MEMORY-EFFICIENT ALGORITHMS FOR RASTER DOCUMENT IMAGE COMPRESSION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Maribel Figuera Alegre

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2008

Purdue University

West Lafayette, Indiana

To my loving parents and sisters.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my research adviser and mentor, Professor Charles A. Bouman, for granting me the honor of working under his direction. His priceless guidance and commitment have helped me face challenging research problems with confidence, enthusiasm, and perseverance. It has been a privilege to work with such a superb Faculty, and I consider myself extremely fortunate for that.

I would also like to thank Professor Jan P. Allebach for his valuable insight and remarks, and for following the progress of my research tasks. I am also deeply grateful to Professor James V. Krogmeier and to Professor Edward J. Delp for giving me the opportunity to come to Purdue, for their support, and for encouraging me to pursue a Ph.D. I would also like to express my gratitude to Professor George T.-C. Chiu for serving on my advisory committee, and for following my research progress. I would also like to thank Professor Mimi Boutin, Professor Lluís Torres, Professor Antonio Bobet, Professor Elena Benedicto, Professor Josep Vidal, and Professor Ferran Marqués for their support and encouragement.

I would like to express my gratitude to Jonghyon Yi from Samsung Electronics and to Peter Majewicz from Hewlett-Packard for their invaluable assistance and generous help during my research.

I would also like to thank all my EISL and WCRLab colleagues, and all my friends at Purdue and in Spain, especially Eri, Tarkesh, Kelly, Buyue, Oriol, Golnaz, Byungseok, Mustafa, Hasib, Animesh, Hector, Edgar, Maria, Kai-Lung, Xiaogang, Burak, Du-Yong, Guotong, Roy, Elvin, Mu, Guangzhi, Wil, Zhou, Theo, Marina, Maggie, Bich-Van, Renata, Mara, Euridice, Rosa Maria, Loly, Gerardo, Ricard, Brad, Greg, Jeremy, Linoy, Jenn, Carmen, Guillermo, Serkan, Gustavo, Cecy, Turgay, Char-

lie, Rob, Kira, Mark, Amy, Bruce, Gemma, Eva, Sílvia, Raquel, Claustre, Lluís, Xavi, Elena, Maribel, Dani, and Marc for their wonderful friendship.

Finally, from the bottom of my heart, I would like to thank my wonderful parents, Francisco and Maribel, my loving sisters, Judith and Meritxell, and my beloved friend Gbile, for their unconditional support and love, and for helping me become who I am.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABBREVIATIONS

bpp	bits per pixel
COS	cost optimized segmentation
CR	compression ratio
dpi	dots per inch
DSC	dynamic symbol caching
JBIG	joint bi-level image experts group
JPEG	joint photographic experts group
LRU	least recently used
MCC	mixed content compression
MMR	modified modified READ
MMSE	minimum mean square error
MRC	mixed raster content
PM&S	pattern matching and substitution
MSE	mean square error
PWXOR	prescreened weighted exclusive-OR
READ	relative element address designate
SPM	soft pattern matching
XOR	exclusive-OR
WAN	weighted and not
WXOR	weighted exclusive-OR

ABSTRACT

Figuera Alegre, Maribel Ph.D., Purdue University, August 2008. Memory-Efficient Algorithms for Raster Document Image Compression . Major Professor: Charles A. Bouman.

In this dissertation, we develop memory-efficient algorithms for compression of scanned document images. The research is presented in two parts. In the first part, we address key encoding issues for effective compression of multi-page document images with JBIG2. The JBIG2 binary image encoder dramatically reduces bit rates below those of previous encoders. The effectiveness of JBIG2 is largely due to its use of pattern matching techniques and symbol dictionaries for the representation of text. While dictionary design is critical to achieving low bit rates, little research has been done in the optimization of dictionaries across stripes and pages. In Chapter 1, we propose a novel dynamic dictionary design that substantially reduces JBIG2 bit rates, particularly for multi-page documents. This dynamic dictionary updating scheme uses caching algorithms to more efficiently manage the symbol dictionary memory. Results show that the new dynamic symbol caching technique reduces the bit rate by between 12% and 53% relative to that of the best previous dictionary construction schemes for lossy compression when encoding multi-page documents. We also show that when pages are striped, the bit rate can be reduced by between 2% and 25% by adaptively changing the stripe size. In addition, we propose a new pattern matching criterion that is robust to substitution errors and results in both low bit rates and high encoding speeds.

In the second part, we propose a hardware-efficient solution for compression of raster color compound documents. Effective compression of a raster compound document comprising a combination of text, graphics, and pictures typically requires

that the content of the scanned document image be first segmented into multiple layers. Then, each layer is compressed as a separate image using a different coding method. Layer-based compression formats such as the mixed raster content (MRC) achieve very low bit rates while maintaining text and graphics quality. However, MRC-based compression algorithms typically require large memory buffers, and are therefore not easily implemented in imaging pipeline hardware. In Chapter 2, we propose a hardware-friendly block-based lossy document compression algorithm which we call mixed content compression (MCC) that is designed to work with conventional JPEG coding using only an 8 row buffer of pixels. MCC uses the JPEG encoder to effectively compress the background and picture content of a document image. The remaining text and line graphics in the image, which require high spatial resolution, but can tolerate low color resolution, are compressed using a JBIG1 encoder and color quantization. To separate the text and graphics from the image, MCC uses a simple mean square error (MSE) block classification algorithm to allow a hardware efficient implementation. Results show that for our comprehensive training suite, the average compression ratio achieved by MCC was 60:1, but JPEG only achieved 35:1. In particular, MCC compression ratios become very high on average (82:1 versus 44:1) for mono text documents, which are very common documents being copied and scanned with all-in-ones. In addition, MCC has an edge sharpening side-effect that is very desirable for the target application.

1. CACHE-EFFICIENT DYNAMIC DICTIONARY DESIGN FOR MULTI-PAGE DOCUMENT COMPRESSION WITH JBIG2

1.1 Introduction

Many document imaging applications such as document storage and archiving, scan-to-print, scan-to-email, image coding, internet fax, wireless data transmission, print spooling, and teleconferencing require the use of raster scanned representations of documents. Raster documents have the advantages of being simple and flexible in applications such as archiving or document exchange; and since they are already in raster format, they can be easily rendered on printing and display devices. However, documents in raster format can be quite large, with a full-color 600 dots per inch (dpi) letter-sized document requiring approximately 100 Megabytes per page in uncompressed form. Therefore, effective raster document compression is very important.

In fact, document compression differs quite substantially from more traditional natural image compression because the synthetically created content of a typical document is quite different in nature than the content of natural images, and the types of distortion that are most visually disturbing are also quite different. Typical document compression formats such as the mixed raster content (MRC) [1] achieve very low bit rates while maintaining text and graphic quality by using a binary mask layer to encode the high-resolution transitions that are essential to text and graphics quality.

The most basic MRC approach separates the compound document, comprising a combination of images, text, and graphics, into a binary mask layer, a background layer, and a foreground layer. The binary mask layer contains high-resolution text; the

background layer typically contains low-resolution pictures and photographic images; and the foreground layer typically contains the colors of text and graphic symbols. Each layer is compressed separately by a suitable encoder that yields the best trade-off between bit rate and quality. In practical situations, the binary mask layer of an MRC compressed document often represents a substantial portion of the total bit rate. Hence, effective encoding of the binary mask layer of an MRC document can significantly reduce the overall size of the compressed compound document.

Fortunately, over the last decade a number of new binary image compression schemes have been developed with dramatically improved compression. The Joint Bi-level Image Experts Group 2 (JBIG2) [2] binary image compression standard is perhaps the most widely used example of a symbol dictionary based method for binary image compression. JBIG2 offers among the highest compression rates for binary images available today, and it is the first international standard that supports both lossless and lossy compression of multi-page binary documents [2]. JBIG2 lossless compression can be used for those applications that require exact bit representation of the original image, and JBIG2 lossy compression can produce a binary image which is visual equivalent to the original while dramatically reducing the bit rate.

JBIG2 compresses binary images much more effectively than existing standards such as T.4 [3], T.6 [4], T.82 (JBIG1) [5] and T.85 [6], and it typically can increase the compression ratio by a factor of 3 to 8 over TIFF G3 or TIFF G4 techniques [2]. The increased compression of JBIG2 is a result of the fact that it represents the compressed document as a series of black symbols (represented by the binary value 1) on a white background (represented by the binary value 0). These black symbols are extracted as connected components in the binary image, and typically these connected components represent text symbols in the document.

In order to limit memory requirements and complexity, the document is first broken into horizontal stripes. Then for each stripe, the connected components are extracted from the image. Then the connected components are grouped together into similarity classes, and a typical representative of each similarity class is stored in a

dictionary. In this way, the symbol dictionary for each stripe is composed of all the distinct symbols contained in that stripe. Finally, the binary content of a stripe is then encoded by specifying a symbol dictionary, and then representing each connected component in the stripe by both a dictionary entry and its position in the stripe.

When the number of stripes is large, it is particularly important to reuse dictionary symbols from one stripe to the next. The number of stripes must be large for long multi-page documents because each page must have at least one or two stripes [2,7]. In addition, it may be advantageous to use a large number of stripes in order to conserve memory in the encoder implementation. In these cases, the dictionaries used for each stripe will typically share symbols, so retransmission of the dictionary for each stripe is very inefficient. While some literature discusses methods for the construction of a symbol dictionary from a single stripe or page [8–15], few of the efforts have addressed how the dictionary should be updated and re-used across stripes and pages [15].

In this chapter, we address three aspects of a JBIG2 encoder design which are essential to good coding performance: symbol dictionary design, pattern matching, and page striping. First, we present three new methods for updating the dictionary for each stripe. In particular, we propose a novel memory efficient dynamic dictionary design which we call Dynamic Symbol Caching that substantially reduces the bit rate when encoding multi-page documents [16]. Dynamic Symbol Caching shares dictionary symbols across stripes and uses caching algorithms to more efficiently manage the limited symbol dictionary memory. Second, we present a new pattern matching criterion which we call Prescreened Weighted Exclusive-OR (PWXOR) that is accurate and reduces the pattern matching time compared to previous criteria. Third, we introduce a page striping method which we call adaptive striping that further reduces the bit rate by optimizing the locations where the page is striped and adaptively changing the stripe size.

In order to experimentally evaluate our algorithms, we use several multi-page test documents that contain a variety of content. Our experimental results show that for lossy compression of multi-page documents using two stripes per page and the

largest possible (1 Mbyte [7]) JBIG2 dictionary size, our proposed Dynamic Symbol Caching method achieves a 32% bit rate reduction relative to that of a static symbol dictionary scheme, and a 19% bit rate reduction relative to that of the previous best dynamic scheme. The relative bit rate reduction achieved by our method relative to previous methods increases as each page is divided into more stripes or as more pages are encoded in a single JBIG2 bitstream. We will also show that, when using multiple stripes per page, the bit rate is reduced by using adaptive striping. Finally, we compare our JBIG2 encoder with several existing JBIG2 commercial encoders in terms of bit rate at similar image quality.

The remainder of this chapter is organized as follows. In Section 1.2, we briefly review previous symbol dictionary formation techniques for JBIG2 encoding. In Sections 1.3 and 1.4, we propose three dynamic symbol dictionary schemes for updating the dictionary across stripes and pages and introduce a new pattern matching criterion. Section 1.5 presents a method for reducing the bit rate by adaptively changing the stripe size. Section 1.6 shows some experimental results and the concluding remarks are presented in Section 1.7.

1.2 Review of Prior Work

The JBIG2 standard supports three data classes: text, generic, and halftone [2]. What makes JBIG2 special is the text class, which employs a symbol dictionary to greatly improve compression of repeating symbols in a document. The generic class is conventional facsimile coding using a basic bitmap coder such as an MMR [4] coder or an arithmetic coder. Throughout this chapter a text symbol refers to a symbol encoded using the text class, and a generic symbol refers to a symbol encoded using the generic class.

Text images typically contain many repeated symbols. JBIG2 exploits the similarities between multiple stripes by using character-based pattern matching techniques. JBIG2 supports two modes: pattern matching and substitution (PM&S) [17] and

soft pattern matching (SPM). The PM&S method works by replacing the original symbol with a representative matching symbol from the symbol dictionary. Since the original and matching symbols differ slightly, PM&S encoding is lossy. Alternatively, the SPM method works by losslessly encoding the original symbol using a context-based arithmetic coder in which the matching dictionary symbol provides the context for the coder [8, 18]. Note that in PM&S-based systems, if the original symbol is replaced with a matching dictionary symbol that represents a different character, a substitution error occurs. So for example, an "o" substituted by a "c", or a non-bold face "F" substituted by a bold face "F" is referred to as a substitution error. Substitution errors are highly undesirable, so PM&S-based systems must be designed so that substitution errors are very unlikely. Fig. 1.1 illustrates the JBIG2 encoding and decoding process. Note that we only use text and generic classes since these are sufficient for the encoding of any binary image.

