary, coding their bitmaps increases the dictionary's bit cost. Also, each reference to a dictionary entry will take more bits, thus the overall *index coding* will require more bits. On the other hand, with more symbols in the dictionary, all the symbols will have a broader range of choices for more accurate matches, and hence *refinement coding* will take fewer bits. Our goal is to design a reasonably small dictionary that can still provide comprehensive and accurate reference information. This involves not only the issue of choosing a good representative small subset from the entire extracted symbol set as the symbol dictionary (and hence putting the remaining symbols into embedded coding), but also the question of how to efficiently compress the dictionary symbols themselves. That is, how should we decide the reference relationship between them in such a way that each symbol can have a very accurate reference and thus be compressed efficiently?

In this section, we will first introduce two simple existing methods for dictionary formation, and then we will elaborate on the proposed dictionary design technique which can give higher compression ratios.

2.1 Simple methods for dictionary formation

In [5], Howard described a sequential way to form the symbol dictionary. As each symbol is extracted from the page, its location is transmitted relative to the previous symbol location, and its bitmap is matched with the existing dictionary entries (this match is prescreened by size) and transmitted with direct or refinement coding depending on whether the best match found is acceptable. Then, the new symbol is added to the dictionary. In this way the dictionary size keeps increasing as the page image gets processed, and by the end of the page, all symbols in the page have been transmitted and the dictionary will contain every extracted symbol. We will refer to this as the *one-pass* technique.

The one-pass dictionary formation is a natural choice for the original SPM encoder because the whole compression scheme is purely sequential and one-pass. However, a JBIG2 encoder pre-processes the entire image (or maybe a portion of the image if memory is limited) and accumulates all symbol information before it starts to send segments. Therefore, the encoder can examine the one-pass dictionary built on the fly and exclude *singletons* from it [6, 7]. Singletons are dictionary entries that are never referenced by other symbols.

They are detrimental to compression efficiency in that they provide no useful reference information yet dictionary indices are allocated to them, thereby increasing the length of all dictionary indices. Singletons are of two types: *direct singletons* and *refinement singletons*. Direct singletons are symbols that neither refer to other symbols nor are referred to by other symbols; usually they are "strange" symbols in the image, e.g., a company logo in a business letter. Direct singletons can be removed from the direct dictionary and relegated to generic region segments to be coded together with other general graphics. The second type of singletons, refinement singletons, are symbols that refer to other symbols but are not referred to by any one else. They can be removed from the refinement dictionary and relegated to the text region segments for embedded coding. We will refer to this second type of dictionary formation method as *singleton exclusion*.

2.2 Two-pass dictionary formation

Even with singleton exclusion, the dictionary formation is still essentially one-pass in the sense that each symbol only finds its best match among all previous symbols. Because a JBIG2 encoder has access to the entire set of extracted symbols before any actual coding is carried out, we can expect a *two-pass* dictionary to fare better by providing each symbol with *all* other symbols as its potential best match. In some previous literature [8] the term "two-pass dictionary" was used to mean doing a first pass through the dictionary symbols to collect the bitmap context statistics and a second pass to actually code the bitmaps using these statistics. We use the term "two-pass dictionary" differently. The arithmetic coders still have no prior knowledge of statistics but encode adaptively; "two-pass" instead refers to the fact that the JBIG2 encoder takes a first pass to extract all the symbols from the page, and then after the dictionary is chosen, takes a second pass to encode the page using that dictionary. In this sense it is similar to the "multi-pass" methods employed in [9, 10] and by MGTIC [6].

To design a good dictionary we need to select a suitable subset of symbols out of the entire symbol set, and find an efficient way to encode the dictionary symbols themselves. To select dictionary symbols, we define a *class* to be a set of symbols for which

(a) each symbol in a class has its best match symbol also within the class, and

(b) there is no way of subdividing the class into subgroups in such a way that each symbol in a subgroup still has its best match in the subgroup.

One way to partition the entire symbol set into classes is to point each symbol to its best match. This way the symbol set is partitioned into small pieces of connected graphs with very similar symbols clustered together. These connected graphs have weighted edges where the weights are simply the mismatch scores between two symbols. It is easy to prove that each such connected graph forms a class as defined above. Since class symbols are very similar to each other, we decide that only one dictionary symbol is needed for each class. This dictionary symbol is called the *class representative*. We choose the class representative as the symbol with the smallest average mismatch within the class.