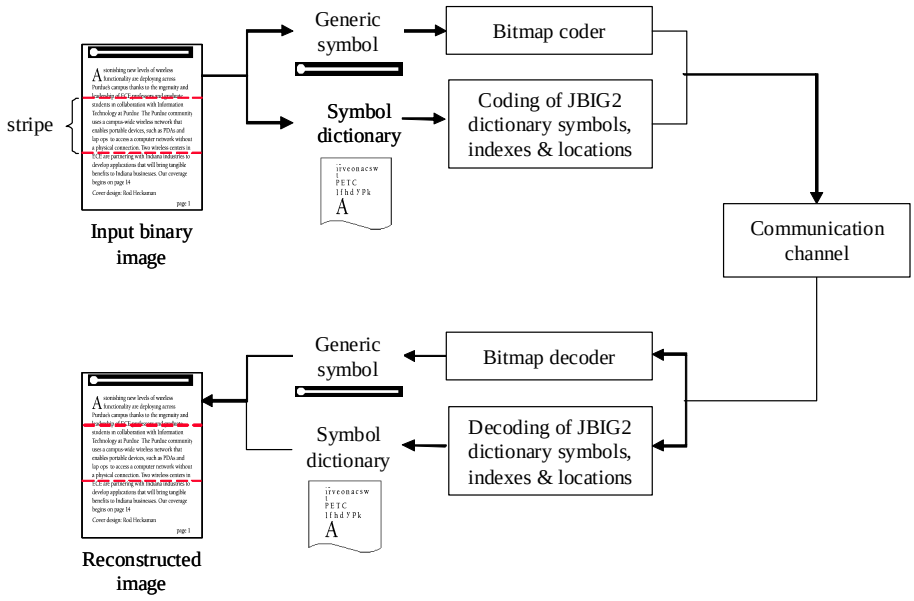


Fig. 1.1. JBIG2 encoding and decoding process. The encoder collects a set of dictionary symbols and encodes the text symbols in each stripe by reference to the dictionary symbols. Symbols classified as generics are coded using a basic bitmap coder similar to that used for traditional facsimile coding.

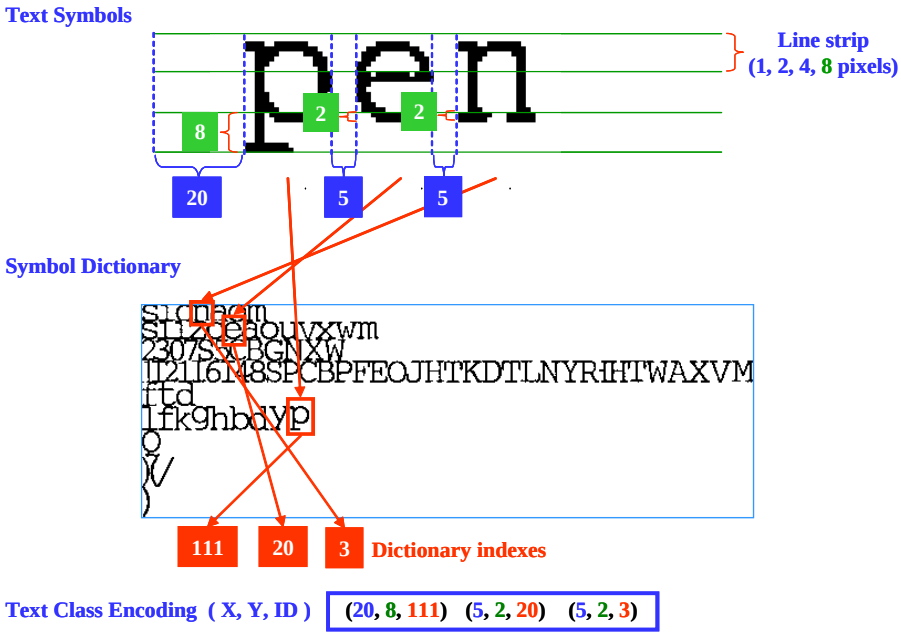


Fig. 1.2. Encoding of the text symbols “p”, “e”, and “n” using the text class. Each text symbol in a stripe is coded by giving its position on the page relative to a previous coded text symbol and the index of its matching symbol in the associated dictionary.

In JBIG2 encoding, each text symbol in a stripe is coded by giving its position on the page and the index of its matching symbol in the associated dictionary. Fig. 1.2 illustrates the encoding of text symbols using the text class, and Fig. 1.3 illustrates the encoding of a JBIG2 symbol dictionary. The JBIG2 standard [2] allows the designer of the JBIG2 encoder to choose between arithmetic coding (the MQ coder), or a combination of Huffman and MMR (Modified Modified READ, T.6) coding. The dictionary symbols may be compressed using an MMR coder or an arithmetic coder; the numerical data such as the dictionary symbol index, the vertical and horizontal position of the symbol in the page, and the width and height of the dictionary symbol may be compressed using either Huffman or arithmetic coding. The position and size of the text symbol are usually coded relative to a previously encoded text symbol. Use of the arithmetic coding option results in the best possible compression ratio at the expense of higher encoder complexity and higher decoding time, while the

use of Huffman/MMR coding enables the highest decoding speeds and lower encoder complexity.

Fig. 1.3. Encoding of a JBIG2 symbol dictionary composed of 23 symbols using Huffman/MMR coding. Dictionary symbols that have the same height are grouped to form a heightclass. Note that the height of the symbols in a heightclass is encoded relative to the height of its previous heightclass. Similarly, the width of a symbol is coded relative to a previously coded symbol in the same heightclass.

1.2.1 Previous Symbol Dictionary Designs

When coding the current stripe, the encoder can reuse some of the dictionary symbols from previous stripes. So the design of the symbol dictionary can have an important impact on the bit rate and reconstructed image quality achieved by a JBIG2 encoder.

Many JBIG2 dictionary construction techniques have been proposed such as the one-pass [8], singleton exclusion [9, 10], class-based [11, 12], tree-based [12, 13], and modified-class [14, 15] symbol dictionary designs. However, we refer to all these schemes as static symbol dictionary designs because they only describe how the dictionary is constructed from a single stripe or page, but they do not specify how the dictionary should be updated and re-used across stripes and pages.

The simplest method for encoding multiple stripes of a document is the Independent Static method. This method creates a completely new dictionary for each new stripe, without taking into account the symbols that were used to encode previous stripes. The Independent Static method results in a high bit rate since at each stripe a new dictionary must be encoded. It is important to note that symbols may often be repeated from one stripe to another. Therefore, the dictionaries used by different stripes may redundantly store the same symbol. This redundancy has two disadvantages. First, the redundant encoding of these symbols increases computation. Second, the retransmission of these redundant symbols increases the overall bit rate.

The dynamic scheme proposed by Ye and Cosman in [15, 19] takes advantage of the fact that symbols may be repeated within consecutive stripes. We refer to this scheme as the Local Dynamic method. At each new stripe, the Local Dynamic method constructs a new dictionary in the following manner. First, it only retains symbols from the previous dictionary that are contained in the current stripe. This can be done by sending a one bit flag for each dictionary symbol to indicate if the symbol is to be retained (bit flag value of 1) or discarded (bit flag value of 0) from the dictionary. Next, any new symbols from the current stripe are added to the new dictionary. The disadvantage of the Local Dynamic method is that some symbols that are already stored in the dictionary are discarded, even though they may be useful in future stripes. Typically, the bit rate achieved with a Local Dynamic method is lower than that of an Independent Static method because the Local Dynamic method more efficiently reuses symbols from previous stripes.

1.3 Dynamic Symbol Caching Algorithm

In this section, we propose three new dictionary schemes for updating the symbol dictionary across stripes which substantially reduce the bit rate relative to that of previous dictionary designs. We refer to these three schemes as the Global, Optimal

Dynamic, and Dynamic Symbol Caching schemes [16]. Table 1.1 summarizes each of these dictionary updating schemes, along with their strengths and weaknesses.

The Global method uses a single dictionary to encode an entire multi-page document. Therefore, it contains all the symbols necessary to encode all the stripes in the document. The Global method results in a low bit rate. However, it is not practical when there are memory limitations on the encoder or decoder since it requires that the entire document be buffered to extract the dictionary before the encoding process can start. Moreover, the Global method also increases computation time because the symbol matching process requires a linear search through a much larger dictionary.

The Optimal Dynamic method adds a symbol to the dictionary at exactly the time the symbol is needed to encode the current stripe and removes it at exactly the time it is no longer needed. This method is optimal in the sense that each dictionary symbol is added to the dictionary only once and the dictionary size is minimized for each stripe. Once again, this algorithm is not practical since it requires that the entire document be buffered and processed before encoding. However, it serves as a useful baseline to gauge the effectiveness of the other methods.

The Dynamic Symbol Caching scheme also recursively updates the dictionary for each stripe, but it differs from previous methods in that it uses a caching algorithm to determine which symbols to retain and discard from the dictionary. In practice, the size of a JBIG2 dictionary must be limited for two reasons. First, the JBIG2 standard limits the size of dictionaries to a maximum of 1 Mbyte [7]. This 1 Mbyte limit determines the memory that a decoder must have available in order to be able to decode any standard-compliant JBIG2 bitstream. However, in many applications it may be desirable to further limit the maximum dictionary size in order to reduce the memory requirements of the JBIG2 encoder. In either case, if the dictionary is full (i.e. it is at its maximum allowed size), then we must decide which existing symbols to retain and which to discard in order to make room for new symbols. Our approach discards those dictionary symbols that have the lowest probability of being referenced again and tries to retain those dictionary symbols that have a good

Table 1.1

Dictionary updating schemes. The Independent Static scheme and the Local Dynamic scheme are dictionary updating schemes already described in the literature. The Global, Optimal Dynamic, and Dynamic Symbol Caching schemes are introduced in this chapter.

Name	Description	Attributes	Comments
Independent Static	A new static dictionary is generated for each new stripe. The dictionary contains exactly those symbols in the current stripe.	Dictionary size: small Raster buffer size: single stripe Bit rate: high Coding time: low Update: new for each stripe	Does not reuse any dictionary symbols from previous stripes.
Local Dynamic	The dictionary is recursively updated with each new stripe. The dictionary only retains symbols that appear in the current stripe and adds any new symbols from the current stripe.	Dictionary size: small Raster buffer size: single stripe Bit rate: medium Coding time: low Update: recursive	Removes previous dictionary symbols if they do not appear in the current stripe.
Global	Single static dictionary composed of all symbols in the multi-page document.	Dictionary size: unlimited Raster buffer size: entire multi-page document Bit rate: low Coding time: high Update: static	Not practical because it requires buffering of the entire document.
Optimal Dynamic	The dictionary is recursively updated with each new stripe. The dictionary for each stripe is composed of symbols from the current stripe and symbols from previous stripes that will appear in future stripes.	Dictionary size: unlimited Raster buffer size: entire multi-page document Bit rate: low Coding time: high Update: recursive	Removes dictionary symbols only when they become obsolete. Not practical because it requires buffering of the entire document.
Dynamic Symbol Caching	The dictionary is recursively updated with each new stripe. The dictionary for each stripe is composed of symbols from the current stripe and most recently used symbols from previous stripes.	Dictionary size: user selectable Raster buffer size: single stripe Bit rate: low Coding time: medium Update: recursive	Removes least recently used dictionary symbols when dictionary size exceeds user selectable limit.

likelihood of being referenced again in the near future. There exist many caching algorithms and variations of those that can be used to predict when a dictionary symbol is more likely to be referenced again. For our JBIG2 encoder we use the least recently used (LRU) algorithm, which discards the least recently used items first and is one of the most effective algorithms for memory caching [20, 21]. The LRU algorithm is implemented by allocating an integer variable, referred to as a key, for each distinct dictionary symbol. Each key keeps track of the last time a symbol was used. To do this, the stripes in the document are numbered in order starting at 0, and each time a symbol is used, the key for that symbol is set to the current stripe number. When it is necessary to discard a symbol, the symbols with the smallest key values are discarded first. Using this method, symbols that were used least recently are discarded first.

Fig. 1.4 illustrates the flow chart of the Dynamic Symbol Caching algorithm. The Dynamic Symbol Caching dictionary is constructed and updated in the following manner. Assume that a multi-page document is divided into P stripes. Let stripe number k, denoted by $stripe\_img_k$ $(0 \leq k < P)$ in Fig. 1.4, be the current stripe. Let D_{k-1} be the previous symbol dictionary. First, we create a new dictionary D_k composed of all previous dictionary symbols and any new symbols in the current stripe. Note that if the current stripe is the first stripe in the document (i.e. $k = 0$), then there will not be any previous dictionary symbols. Next, for each dictionary symbol in D_k that is referenced in the current stripe, we set its key to k to indicate that the symbol last appeared in stripe number k. Then, we compute the current dictionary memory size using equation (1.1) below, and we check if this value exceeds the maximum allowed memory size, M. If the memory limit is exceeded, then we remove symbols with the smallest keys first until the memory constraint is met.

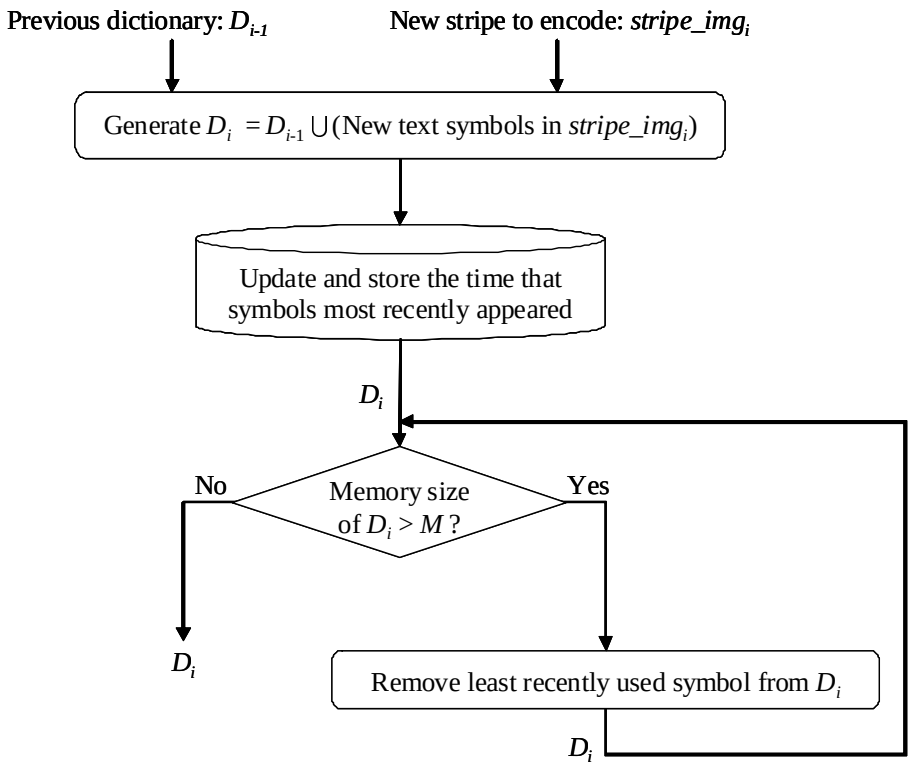


Fig. 1.4. Flow chart of the Dynamic Symbol Caching algorithm. The least recently used symbols are removed from the dictionary whenever the dictionary memory reaches its limit. This algorithm optimizes symbol dictionary memory usage and allows efficient encoding of binary documents with much smaller JBIG2 encoder and decoder buffers. It is practical and allows for low bit rates.

The memory usage of the symbol dictionary (MSD), when using Huffman coding, is given by [7]

$$\begin{aligned} MSD &= \text{fixed component} + \text{per symbol component} \\ &= c \cdot 8K + \sum_{i=1}^{N} \frac{32+\text{round}(W(i)\times H(i))}{8} \text{ bytes} \end{aligned} \tag{1.1}$$

where $c = 0$ if default Huffman tables are used, which is our case, otherwise, if custom Huffman tables are used, then $c = 1$. N is the total number of symbols in the decoded dictionaries being retained (i.e. in those decoded dictionaries for which the retain flag is set to 1), round$(\cdot)$ rounds up to the next multiple of 32 bits, $W(i)$ and $H(i)$ are the symbol width and height, respectively, and 32 bytes are the overhead per symbol.

From equation (1.1) we can see that the memory cost associated with adding a new symbol is $\frac{32+\text{round}(W\times H)}{8}$ bytes, where W and H are the width and height of the symbol, respectively.

In the construction of the dictionary, we use the PM&S pattern matching based coding system, although the three algorithms we propose in this section can also be applied to a SPM-based JBIG2 system. Each stripe is scanned in raster order and symbols are extracted based on an 8-point neighborhood criterion. For the extraction of the symbols, we use the fast boundary tracing algorithm proposed in [22]. Each new extracted text symbol is matched against the symbols in the current dictionary. Only symbols that differ a maximum of 2 pixels in width and/or height are compared. This selection process is called screening. If a match is found, then the extracted symbol is encoded using the symbol's location offset relative to preceding symbols, and an index pointing to the dictionary symbol to which the extracted symbol was matched. If a match is not found, then the new extracted symbol is added to the current symbol dictionary.

1.4 Prescreened Weighted Exclusive-OR Pattern Matching Algorithm

Pattern matching consumes the most time in JBIG2 encoding, and the matching criterion is critical to achieving high reconstructed image quality and low bit rates. Pattern matching is addressed in [23,24] where the authors use a cross-entropy approach to measure the similarity between two symbols. A number of pattern matching techniques have been described in the literature that operate on an error map, which is the bitwise exclusive-OR between the extracted symbol and a dictionary symbol [9, 25–27]. The simplest method is the Hamming distance or exclusive-OR (XOR), which is calculated by summing the number of black pixels in the error map. The XOR operator is a fast matching criterion; however, it is not robust to substitution errors. To solve this problem, Pratt et. al. proposed the weighted exclusive-OR

(WXOR) [26], which computes a weighted sum of pixels in the error map, where error pixels that occur in clusters are weighted more highly. A similar method to WXOR is the weighted and-not (WAN) of Holt and Xydeas [27], which distinguishes black-to-white errors from white-to-black ones. WXOR and WAN are robust to substitution errors and allow for low bit rates. However, they are more computationally expensive than XOR.

We introduce a new matching criterion, the Prescreened Weighted Exclusive-OR (PWXOR), that combines XOR and WXOR to achieve computational speed close to XOR while retaining a bit rate comparable to WXOR [16]. We do this by using the XOR criterion first to quickly identify matches that are either very good or very bad. In this case, only matches that are uncertain need to be processed by the more accurate WXOR criterion. By using the XOR criterion as a screening mechanism, we are able to substantially reduce the number of WXOR operations.

Fig. 1.5 illustrates the flow chart of the XOR, WXOR, and PWXOR pattern matching criteria. Note that only symbols that are classified as text symbols are matched against symbols in the dictionary. In our implementation, a text symbol is an extracted symbol such that the maximum of its width and height in pixels is less or equal than a threshold T_s. We refer to threshold T_s as the generic size threshold. Using this method, larger symbols are encoded as generics rather than being stored in a dictionary.

The PWXOR criterion works in the following manner. Let e denote an error map with size $N \times M$ pixels, and let $e(i, j)$ denote a pixel at position (i, j) in the error map, where i and j indicate the row and column, respectively. First, we align the extracted symbol and the dictionary symbol by aligning their respective centroids, and then we compute the error map, e, by applying the bit-wise "xor" operation between corresponding pixels in the aligned symbols. Next, we compute the XOR distance

using equation (1.2) below. The XOR distance, d_{XOR}, measures the percentage of pixels set in the error map and it is given by

$$d_{XOR} = \frac{100}{NM} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} e(i,j) \tag{1.2}$$

The screening method determines if the WXOR distance between the two symbols needs to be computed. If the XOR distance between the two symbols is greater than a predetermined mismatch threshold, T_2, then the match is rejected without further consideration, while if the distance is less than a lower threshold $T_1 < T_2$, the match is accepted and the algorithm ends. On the other hand, if the XOR distance lies between T_1 and T_2, we compute the WXOR distance using equation (1.3) below. The WXOR distance differs from the XOR distance in that each error pixel in the error map contributes an additional amount equal to the number of error pixels in its 3×3 neighborhood. As a result, error pixels that occur in a cluster are more significant. The WXOR distance is given by

$$d_{WXOR} = \frac{100}{NM} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} w_{ij}\ e(i,j) \tag{1.3}$$

where the weights w_{ij} $(1 \leq w_{ij} \leq 9)$ are computed using equation (1.4) below.

$$w_{ij} = \sum_{k=-1}^{1} \sum_{l=-1}^{1} e(i+k, j+l) \tag{1.4}$$

Finally, if the WXOR distance is less than a predetermined threshold, T_3, the match is accepted, otherwise the match is rejected. For each matching criterion an appropriate threshold is determined. The selection of each threshold is important because it determines whether substitution errors will occur in the reconstructed images. Higher thresholds may produce substitution errors, while lower thresholds may not detect a match between two symbols that represent the same character. The thresholds T_1 and T_2 must be chosen to balance two competing objectives. On one hand, the interval $[T_1, T_2]$ must be sufficiently large that the final result is nearly the same as what would result if only the WXOR criterion were used, but on the other

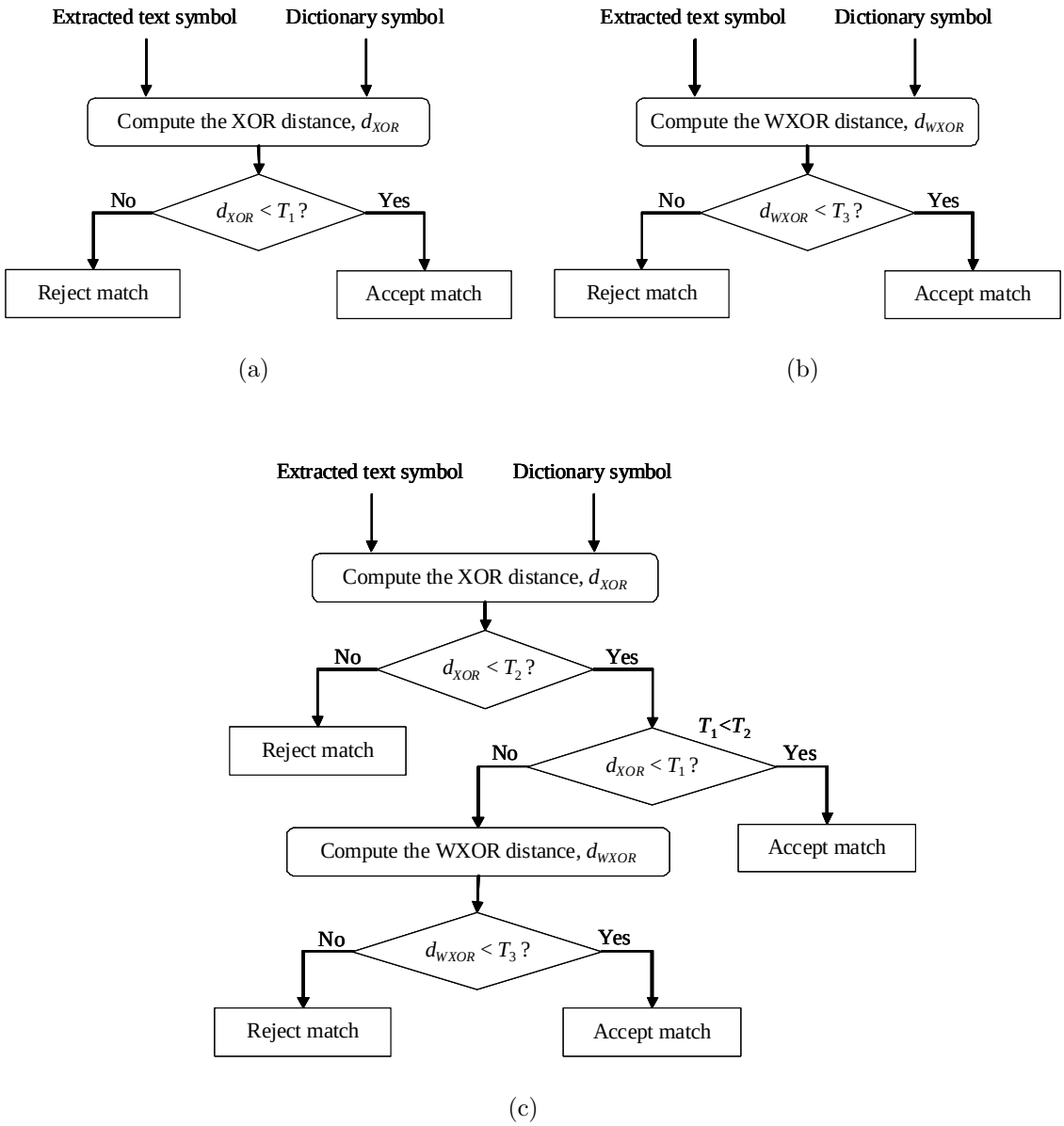


Fig. 1.5. Flow charts of the three pattern matching criteria: (a) XOR (b) WXOR (c) PWXOR. PWXOR uses the XOR criterion first to quickly identify matches that are either very good or very bad. Only matches that are uncertain are processed by the more accurate WXOR criterion. PWXOR achieves computational speed close to XOR while retaining a bit rate comparable to WXOR. For all three criteria, only text symbols are matched against dictionary symbols. In our implementation, a text symbol is an extracted symbol such that the maximum of its width and height in pixels is less or equal than a threshold T_s.

hand, the interval $[T_1, T_2]$ must be sufficiently small that the number of evaluations of the WXOR operator is minimized. The default thresholds for our JBIG2 encoder are $T_1 = 6$, $T_2 = 21$, $T_3 = 27$, and $T_s = 600$.

1.5 Adaptive Striping

A fixed stripe size is commonly used when splitting an entire document into two or more stripes per page. However, while this scheme is simple, it may cause symbols falling on the boundaries between stripes to be split into two or more connected components. These resulting components will then be treated as new symbols, which will be added to the dictionary, thus increasing the bit rate.