Now that a dictionary containing class representatives has been formed, we follow a similar procedure to efficiently compress the dictionary itself. Each dictionary symbol is pointed to its best match among other dictionary symbols, so the dictionary symbols are partitioned into small connected graphs, each of which has one and only one loop inside; we call these connected graphs *super-classes* (see Figure 2). Each loop is broken by eliminating the edge with the highest weight (i.e., biggest mismatch score); this generates one super-class leader for each super-class. Then these super-class leaders find their best matches among *other* super-classes (not just other super-class leaders). In this way more super-classes are merged together and new bigger super-classes are generated. While repeating this merging procedure each time we will have another unique loop within each newly generated super-class, which is broken in the same manner as before. In this way, super-classes keep getting connected to each other; the total number of existing super-classes keeps decreasing and the size of each super-class keeps increasing. This iteration is repeated until no super-class leader can find an acceptable best match among the other super-classes, i.e., the smallest mismatch score is no longer below the preset threshold. At this point the dictionary symbol set is partitioned into disjoint super-classes that are sufficiently dissimilar from each other, and we have obtained a suboptimal solution to the reference relationship among the dictionary symbols. The final super-class leaders go into the direct dictionary and others into the refinement dictionary. Note that with this two-pass dictionary the natural order by which symbols are extracted has been changed, and we will need to re-order them so that each symbol comes after its reference symbol.

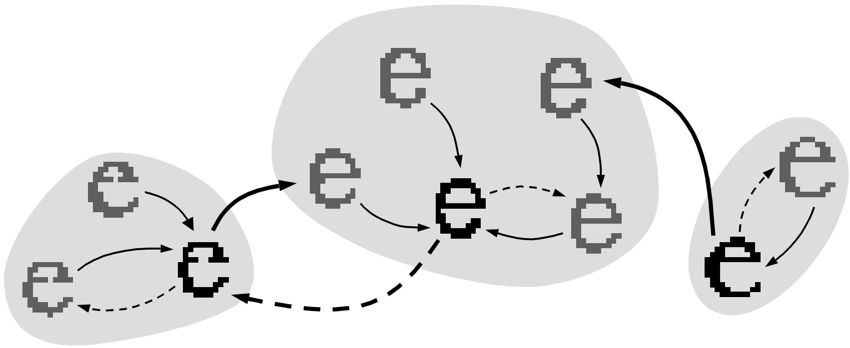

Figure 2: An example of how super-classes are merged. Shaded areas indicate three existing super-classes. Symbols shown in black are super-class leaders. Thin arrows show the reference relationship within each super-class. Each super-class contains a unique loop; thin dashed arrows show where to break these loops. Thick arrows show how the three leaders are matched at the next round. The thick dashed arrow is how the new unique loop is broken. In this way the three existing super-classes get merged into one new super-class.

3 Lossy compression using SPM

Our lossy compression incorporated three approaches introduced in [5]. These approaches have two outstanding advantages:

(a) except in the special case of very tiny symbols, no substitution error is introduced and the loss of visual information is almost imperceptible. This is because flipping of pixels takes place only in a very restricted fashion;

```
void build_super_classes() {
  /* init old super classes */
  total number of old super classes = total number of dictionary symbols;
  for(each dictionary symbol) copy it into a super class and make it the leader;

  do {
    /* match the old super classes */
    for(each old super class leader) find its best match from other super classes;

    /* merge old super classes into new super classes */
    merge old super classes into new ones;
    for(each new super class) {
      find and break its unique loop;
      decide its leader;
    }

    if(number of new super classes == number of old super classes) break;
    old super classes = new super classes;
  } while(1);
}
```

Figure 3: Psuedo-code for generating super-classes among dictionary symbols.

(b) if a lossless version of the original image is to be requested later, through refinement coding, the encoder can very efficiently transmit the original image by referring to its lossy version that has already been sent. This is because the difference between the two images will mostly occur around the borders of symbols, which is a fairly predictable event and thus the arithmetic coder can learn quickly.