We can reduce the number of split components by adapting the height of each stripe to the content of the page. Consider a page which is N pixels high by M pixels wide that we would like to split into n stripes. Then, the nominal locations of the stripe breaks is given by $k \left\lfloor \frac{N}{n} \right\rfloor$ where $1 \leq k \leq n-1$, and where each stripe break is the last row of a stripe. In adaptive striping, we vary the location of these break points by searching in the range $\left[k \left\lfloor \frac{N}{n} \right\rfloor - 25, k \left\lfloor \frac{N}{n} \right\rfloor + 25\right]$ and select the row with the minimum number of black-to-white transitions. The number of black-to-white transitions is computed using equation (1.5) below, where $b(i, j)$ is the value of the pixel located at row i and column j from the top-left of the page. This strategy selects stripe locations that produce fewer broken symbols, and thereby reduces the required bit rate.

$$C(i) = \sum_{j=1}^{M-1} b(i, j-1) \left[1 - b(i, j)\right], \qquad i \in \left[k \left\lfloor \frac{N}{n} \right\rfloor - 25, k \left\lfloor \frac{N}{n} \right\rfloor + 25\right] \tag{1.5}$$

1.6 Experimental Results

For our experiments, we used an image database consisting of 162 pages of text and graphics. These pages come from a variety of sources, including books and

magazines. First, the multi-page documents were scanned at 300 dpi and 24 bits per pixel using the HP LaserJet 4345 multi-function printer (MFP). Then, the scanned document images were segmented using the COS [28] algorithm, and we used the resulting binary masks as our test documents. In general, the scanned documents contain some skew resulting from mechanical misalignment of the page. The full test suite is shown in Table 1.2, and Fig. 1.6 illustrates a sample of these test images. The majority of our experimental results are given in terms of bit rate in bits per pixel (bpp).

Table 1.2
Full test suite consisting of 162 pages of text and graphics.

Test Document	# of Pages	BMPs Total Size (bytes)	Page Size / Pixel Dimensions
Purdue Wavelinks Magazine	23	24,142,226	8.533 in×10.933 in / 2560×3280
Photoshop CS Manual	46	48,578,852	8.5 in×11in / 2550×3300
ITU-T T.89 Standard	18	19,009,116	8.5 in×11 in / 2550×3300
Time Magazine	28	29,569,736	8.5 in×11 in / 2550×3300
Purdue ECE Impact Magazine	28	29,569,736	8.5 in×11 in / 2550×3300
The Data Compression Book	19	20,065,178	8.5 in×11 in / 2550×3300

Table 1.3 lists the threshold values used for each of the three matching criteria. These thresholds were used throughout all our experiments. In our implementation, we selected a generic size threshold $T_s = 600$. The matching thresholds for the XOR, WXOR, and PWXOR criteria were then determined from the full test suite so that they produced the lowest bit rate with no visible substitution errors.

For our first experiment, we encoded the binary image illustrated in Fig. 1.6(a) using the XOR, WXOR, and PWXOR criteria. This image is a page of the *Purdue Wavelinks Magazine* test document. It contains 3971 symbols, has dimensions of 2560×3280 pixels, is sampled at a resolution of 300 dpi, and has an uncompressed file size of 1 Mbyte.

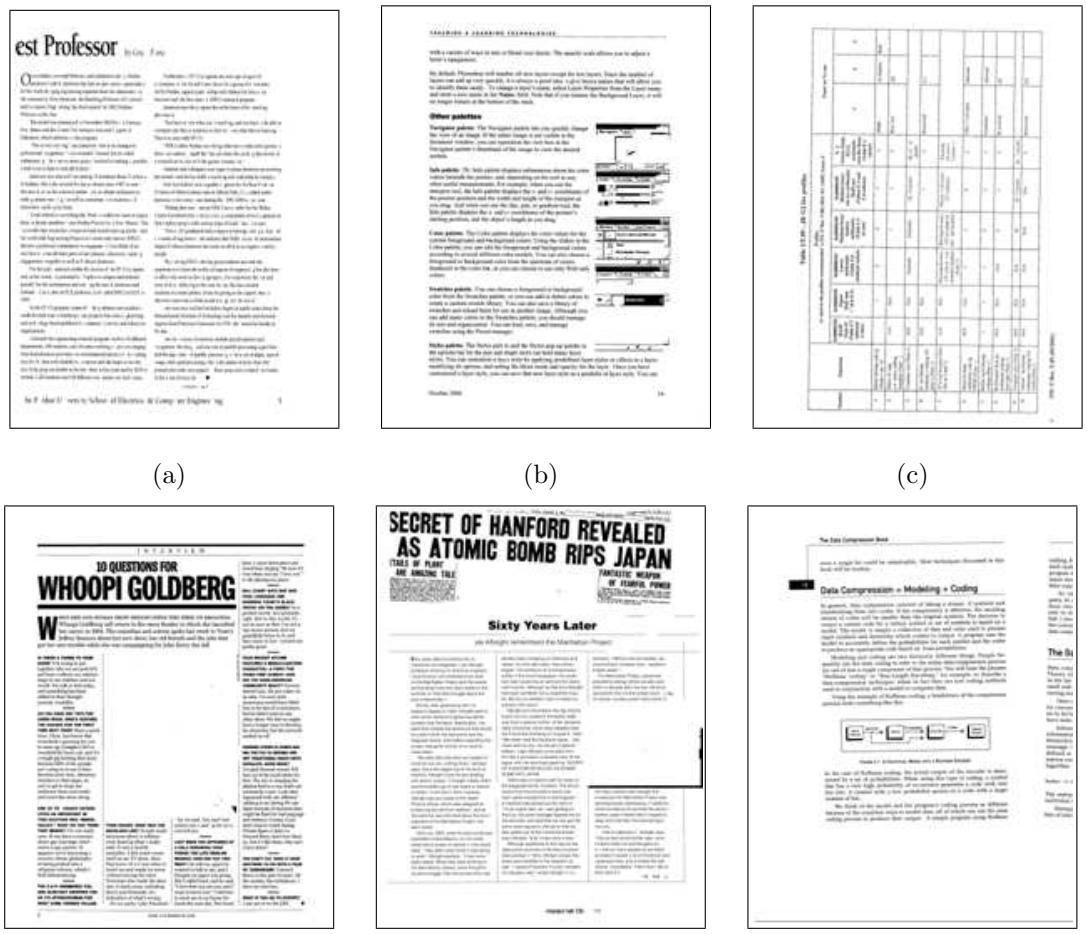


Fig. 1.6. Sample test pages taken from (a) Purdue Wavelinks Magazine (b) Photoshop CS Manual (c) ITU-T T.89 Standard (d) Time Magazine (e) Purdue ECE Impact Magazine (f) The Data Compression Book. These binary images are approximately letter-sized, are sampled at a resolution of 300 dpi, and have an uncompressed file size of 1 Mbyte.

Table 1.3

Generic size threshold (T_s) and matching thresholds (T_1, T_2, T_3) for the XOR, WXOR, and PWXOR criteria. Each matching threshold was determined from the full test suite so that it produced the lowest bit rate with no visible substitution errors. These thresholds were used throughout all our experiments.

Matching Criterion	T_s	T_1	T_2	T_3
XOR	600	6	-	-
WXOR	600	-	-	27
PWXOR	600	6	21	27

Table 1.4

Comparison of the XOR, WXOR, and PWXOR matching criteria. Each criteria uses a matching threshold that produces the lowest bit rate with no visible substitution errors in the reconstructed images. Note that PWXOR is the fastest in finding a match and achieves very nearly the same bit rate as WXOR.

Matching Criterion	Bit Rate (bpp)	Pattern Matching Time (seconds)	Dictionary Size		Average # of Matching Operations	
			# of Symbols	Bytes	XOR	WXOR
XOR	0.021410	0.81	531	40,553	21.57	0
WXOR	0.016386	1.56	310	29,798	0	9.83
PWXOR	0.016401	0.55	311	29,803	9.85	1.64

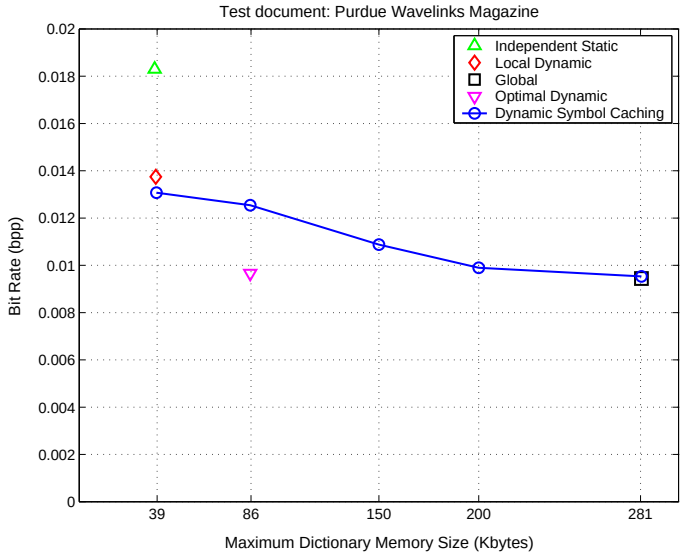
Table 1.4 compares the XOR, WXOR, and PWXOR criteria in terms of bit rate, pattern matching time, dictionary size, and average number of matching operations per symbol. We can that PWXOR is the fastest in finding a match among the three criteria. PWXOR also achieves very nearly the same bit rate as that of WXOR, but PWXOR only requires 35% of the matching time of WXOR. XOR requires the highest bit rate and is also the slowest in finding a match. This is because XOR results in a much larger symbol dictionary than WXOR and PWXOR, and a larger symbol

dictionary requires both a higher bit rate and a longer linear search through the larger dictionary. From Table 1.4, we can also see that by using the XOR criterion as a screening mechanism, PWXOR is able to reduce the number of WXOR operations by 83% as compared to those of WXOR.

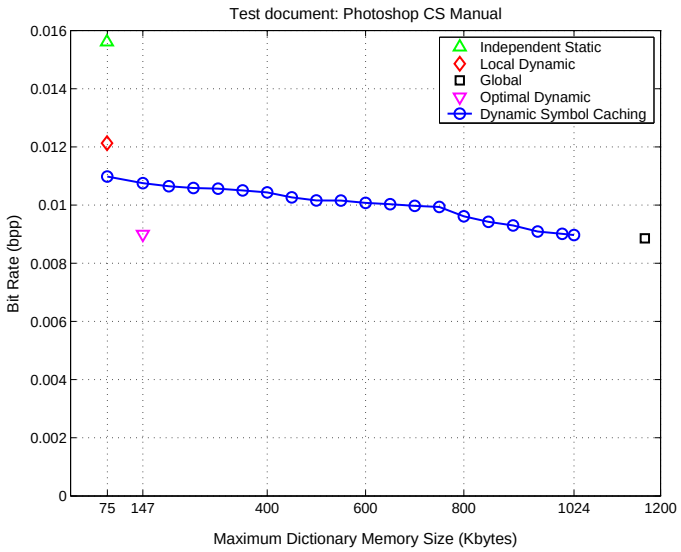
Next, we encoded the *Purdue Wavelinks Magazine* and *Photoshop CS Manual* multi-page test documents using the five algorithms listed in Table 1.1. Fig. 1.7(a) illustrates the bit rate as a function of varying dictionary memory size in kilobytes for the *Purdue Wavelinks Magazine*. We can see that when the dictionary memory is large, Global and Dynamic Symbol Caching achieve a similar bit rate to that of Optimal Dynamic. To achieve this bit rate, Global and Dynamic Symbol Caching require approximately 281 Kbytes of dictionary memory, while Optimal Dynamic only requires a maximum of 86 Kbytes. With a dictionary size of approximately 86 Kbytes, Dynamic Symbol Caching requires nearly 30% higher bit rate than Optimal Dynamic. However, the Optimal Dynamic method is not practical because it requires that the entire multi-page document be buffered before the encoding process can start. With a larger dictionary size of approximately 281 Kbytes, Dynamic Symbol Caching reduces the bit rate by 48% relative to that of Independent Static and by 31% relative to that of Local Dynamic. Even with a smaller dictionary size of approximately 39 Kbytes, Dynamic Symbol Caching achieves 29% lower bit rate than Independent Static and 5% lower bit rate than Local Dynamic.

Fig. 1.7(b) illustrates the bit rate as a function of varying dictionary memory size in kilobytes for the *Photoshop CS Manual*. We again see that when the dictionary memory is large, Global and Dynamic Symbol Caching achieve similar bit rates. However, in this case, the Global method exceeds the 1 Mbyte dictionary memory limit specified by the JBIG2 standard [2, 7].

Next, we encoded 19 pages of *The Data Compression Book* [29] using different numbers of stripes per page. For this experiment, the Dynamic Symbol Caching method was constrained to a 1 Mbyte of dictionary memory. Fig. 1.8 illustrates the compressed bit rate as a function of the number of stripes per page for each of the



(a)



(b)

Fig. 1.7. Bit rate comparison as a function of varying dictionary memory size for five algorithms. The multi-page test document used is (a) *Purdue Wavelinks Magazine* (23 pages) (b) *Photoshop CS Manual* (46 pages). Note that Dynamic Symbol Caching always achieves lower bit rates than Local Dynamic and Independent Static. Also, when the dictionary memory is large, Dynamic Symbol Caching and Global achieve a similar bit rate to that of Optimal Dynamic. However, Global and Optimal Dynamic are not practical. Furthermore, Global exceeds the 1 Mbyte memory limitation specified by the JBIG2 standard for the *Photoshop CS Manual* test document.

algorithms listed in Table 1.1. We can see from Fig. 1.8 that the compressed bit rate increases as the page is divided into more stripes. So, while page striping reduces encoder and decoder memory usage, it also introduces a bit rate penalty. However, we can also see that Dynamic Symbol Caching minimizes this bit rate penalty as compared to Local Dynamic and Independent Static. Furthermore, Dynamic Symbol Caching achieves between 13% and 32% lower bit rate than Local Dynamic, and between 30% and 46% lower bit rate than Independent Static. On the other hand, Dynamic Symbol Caching requires between 0.5% and 2% higher bit rate than Global. However, the Global method is not practical because it exceeds the 1 Mbyte dictionary memory limit proposed by the JBIG2 standard, and because it requires that the entire multi-page document be buffered to construct the symbol dictionary.