These techniques pre-process symbols extracted from the input image. To briefly review them, we will name them *speck elimination*, *edge smoothing* and *shape unifying*. Speck elimination wipes out very tiny symbols or flying specks. In our experiments we define flying specks as symbols whose size is no bigger than 2×2. Edge smoothing tries to fix jagged edges. Protruding single black pixels or indented single white pixels are flipped. Both of these techniques can improve the compression by decreasing the number of "unexpected" events, thus improving the prediction accuracy of the arithmetic coders. This improvement is usually around 8% compared with the lossless case. On the other hand, considerable compression improvement can result from the third technique, shape unifying (see Figure 4). Shape unifying tries to modify the current symbol's bitmap at isolated locations to make it as similar to its reference bitmap as possible, without introducing "visible" changes. This is achieved by

(1) calculating the error bitmap between the two bitmaps;

(2) flipping black pixels inside the error map *only* if they are isolated single pixels or groups of 2 pixels.

Experiments show shape unifying together with speck elimination and edge smoothing improves compression by around 40% over purely lossless compression, while guaranteeing very little "perceptible visual change". The reason why shape unifying is so efficient is

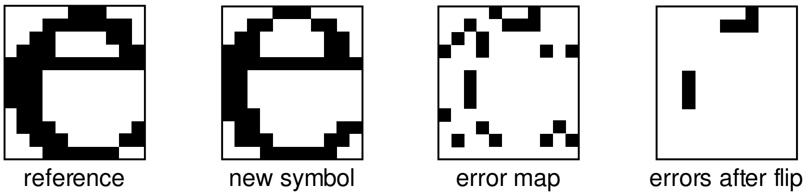

Figure 4: An example of shape unifying using reference symbol.

that isolated differences between the current bitmap and its reference are very detrimental to arithmetic coding efficiency (even more so than clustered differences); shape unifying completely eliminates such events.

Another important observation for lossy compression with shape unifying is the existence of *perfect symbols*. Perfect symbols are those refinement symbols that become exactly the same as their references after the shape unifying procedure. For perfect symbols, the following two facts hold true:

(1) there is no need to encode a symbol's bitmap if it is perfect, and

(2) there is no need to assign a dictionary index to a perfect refinement dictionary symbol.

In terms of compression ratio, the saving from (1) is trivial since the arithmetic coders can encode perfect symbol bitmaps with very few bits anyway, but the saving from (2) is not trivial because this can further reduce the dictionary size and hence bits spent on index coding. On the other hand, obviously (1) saves a non-trivial amount of compression time by skipping the expensive arithmetic coding procedure for all perfect symbols. Experimental results in Section 4.2 will show that shape unifying can render a significant number of perfect symbols.

Notice that speck elimination and edge smoothing are truly pre-processing techniques. Shape unifying, on the other hand, is not strictly a pre-processing procedure since it is not done until after symbols find their best matches in the dictionary, which, in the case of a two-pass dictionary, will not happen until the dictionary itself is finally decided. Once the entire symbol set is altered with these three approaches, they are compressed losslessly as usual; no further loss will be introduced.

4 Results

In this section we show the experimental results using the one-pass, singleton exclusion and two-pass dictionaries. The test images are the two standard CCITT images f01_200 and f04_200. These two images contain mostly text; f01_200 is a business letter in English with about 1,000 symbols, and f04_200 is a page from a French book with about 4,400 symbols. Both lossless and lossy compression are carried out to compare the coding performance of the three dictionary design approaches. The one-pass approach is consistently the most inefficient. The singleton exclusion approach can improve compression by a noticeable amount, and the proposed two-pass technique is shown to always give the best results, with improvements up to 7-9% in the lossless case and 10-18% in the lossy case. In lossy compression, we have further observed a very efficient way to speed up the bitmap coding procedure by up to 48%.

4.1 Lossless Compression

In all the dictionary formation methods described above, there is a parameter, the symbol mismatch threshold, that is to be decided. Yet the overall compression of SPM-based algorithms is not very sensitive to this mismatch threshold [5]. This is because a higher threshold allows more symbols to be refinement coded (refinement coding is usually more efficient than direct coding) but the quality of the reference bitmaps will be lower, and vice versa. Therefore, as long as the threshold is not too high or low, the overall compression is not affected much. Using the percentage of different pixels in two bitmaps as the mismatch measure, our experiments show that the overall compressed bit rates differ by less than 1% under different thresholds between 0.1 and 0.2. The optimal value occurs around 0.15; therefore 0.15 is used as the mismatch threshold for all the results reported.

Table 1 shows the total number of bits required to losslessly encode the test images f01_200 and f04_200 using the one-pass, singleton exclusion, and proposed two-pass techniques. The images are of size 1728×2339. The singleton exclusion gives 4-6% improvement over the one-pass technique, and the two-pass technique further brings this percentage of improvement up to 7-9%.

In [10], the question of dictionary generation following symbol extraction is also addressed; this is perhaps the closest prior work to ours. Our lossless compression ratios of 51.8:1 and 15.7:1 for f01_200 and f04_200 are substantially higher than the results in [10]. However, the overall systems differ in the methods of index coding, location coding, and the use of SPM versus PM&S, in addition to the difference in codebook generation algorithms, so the resulting compression ratios cannot be taken as a comparison of the codebook generation algorithms.

	one-pass	singleton excl. (% imprv.)	two-pass (% imprv.)
f01_200	83,731	80,514 (3.8%)	77,963 (6.9%)
f04_200	281,037	264,884 (5.8%)	256,965 (8.6%)

Table 1: Total bits required for lossless compression with the 3 different methods.

To examine more closely the trade-off associated with symbol dictionary design, and the benefit from using the two-pass dictionary, in Figure 5 we compare the dictionary sizes and refinement coding efficiencies of the singleton exclusion and two-pass approaches. The purely one-pass approach is omitted here because of its inferior performance. For both images, the size of the two-pass dictionary was substantially reduced from that of the singleton exclusion dictionary, with most of the reduction coming from the refinement portion of the dictionary. Yet with this much smaller dictionary, the refinement coding achieved is still nearly as efficient as before. This shows that the two-pass approach provides a better way to choose the dictionary symbols, thus resolving the bit rate trade-off at a better point.

4.2 Lossy Compression

Lossy compression is also carried out to compare the proposed dictionary design technique with the other two methods. Results are given in Table 2.

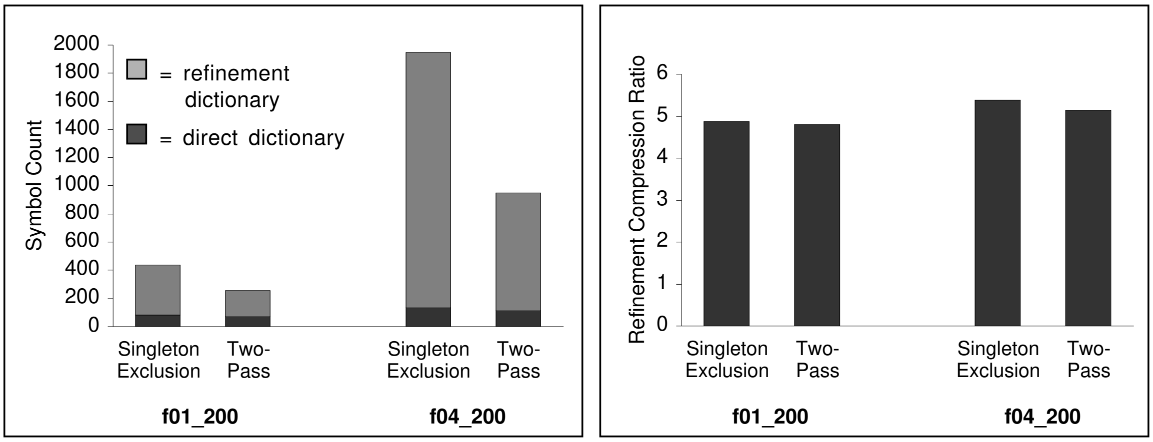

Figure 5: Dictionary sizes (left side) and refinement compression ratios (right side) for the singleton exclusion and two-pass approaches.