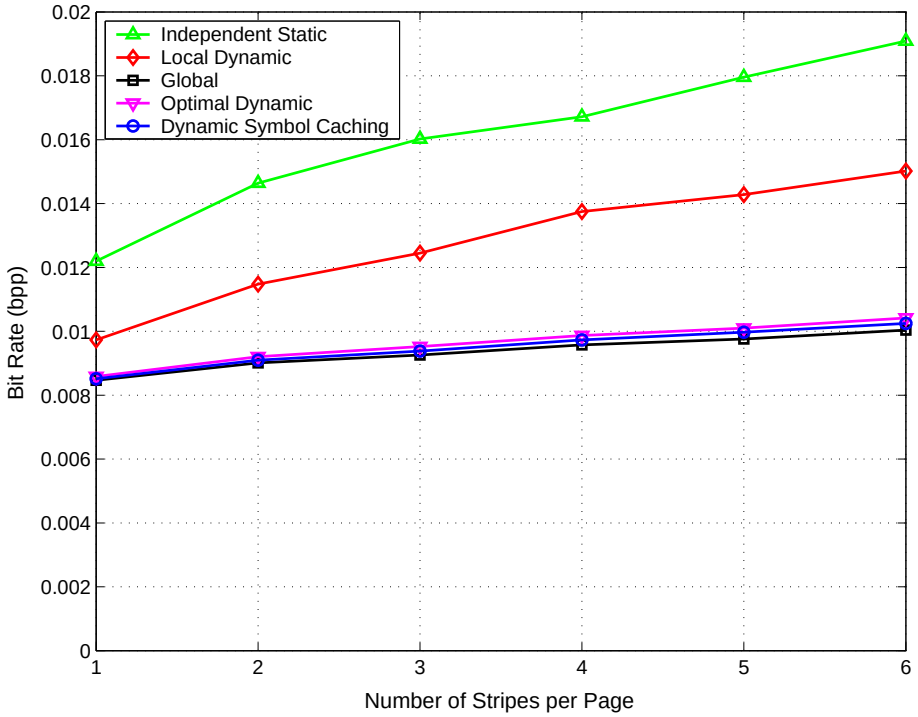


Fig. 1.8. Compressed bit rate as a function of the number of stripes per page using five dictionary updating methods. For this experiment, we used the *The Data Compression Book* multi-page test document. Note that as the page is divided into more stripes, Dynamic Symbol Caching is more effective than Local Dynamic and Independent Static.

For our next experiment, we encoded the full test suite shown in Table 1.2, a total of 162 pages of text and graphics, using the five algorithms listed in Table 1.1. Table 1.5 illustrates the reduction in average bit rate achieved by Dynamic Symbol Caching relative to that of Local Dynamic and Independent Static as a function of the number of stripes per page. From Table 1.5, we can see that as the page is divided into more stripes, the advantage of Dynamic Symbol Caching increases relative to Local Dynamic and Independent Static. Therefore, in systems where the encoder requires small buffers, the Dynamic Symbol Caching method can use very small stripes without as great a sacrifice in bit rate.

Table 1.5

Percentage of reduction in average bit rate achieved by Dynamic Symbol Caching relative to that of Local Dynamic and Independent Static for different numbers of stripes per page. For this experiment, we used the full test suite shown in Table 1.2. Note that as the page is divided into more stripes, the advantage of Dynamic Symbol Caching increases relative to both Local Dynamic and Independent Static.

Stripes per page	1	2	3	4	5	6	7	8	16	32	64
Bit rate reduction of Dynamic Symbol Caching relative to Local Dynamic	12%	18%	21%	24%	25%	27%	28%	29%	33%	38%	38%
Bit rate reduction of Dynamic Symbol Caching relative to Independent Static	25%	31%	34%	37%	39%	40%	41%	42%	46%	48%	44%

Fig. 1.9 compares the average encoding time per page for the five dictionary updating methods. The number of stripes per page used varies from one to six. These simulations were run on a 3.20 GHz Linux server. Note that Dynamic Symbol Caching is faster than both Global and Optimal Dynamic, but slower than both Independent Static and Local Dynamic. This is because a Dynamic Symbol Caching dictionary will typically be larger than both an Independent Static and a Local Dynamic dictionary.

Therefore, Dynamic Symbol Caching will typically require more time for the symbol matching process. In addition, Dynamic Symbol Caching requires that the cache statistics (dictionary memory size and frequency of use of dictionary symbols) be updated for each new stripe. We can also see from Fig. 1.9 that, as the page is divided into more stripes, the encoding time for Dynamic Symbol Caching increases, while the encoding time for both Local Dynamic and Independent Static decreases. This is because, as the number of stripes per page increases, the Local Dynamic and Independent Static dictionaries become smaller since they only contain symbols in the current stripe.

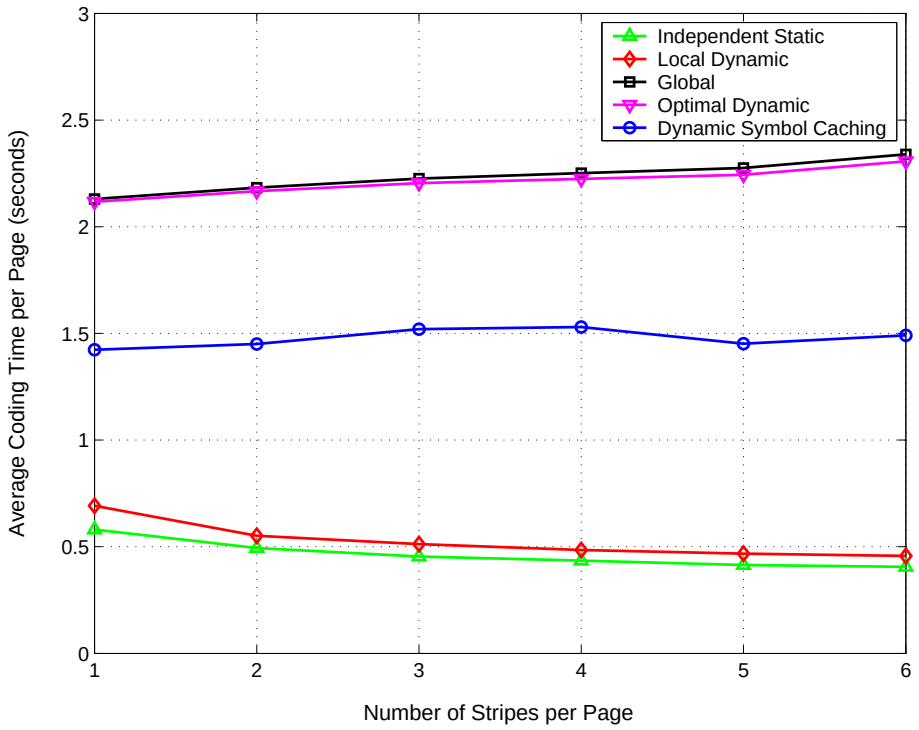


Fig. 1.9. Average encoding time required to encode a page using five dictionary updating methods. For this experiment, we used the full test suite shown in Table 1.2. Note that Dynamic Symbol Caching is slower than Local Dynamic and Independent Static because its dictionary will typically be larger. Local Dynamic and Independent Static dictionaries only contain symbols in the current stripe. Hence, they will typically become smaller as the number of stripes per page increases.

All the results shown until this point assume that all stripes in a single document have a fixed height. Next, we show that, when using page striping, the bit rate penalty can be reduced by adaptively changing the stripe size to minimize the splitting of symbols during the striping process. For this experiment, we encode the full test suite shown in Table 1.2 using fixed and adaptive striping. We test three dictionary updating methods: the Independent Static, Local Dynamic, and Dynamic Symbol Caching. Fig. 1.10 compares the average bit rates as a function of the number of stripes per page for each dictionary updating method using fixed and adaptive striping.

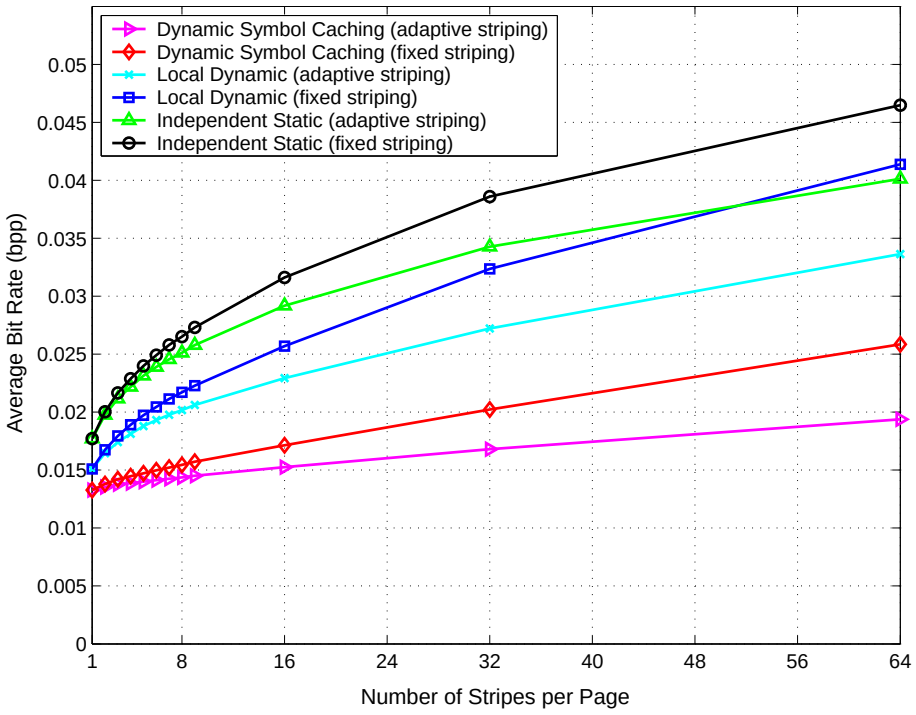


Fig. 1.10. Comparison of the average bit rate achieved by three dictionary updating methods as a function of the number of stripes per page using fixed and adaptive striping. For this experiment, we used the full test suite shown in Table 1.2. Note that Dynamic Symbol Caching with adaptive striping achieves a bit rate reduction of between 12% and 53% relative to that of the best previous dictionary design (Local Dynamic with fixed striping).

Table 1.6

Percentage of reduction in average bit rate achieved by Dynamic Symbol Caching (DSC) with adaptive striping relative to that of three dictionary methods that use fixed striping. For this experiment, we used the full test suite. Note that as the page is divided into more stripes, the percentage of reduction in average bit rate achieved by Dynamic Symbol Caching with adaptive striping increases relative to that of dictionary methods that use fixed striping.

Stripes per page	1	2	3	4	6	8	16	32	64
Bit rate reduction of DSC (adaptive) relative to DSC (fixed)	0%	2%	3%	4%	6%	7%	11%	17%	25%
Bit rate reduction of DSC (adaptive) relative to Local Dynamic (fixed)	12%	19%	23%	27%	31%	34%	41%	48%	53%
Bit rate reduction of DSC (adaptive) relative to Independent Static (fixed)	25%	32%	36%	39%	43%	46%	52%	56%	58%

Table 1.6 illustrates the reduction in average bit rate achieved by Dynamic Symbol Caching with adaptive striping relative to Dynamic Symbol Caching with fixed striping and also relative to previous dictionary methods as the number of stripes varies from 1 to 64. We can see from Table 1.6 that Dynamic Symbol Caching with adaptive striping achieves a bit rate reduction of between 12% and 53% relative to that of the best previous dictionary design (Local Dynamic with fixed striping). For example, using 32 stripes per page, Dynamic Symbol Caching with adaptive striping achieves a 48% bit rate reduction relative to that of Local Dynamic. The use of multiple stripes per page results in a bit rate increase for each dictionary method; however, we can see from Fig. 1.10 that adaptive striping minimizes this bit rate penalty. Compared to using one stripe per page, Dynamic Symbol Caching with adaptive striping reduces the bit rate penalty by half. Local Dynamic with adaptive striping reduces the bit rate penalty by between 18% and 30% as the number of stripes increases from 2 to 64. Similarly, Independent Static with adaptive striping reduces the bit rate penalty

by between 12% and 22%. We can also see from Fig. 1.10 that Dynamic Symbol Caching with adaptive striping achieves the lowest bit rate among the three methods. Hence, by using the Dynamic Symbol Caching method with adaptive striping, it is possible to use more stripes (allowing smaller encoding and decoding buffers) with a reduced bit rate penalty.

For our last experiment, we encoded the full test suite shown in Table 1.2, and we compared our JBIG2 encoder which we call PJBIG2 to existing commercial JBIG2 encoders. Table 1.7 lists each of these JBIG2 encoders, and summarizes the settings we used to perform the tests. Our PJBIG2 encoder uses 1 stripe per page, Dynamic Symbol Caching with a dictionary memory size constrained to 1 Mbyte, and Huffman/MMR coding. Note that within the existing commercial JBIG2 encoders, only ImageGear, PdfCompressor, and Pegasus use Huffman/MMR coding, while the remaining encoders use arithmetic coding.

Table 1.7
Parameters settings for the existing commercial JBIG2 encoders.

Application	Version	Parameters Settings
Pegasus Imaging JB2PTEST	1.0.0.7	Encmode="Lossy Text MQ" or "Lossy Text MMR" with looseness param=40 (where 0: exact match - 100: loosest match)
CVista PdfCompressor	3.1	Do Not Include OCR Do Not Optimize files for Web transmission Bitonal compression filter="JBIG2" Compression Mode="Perceptually Lossless" Compression Algorithm Selection="Better Compression" Display/Print Speed: "No MMR encoding" or "Full MMR"
LuraTech LuraDocument PDF Compressor Desktop	2.2.01.08	Profile: B/W (text only)
Snowbound Snowbatch	3.0	Default settings
AccuSoft ImageGear JBIG2 Plug-in	14.8	Default settings

All the JBIG2 encoders in our comparison used lossy compression, but showed no visible substitution errors in our test suite of images. The parameters for each JBIG2 encoder were selected in the following manner. For PJBIG2, Pegasus, and PdfCompressor we selected the quality level that produced the lowest bit rate with no visible substitution errors. LuraDocument PDF, Snowbatch, and ImageGear did not allow us to select a quality level. For these encoders, we used their default settings, and then we verified that no visible substitution errors occurred.

Note that within the existing commercial JBIG2 encoders, both LuraDocument PDF and PdfCompressor embed the JBIG2 bitstream inside a portable document format (PDF) file. The compression efficiencies shown in Table 1.8 for LuraDocument PDF and PdfCompressor include the PDF wrapper overhead. However, this overhead is minimal [30, 31].

Table 1.8

Compression efficiency of several lossy JBIG2 encoders compared to that of PJBIG2. For each encoder, we set its parameters so that no visible substitution errors occurred in the full test suite. Note that even though PJBIG2 uses Huffman/MMR coding, it still achieves the lowest bit rate among all the encoders. This is because PJBIG2 makes up for the deficiencies of the Huffman/MMR coders relative to the arithmetic coder by using a Dynamic Symbol Caching dictionary.

Encoder	Entropy/Bitmap Coding Method	Average Bit Rate (bpp)	Compression Ratio Average
PJBIG2	Huffman/MMR	0.01327	75:1
ImageGear	Huffman/MMR	0.01597	63:1
Pegasus	Huffman/MMR	0.01842	54:1
PdfCompressor	Huffman/MMR	0.01858	54:1
Pegasus	Arithmetic (MQ)	0.01425	70:1
PdfCompressor	Arithmetic (MQ)	0.01465	68:1
LuraDocument PDF	Arithmetic (MQ)	0.02344	43:1
Snowbatch JBIG2	Arithmetic (MQ)	0.03397	29:1

Table 1.8 compares the compression efficiency of PJBIG2 to that of existing JBIG2 commercial encoders. We can see that, while maintaining a visually lossless reconstructed image quality, PJBIG2 achieves the lowest average bit rate among all the encoders. Compared to the JBIG2 encoders that use Huffman/MMR coding, PJBIG2 achieves a 17% average bit rate reduction relative to that of ImageGear, and a 28% and a 29% average bit rate reduction relative to that of Pegasus (Huffman/MMR) and PdfCompressor (Huffman/MMR), respectively. Compared to the JBIG2 encoders that use arithmetic coding, PJBIG2 achieves 7%, 9%, 43%, and 62% lower average bit rate than Pegasus (MQ), PdfCompressor (MQ), LuraDocument PDF, and Snowbatch, respectively. Note that although Snowbatch uses arithmetic coding, Snowbatch requires substantially higher bit rate than the rest of the encoders because it does a nearly-lossless image coding.

Overall, at a visually lossless reconstructed image quality, our PJBIG2 encoder achieves lower bit rates than all other encoders. We believe that a major advantage of our encoder is that it uses a novel dynamic dictionary structure which can efficiently share dictionary symbols across multiple pages (and stripes) of a document. This dynamic symbol dictionary makes up for the deficiencies of the Huffman/MMR coders relative to the arithmetic coder.

1.7 Conclusion

In this chapter, we investigated several methods for updating a JBIG2 symbol dictionary, and we presented a novel adaptive dictionary structure that shares a dynamic dictionary across stripes of a multi-page document. The size of the dictionary in JBIG2 is limited and, when the dictionary is full, we must decide which symbols to keep and which to discard in order to make room for new symbols. The Dynamic Symbol Caching method uses the dictionary memory size and the frequency of use of each symbol to determine how the dictionary will be updated at each stripe. By doing this, Dynamic Symbol Caching can retain the dictionary symbols that are most likely

to be used in the future while not exceeding the 1 Mbyte memory limitation specified by the JBIG2 standard. Results show that our Dynamic Symbol Caching method reduces the bit rate by between 12% and 53% relative to that of the best previous dictionary design. The relative reduction increases as more pages or more stripes are encoded in a single JBIG2 bitstream. We also showed that by adapting the stripe size to the content of the page, the bit rate was reduced by 2% or more as the page is divided into more stripes. In addition, the Dynamic Symbol Caching method with adaptive striping offers the potential to use more stripes with less bit rate penalty to reduce the encoder and decoder raster buffer sizes. Furthermore, we presented a new pattern matching criterion, the PWXOR, which results in both low bit rates and high encoding speeds while providing visually lossless reconstructed image quality.

2. HARDWARE-FRIENDLY MIXED CONTENT COMPRESSION ALGORITHM

2.1 Introduction

With the wide use of document imaging applications such as scan-to-print and scan-to-email, it has become more important to efficiently compress, store, and share large document files. Without compression, a letter-sized color document scanned at a typical sampling resolution of 300 dots per inch (dpi) becomes approximately 24 Megabytes; and scanned at 600 dpi, the document in raster format occupies nearly 100 Megabytes. The Joint Photographic Experts Group (JPEG) [32] is a compression algorithm commonly used in the scanning pipeline. While being suitable for photographic images, JPEG is not suitable for compound documents comprising a combination of text, graphics, and pictures. JPEG-compressed images with sharp edges typically contain ringing artifacts, and choosing a suitably high quality level to suppress artifacts results in large compressed files.

Most existing compound document compression algorithms may be classified either as block-based approaches or as layer-based approaches. The guaranteed fit (GRAFIT) [33] is a low-complexity block-based document compression algorithm suitable for compression of continuous tone compound documents, but it is designed for synthetic data, and it is not as effective for scanned documents which contain halftone and sensor noise. Layer-based compression algorithms such as the mixed raster content (MRC) [1] format and the compression algorithm proposed by Cheng and Bouman in [34] achieve very low bit rates while maintaining text and graphics quality by using a binary mask layer to encode the high-resolution transitions that are essential to text and graphics quality. In particular, the most basic MRC approach separates the compound document into a binary mask layer, a background

layer, and a foreground layer, and then compresses each layer as a separate image using a different coding method.

In this chapter, we propose a new block-based compression algorithm for raster scanned versions of compound color documents. In particular, we have developed a content-adaptive lossy document compression algorithm, which we call mixed content compression (MCC). MCC is distinct from previous MRC compression algorithms in that it is designed to work with conventional JPEG coding using only an 8 row buffer of pixels. In contrast, traditional MRC-based compression algorithms buffer image strips that typically require large memory buffers, and are therefore not easily implemented in imaging pipeline hardware. In particular, we based our algorithm design on 300 dpi scans from Hewlett-Packard's low cost all-in-ones. MCC first separates the text and graphics from an image by using a simple minimal mean square error (MSE) block classification algorithm, which allows a hardware efficient implementation. Then, MCC uses the JPEG encoder to effectively compress the background and picture content of the document image. The remaining text and line graphics in the image, which require high spatial resolution, but can tolerate low color resolution, are compressed using a JBIG1 [5] encoder and color quantization. JBIG1 is a lossless binary image compression standard that uses the IBM arithmetic Q-coder. It predicts and codes each bit based on the previous bits and the previous lines of the image. Because JBIG1 does not reference future bits, it allows compression and decompression of images in scanning order. In our implementation, the JBIG1 encoder uses sequential compression and 8 lines per stripe.

MCC first divides the image into 8×16 nonoverlapping blocks of pixels and classifies them into two different classes. The classes are background/picture and text/graphics. Then, MCC compresses each class differently according to its characteristics. In particular, MCC compresses the background and picture content of a document image by first producing a background image that is similar to the original image but with the text and graphics removed. Because the background image typically does not contain any high resolution transitions, it can be JPEG coded using a

much larger quantization step size that can be used for text. The remaining text and graphics in the original image is then compressed using a JBIG1 compressed binary mask image and a separate set of quantized foreground colors. The binary mask image and the foreground colors result from segmenting a text/graphics block into a binary mask, a background color, and one or multiple foreground colors. To do this segmentation, we use a minimal MSE thresholding algorithm.

In order to experimentally evaluate our proposed MCC algorithm, we use an image database consisting of 600 scanned document images, comprising a combination of text images, photographic/picture images, and mixed images. Text images typically contain typed or handwritten text, line art, and simple graphics. Photographic/picture images typically contain continuous tone natural scenes and high dynamic range halftoned images. Mixed images contain both text and photographic/picture content. We compare the compression efficiency of our MCC encoder to that of JPEG at similar reconstructed image quality.

Results show that for our full test suite the MCC algorithm achieves an average bit rate reduction of 38% relative to that of JPEG at similar or better quality. The relative bit rate reduction is typically higher for text and mixed images. In particular, for lossy compression of mixed documents, MCC achieves a 43% bit rate reduction relative to that of JPEG. For text documents, MCC achieves a relative bit reduction of 44%, and for photographic and picture images, MCC reduces the average bit rate by 23% relative to that of JPEG.

The remainder of this chapter is organized as follows. Section 2.2 describes in detail our proposed MCC algorithm. Section 2.3 shows our experimental results, and the concluding remarks are presented in Section 2.4.

2.2 Mixed Content Compression Algorithm

Our proposed MCC encoder is a simple low bit rate block-based algorithm specifically designed for compression of 300 dpi scans of color compound documents. We

will show that MCC efficiently compresses documents that contain mixed data by adapting to the content automatically. In contrast, a baseline JPEG encoder does not adapt to the content of the document, and therefore requires quantization tables which have low quantization steps to guarantee the text and graphics quality. Our MCC document compression algorithm is distinct from a standalone JPEG encoder in that it first pulls out the text and graphics from the scanned image, and then encodes them as side information using a JBIG1 encoder and color quantization. The remaining background and picture content in the scanned image is then JPEG compressed using quantization tables which have much higher quantization steps than those of the standalone JPEG coder. In particular, to guarantee acceptable text and graphics quality, the standalone JPEG encoder typically requires quantization tables scaled at quality level 50 (JPEG Q50), while the JPEG encoder used by MCC can achieve similar text and graphics quality by using quantization tables scaled at quality level 25 (JPEG Q25). Fig. 2.1 compares the general structure of the JPEG encoder and the MCC encoder.

Our MCC algorithm works with $N \times M$ nonoverlapping blocks of pixels, one at a time, classifying each block into one of four possible classes: background/picture, 2-color, 3-color, and 4-color. In this chapter, we use $N = 8$ and $M = 16$. Background/picture blocks typically contain continuous tone images or smoothly varying background regions. Alternatively, text and graphic content are ideally classified as 2, 3, or 4-color blocks since local regions of text and graphics can typically be represented by a small number of distinct colors. Then, MCC encodes each class differently using a compression algorithm specifically designed for that class.

Throughout this chapter, we use $\boldsymbol{x}$ to denote the original image, $\boldsymbol{z}$ to denote the background image, and $\boldsymbol{b}$ to denote the binary image produced by the MCC algorithm. Then, we use $\boldsymbol{x}_k$ to denote the kth block in the image, where the blocks are taken in raster order, where $0 \leq k < L$, with L being the number of blocks. Also, $x_k(i, j)$ denotes a pixel at position (i, j) in an $N \times M$ block. Note that i and j indicate the row and column, respectively.