Next, as a numerical measure of the lossy image quality, we show in Table 3 the total number and percentage of pixels flipped in the lossy images. We notice the numbers of flips are very similar for the singleton exclusion dictionary and the two-pass dictionary.

Finally, from Table 4, we see a signicant portion of all the refinement symbols (40-50%) are perfect symbols. Moreover, 20-30% of the refinement dictionary symbols are perfect. They can be deleted from the refinement dictionary, thereby further reducing the dictionary size. In terms of compression time, omitting the arithmetic coding procedure for 40-50% of all the refinement symbols (this corresponds to 32-45% of all the extracted symbols) will significantly speed up the compression.

	one-pass	singleton excl.(% imprv.)	two-pass (% imprv.)
f01_200	54,891	50,411(8.2%)	49,189(10.4%)
f04_200	176,077	151,737(13.8%)	144,853(17.7%)

Table 2: Total number of bits in lossy compression with the 3 different methods.

5 Conclusion

In this paper we propose a two-pass symbol dictionary design technique for the JBIG2 standard. We describe a practical procedure for choosing the dictionary symbols and efficiently compressing the dictionary itself. We test the proposed technique on two CCITT standard images and find that it outperforms the one-pass dictionary formation approach by 4-9% for lossless compression. The proposed technique is also tested for lossy compression, where greater compression improvement (10-18%) is achieved. We also point out one way to significantly speed up the refinement coding process in lossy compression.

	singleton excl.		two-pass	
	# flips	% flips	# flips	% flips
f01_200	10,779	0.27%	10,863	0.27%
f04_200	51,280	1.27%	50,960	1.26%

Table 3: Number and percentage of flipped pixels in lossy compression.

	total # refinement symbols	total # perfect symbols (%)	# refinement dict symbols	# perfect refinement dict symbols (%)
f01_200	871	328(38%)	186	38(20%)
f04_200	4,175	2,009(48%)	849	269(32%)

Table 4: Number and percentage of perfect symbols in lossy compression.

References

[1] ISO/IEC JTC1/SC29/WG1 N1359. *JBIG2 Final Committee Draft*, July 1999.

[2] P. Howard, F. Kossentini, B. Martins, S. Forchhammer, W. Rucklidge, F. Ono. The Emerging JBIG2 Standard. *IEEE Trans. on Circuit and Systems for Video Technology*, pages 838–848, Vol. 8, No. 5, September 1998.

[3] D. Tompkins and F. Kossentini. A Fast Segmentation Algorithm for Bi-level Image Compression Using JBIG2. *Proceedings of the 1999 IEEE International Conference on Image Processing (ICIP)*, Kobe, Japan, October 1999.

[4] R.N. Ascher and G. Nagy. Means for Achieving a High Degree of Compaction on Scan-digitized Printed Text. *IEEE Trans. on Computers*, pages 1174–1179, November 1974.

[5] P. Howard. Lossless and Lossy Compression of Text Images by Soft Pattern Matching. In J.A. Storer and M. Cohn, editors, *Proceedings of the 1996 IEEE Data Compression Conference (DCC)*, pages 210–219, Snowbird, Utah, March 1996.

[6] I.H. Witten, A. Moffat, and T.C. Bell *Managing Gigabytes*. Morgan Kaufmann, San Francisco, California, 1999.

[7] ISO/IEC JTC1/SC29/WG1 N339. *Xerox Proposal for JBIG2 Coding*, June 1996.

[8] K. Mohuiddin, J. Rissanen, and R. Arps. Lossless Binary Image Compression Based on Pattern Matching. *International Conference on Computers, Systems and Signal Processing*, pages 447–451, Bangalore, India, December 1984.

[9] Q. Zhang and J.M. Danskin. Bitmap Reconstruction for Document Image Compression. *Proceedings of the SPIE - The International Society for Optical Engineering, vol.2916*, pages 188–99, Boston, Massachusetts, November 1996.

[10] Q. Zhang, J.M. Danskin and N.E. Young. A Codebook Generation Algorithm for Document Image Compression. *Proceedings of 1997 IEEE Data Compression Conference (DCC)*, pages 300–309, Snowbird, Utah, March 1997.