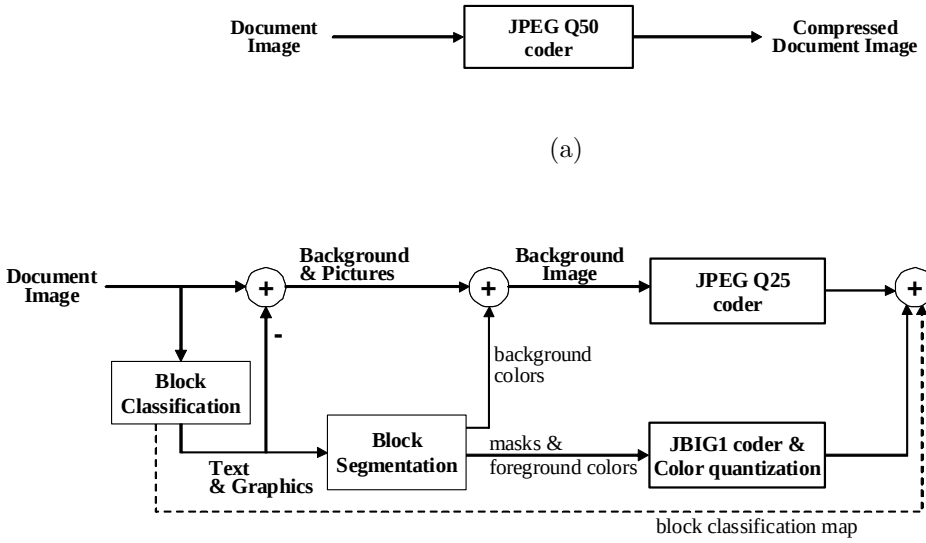


Fig. 2.1. General structure of the encoders. (a) JPEG encoder. (b) Mixed Content Compression encoder. The MCC encoder first removes the text and graphics content from the original image. The background image that results from this process is then JPEG compressed using a much lower quality level than the JPEG encoder in (a). Then, MCC encodes the text and graphics content as side information using a JBIG1 encoder and color quantization.

Fig. 2.2 illustrates the flow diagram of the MCC algorithm, and Fig. 2.3 shows an example of the MCC compression of a document image. For each new block, $\boldsymbol{x}_k$, in the original image, $\boldsymbol{x}$, MCC first classifies the block either as an n-color block, $2 \leq n \leq 4$, or as a background/picture block. A detailed description of the MCC block classification algorithm is given in Section 2.2.1. The MCC block classification algorithm produces a background image, $\boldsymbol{z}$, which is the same size as the original image, $\boldsymbol{x}$, and that is compressed by JPEG. Blocks in the original image that are classified as background/picture are simply copied from the original image to the background image. However, n-color blocks in the background image are filled-in with blocks containing a constant color corresponding to the background color of the

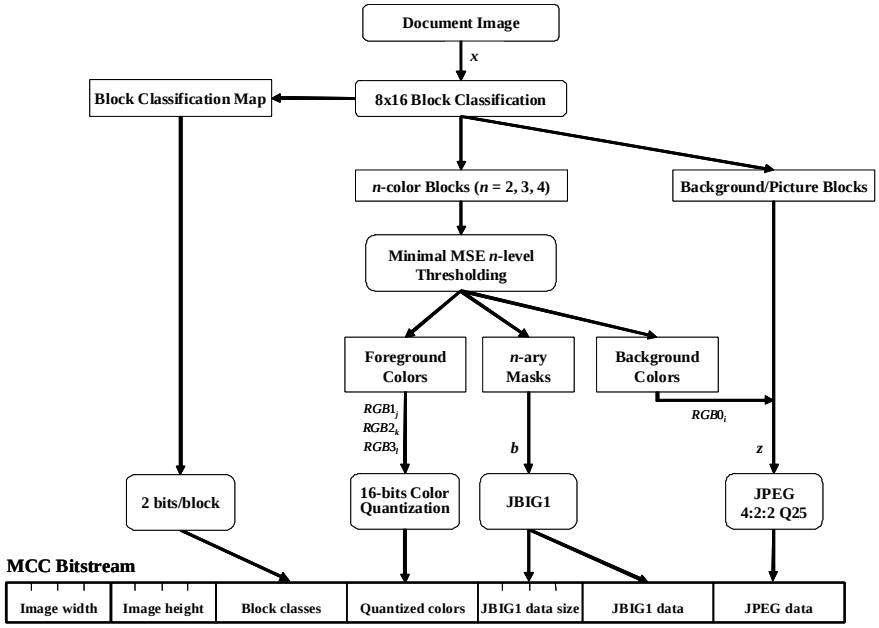


Fig. 2.2. Flow diagram of the mixed content compression (MCC) algorithm

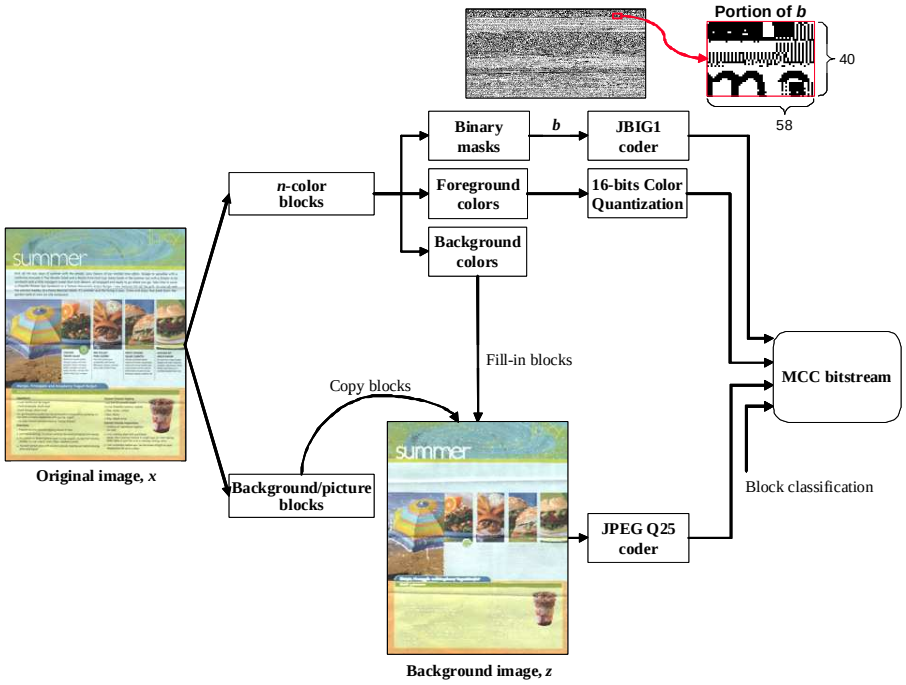


Fig. 2.3. Example of the MCC encoding process

n-color block. From Fig. 2.3, we can see that this process essentially produces a background image with the text and graphics removed.

A block classified as an n-color block, is then segmented into an n-ary mask, a background color, and $n-1$ foreground colors using the segmentation algorithm described in Section 2.2.2 and Section 2.2.3. The n-ary masks from all the n-color blocks are grouped together to form a binary image, $\boldsymbol{b}$, that has a width constraint to a maximum of W pixels. In this chapter, we set W to the width of the original image. Then, the binary image, $\boldsymbol{b}$, is compressed by JBIG1 [5]. The foreground colors from all the n-color blocks are quantized to 16 bits by transmitting the six most significant bits of the red and green components, and the four most significant bits of the blue component, and then rounding. Finally, the block classes are also packed in raster order, and then transmitted using two bits per block. All the distinct data bitstreams are concatenated to form the MCC bitstream.

The MCC bitstream is organized as shown in Fig. 2.2. The image width is a 4-byte field that gives the width in pixels of the encoded image. Similarly, the image height is a 4-byte field that gives the height in pixels of the encoded image. The block classes field contains the class of each block in the image. The size in bytes of this field is determined from the image width and height fields. Each block class is specified with two bits per block since we have four distinct classes. The quantized colors field contains the foreground colors, quantized to 16 bits, from all the n-color blocks; hence every 2 bytes represents one color. The size of this field in bytes is determined from the block classes field. JBIG1 data length is a 4-byte field that contains the length of the JBIG1 bitstream in bytes. The JBIG1 data field contains the JBIG1 bitstream, which encodes the n-ary masks from all the n-color blocks. The JBIG1 data length and data fields appear only if the MCC compressed file is encoding n-color blocks. Finally, the JPEG data field contains the JPEG bitstream, which encodes all the background/picture blocks and the background colors from all the n-color blocks.

Each field length is an integer number of bytes. Thus, the block classes bitstream may need to be padded with zeros to make its data length an integer number of bytes.

Also note that the image width and height fields are required because sometimes they cannot be determined from the JPEG bitstream. The JPEG compressed background image will always have dimensions that are a multiple of 8. So, in situations where the width or height of the original image is not a multiple of 8, the dimensions of the JPEG compressed background image may differ slightly from the true dimensions of the original image compressed by MCC.

2.2.1 MCC Block Classification Algorithm

Blocks in the original image, each block being N pixels wide by M pixels high, are classified in raster order. We use a minimum mean square error (MMSE) approach to classify each block into one of four possible classes: background/picture, 2-color, 3-color, and 4-color. The class of each block is chosen so that the mean square error (MSE) between the original block and the compressed block is below a predetermined threshold. Let $\boldsymbol{x}_k = (\boldsymbol{x}_{k,\mathrm{R}}, \boldsymbol{x}_{k,\mathrm{G}}, \boldsymbol{x}_{k,\mathrm{B}})$ be the RGB representation of the k^{th} block in the original image. Let $\boldsymbol{x}_{k,\alpha}$ denote the α component of block $\boldsymbol{x}_k$, where α can be the red (R), the green (G), or the blue (B) component. Also, let $\tilde{\boldsymbol{x}}_{n,k}$ denote the original block represented by n colors. Then, the mean square error between the α component of the original block, $\boldsymbol{x}_{k,\alpha}$, and that of the block represented by n colors, $\tilde{\boldsymbol{x}}_{n,k,\alpha}$, is given by

$$\mathrm{MSE}_n(\alpha) = \frac{1}{NM} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \left(x_{k,\alpha}(i,j) - \tilde{x}_{n,k,\alpha}(i,j)\right)^2 \tag{2.1}$$

Let MSE_n be the maximum MSE among the three color components associated with the block when it is represented by n colors

$$\mathrm{MSE}_n = \max\left\{\mathrm{MSE}_n(\mathrm{R}), \mathrm{MSE}_n(\mathrm{G}), \mathrm{MSE}_n(\mathrm{B})\right\} \tag{2.2}$$

The block classification algorithm illustrated in Fig. 2.4 works in the following manner. For each block, $\boldsymbol{x}_k$, we first determine if the block is a smooth background block. To do this, we compute MSE_1, which is the maximum MSE associated with

the block when pixels in the block are represented by the mean color of the block. If $\text{MSE}_1 < T_1$, where T_1 is a predetermined threshold, we classify the block as a background/picture block. But, if $\text{MSE}_1 \geq T_1$, then we check if the block can be represented by 2, 3, or 4 colors.

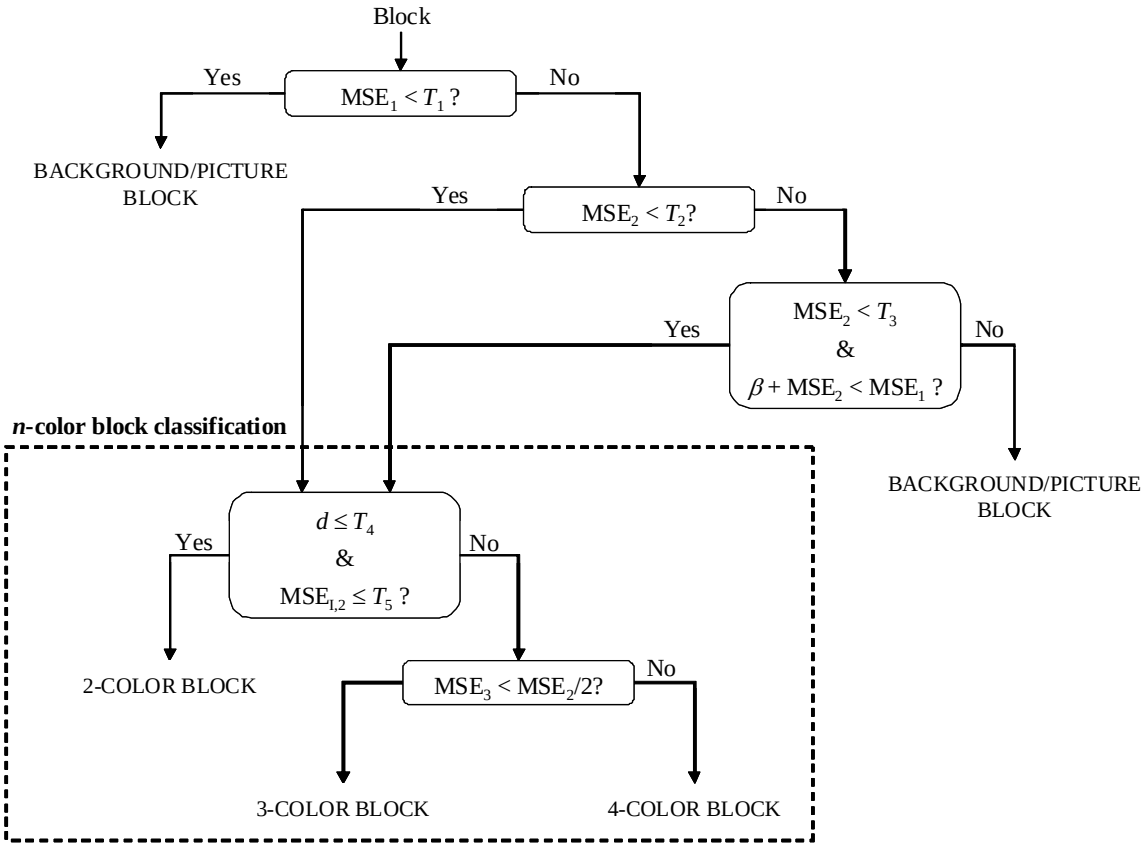


Fig. 2.4. Flow chart of the MCC block classification algorithm

In order to check if a block can be accurately represented by 2, 3, or 4 colors, we first convert the original block, $\boldsymbol{x}_k$, to a 2-color block by using the segmentation algorithm described in Section 2.2.2. Then, we compute the maximum MSE associated with the block when it is represented by 2 colors, MSE_2. If $\text{MSE}_2 < T_2$, then the block is classified as an n-color block. On the other hand, if $\text{MSE}_2 \geq T_3$, where threshold $T_3 > T_2$, then the block is classified as a background/picture block. But,

if MSE_2 lies in the interval $[T_2, T_3)$, then we compare the maximum MSE associated with the block represented by one color to that of the block represented by two colors.

A block that can be represented by one color is classified as a background/picture block and JPEG compressed at quality level 25; hence, it will typically require a lower bit rate than that of a block compressed as a 2-color block. Let β be an MSE penalty we add to MSE_2. Then, if $(\beta + \text{MSE}_2) \geq \text{MSE}_1$, the block is classified as a background/picture block. Otherwise, the block is classified as an n-color block.

If a block is classified as an n-color block, we use the n-color block classification algorithm described in Section 2.2.1 to determine if the block requires 2, 3, or 4 colors to be accurately represented. Note that all the thresholds we use in this block classification algorithm are determined from the test images. Threshold T_1 is used to quickly identify blocks in the original image that consist mostly of a smooth background color. Hence, T_1 must be sufficiently small that blocks that contain two distinct colors are not classified as background/picture blocks. Similarly, the threshold T_2 is used to quickly identify blocks that can be represented by n colors. The interval $[T_2, T_3]$ must be chosen so that local regions of text and graphics are classified as n-color blocks. Finally, the thresholds T_4 and T_5 in Fig. 2.4 are used to determine if a block requires more than two colors to be represented accurately. The thresholds values for our MCC encoder are $T_1 = T_2 = 200$, $T_3 = 20T_2$, $T_4 = 20$, $T_5 = 200$, and the MSE penalty is $\beta = 500$.

n-Color Block Classification

The background image, $\boldsymbol{z}$, which is compressed by JPEG Q25 using horizontal subsampling, is composed of both background/picture blocks copied from the original image, and constant color blocks filled-in with the background color from an n-color block. In order to avoid any possible aliasing effect in the boundary between a reconstructed background/picture block and a previous neighboring constant color block, we enlarge the width of an n-color block by one pixel so that the current n-

color block, $\boldsymbol{x}_k$, and the next block, $\boldsymbol{x}_{k+1}$, overlap. This enlargement is done as long as the current n-color block is not the last block in the horizontal direction in the original image. Throughout this chapter we use M' to denote the width of an n-color block, $\boldsymbol{x}_k$, where $M \leq M' \leq M + 1$.

A block classified as an n-color block, may require 2, 3, or 4 distinct colors to be accurately represented. First, we represent an n-color block by two colors, and then we check if that representation is accurate. If the 2-color representation is not accurate, then we check if the block requires either 3 or 4 colors. A 2-color block is represented by a foreground color, $RGB0_k$, a background color, $RGB1_k$, and a binary mask, $\boldsymbol{b}_k$, that indicates if the associated pixel is a foreground pixel or a background pixel. Let $b_k(i,j)$ denote a pixel at position (i,j) in the binary mask, $\boldsymbol{b}_k$, associated with pixel $x_k(i,j)$, where $b_k(i,j) \in \{0,1\}$, $0 \leq i < N$, and $0 \leq j < M'$. A binary pixel $b_k(i,j) = m$ is said to be of class m, where $m \in \{0,1\}$. We define as a boundary pixel any color pixel, $x_k(i,j)$, whose associated pixel in the binary mask, $b_k(i,j)$, has at least one neighboring pixel in its 8-point neighborhood which is of a different class. On the other hand, if all the pixels in the 3×3 neighborhood of pixel $b_k(i,j)$ are of the same class as that of $b_k(i,j)$, then pixel $x_k(i,j)$ is defined as an interior pixel.

In order for a block to be classified as a 2-color block, two conditions must be met. First, the average distance, d, of all the boundary pixels to the line determined by the background color, $RGB0_k$, and the foreground color, $RGB1_k$, must be less or equal than a threshold T_4. Second, the maximum mean square error among the three color components between the interior pixels in the the original block, and those in its 2-color representation must be less or equal than a threshold T_5.

If a block requires more than two colors to be represented accurately, we use equation (2.2) to obtain the maximum MSE associated with the block represented by three colors, MSE_3. Then, if $\text{MSE}_3 < \text{MSE}_2/2$, the block is classified as a 3-color block. Otherwise, the block is classified as a 4-color block.

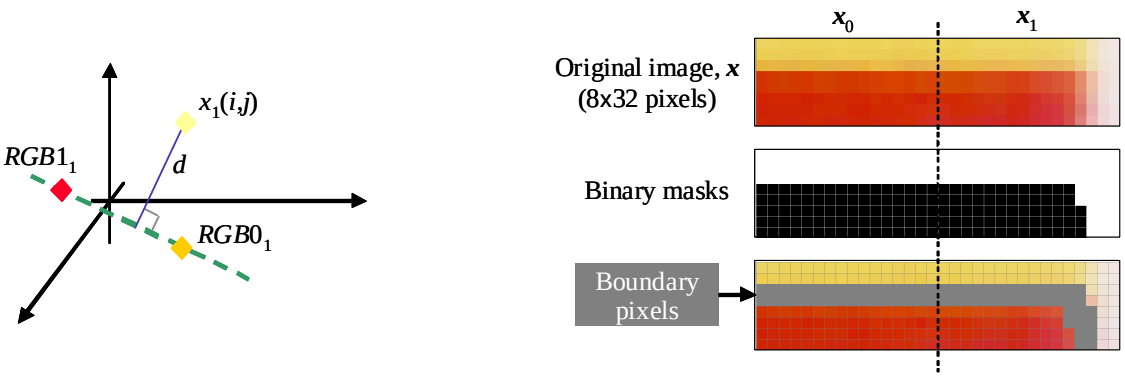


Fig. 2.5. Example of an n-color block classification. In order for a block, $\boldsymbol{x}_1$, to be classified as a 2-color block, two conditions must be met. First, the average distance, d, of all the boundary pixels to the line determined by the background color, $RGB0_1$, and the foreground color, $RGB1_1$, must be less or equal than a threshold T_4. Second, the maximum mean square error among the three color components between the interior pixels in the original block, and those in its 2-color representation must be less or equal than a threshold T_5.

2.2.2 Segmentation and Compression of a 2-Color Block

The 2-color class is designed to compress blocks that can be accurately represented by two colors, such as text. Each 2-color block is represented by one foreground color, one background color, and a binary mask, which is a $N \times M'$ binary block of pixels. To extract the two colors and the binary mask, each block is segmented by the clustering method used by Cheng and Bouman [34], which is a bi-level thresholding algorithm based on a minimal mean squared error thresholding. This blockwise segmentation algorithm is independent between blocks, and works in the following manner. For each block, $\boldsymbol{x}_k$, we first choose the color component, α, which has the largest variance among the three color components (R, G, or B). Then, we find a threshold, t, on the chosen color component which partitions the $N \times M'$ block of pixels into two groups, foreground (represented by the binary value 1) and background (represented by the binary value 0). Let P_1 and P_0 denote the foreground and background group,

respectively. Threshold t is then chosen to minimize the total class variance, γ_α^2, given by

$$\gamma_\alpha^2 = \frac{N_{\alpha,0}\ \sigma_{\alpha,0}^2 + N_{\alpha,1}\ \sigma_{\alpha,1}^2}{N_{\alpha,0} + N_{\alpha,1}} \tag{2.3}$$

where $N_{\alpha,0}$ and $N_{\alpha,1}$ are the number of pixels classified as background and foreground, respectively, and $\sigma_{\alpha,0}^2$ and $\sigma_{\alpha,1}^2$ are the variances within each group of pixels when colors in each group are represented by the mean color of that group. Let $RGB0_k$ and $RGB1_k$ be the the median color of the pixels in each group, where $\|RGB0_k\|_1 \geq \|RGB1_k\|_1$ is true for all k. Then, we call $RGB0_k$ the background color of block k, and $RGB1_k$ the foreground color of block k when it is being represented by 2 colors. That is, in MCC, the background color is always the median color that has the largest l_1 norm among the two colors. This means that the background color will typically be the lighter color among the two median colors.

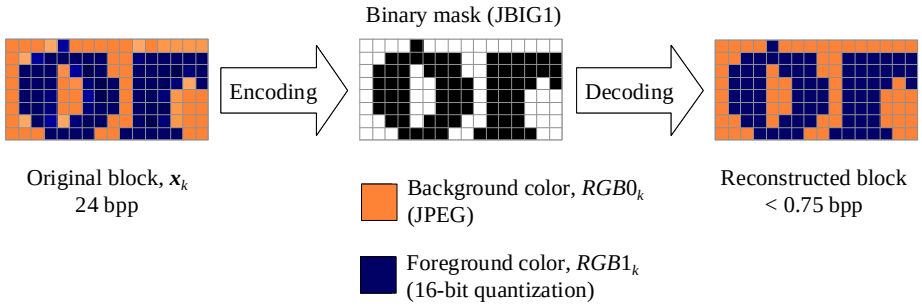


Fig. 2.6. Example of the segmentation and compression of a 2-color block. The 8×16 24-bit color block is represented by a background color, a foreground color, and a binary mask that indicates if the associated pixel is a foreground pixel or a background pixel. The binary masks from all n-color blocks are formed into a single binary image, which is JBIG1 compressed. The foreground colors from all n-color blocks are packed in raster order, and quantized to 16 bits. The background color of a 2-color block is used to fill-in the associated block in the background image. Then, the background image is compressed by JPEG Q25.

Then, MCC compresses a 2-color block by compressing the binary mask and each of the two colors separately. To compress the binary mask, $\boldsymbol{b}_k$, we form all the binary

masks from all n-color blocks into a single binary image, $\boldsymbol{b}$, and then we compress $\boldsymbol{b}$ using JBIG1 [5]; the foreground colors from all n-color blocks are packed together, and quantized to 16 bits; and the background color is copied to fill-in the associated block $\boldsymbol{z}_k$ in the background image, $\boldsymbol{z}$. Then, $\boldsymbol{z}$ is compressed by JPEG Q25. Fig. 2.6 illustrates an example of the MCC compression of a 2-color block.

2.2.3 Segmentation and Compression of 3 and 4-Color Blocks

The 3 and 4-color classes are designed to compress local regions of text and line graphics that require either 3 or 4 colors for an accurate representation. Each n-color block ($n = 3,\ 4$) is represented by one background color, $n-1$ foreground colors, and a grayscale mask that indicates if the associated pixel is a foreground pixel or a background pixel. To extract the n colors and the mask, we use an n-level thresholding algorithm based on a minimal MSE thresholding, which is similar to the clustering method used by Cheng and Bouman in [34], but extended to n levels, $n > 2$. This segmentation algorithm differs from the 2-color segmentation algorithm described in Section 2.2.2 in that the thresholding is done on the luminance component of the block.

Let $\boldsymbol{x}_k = (\boldsymbol{x}_{k,\mathrm{R}}, \boldsymbol{x}_{k,\mathrm{G}}, \boldsymbol{x}_{k,\mathrm{B}})$ be the original $N \times M'$ block of pixels, and let $\boldsymbol{y}_k$ denote the luminance component associated with the original block. The luminance component is computed as a linear transformation from the RGB color space as given in equation (2.4) using the luminance coefficients defined by CCIR Recommendation 601-1, and then clipping them in a range of 0 to 255.

$$\boldsymbol{y}_k = 0.299\ \boldsymbol{x}_{k,\mathrm{R}} + 0.587\ \boldsymbol{x}_{k,\mathrm{G}} + 0.114\ \boldsymbol{x}_{k,\mathrm{B}} \tag{2.4}$$

Then, we find $n-1$ thresholds, $t_1 \ldots t_{n-1}$, on the luminance component which partition the luminance component into n groups, $P_0 \ldots P_{n-1}$. The $n-1$ luminance thresholds are chosen to minimize the total class variance, γ^2_{RGB}, given by

$$\gamma^2_{\mathrm{RGB}} = \gamma^2_{\mathrm{R}} + \gamma^2_{\mathrm{G}} + \gamma^2_{B} \tag{2.5}$$

where γ_α^2 ($\gamma \in \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$) is computed as

$$\gamma_\alpha^2 = \frac{\sum_{i=0}^{n-1} N_{\alpha,i}\ \sigma_{\alpha,i}^2}{\sum_{i=0}^{n-1} N_{\alpha,i}} \tag{2.6}$$

where $N_{\alpha,i}$ is the number of pixels in group P_i, and $\sigma_{\alpha,i}^2$ is the variance associated with group P_i when colors in each group are represented by the mean color of that group.

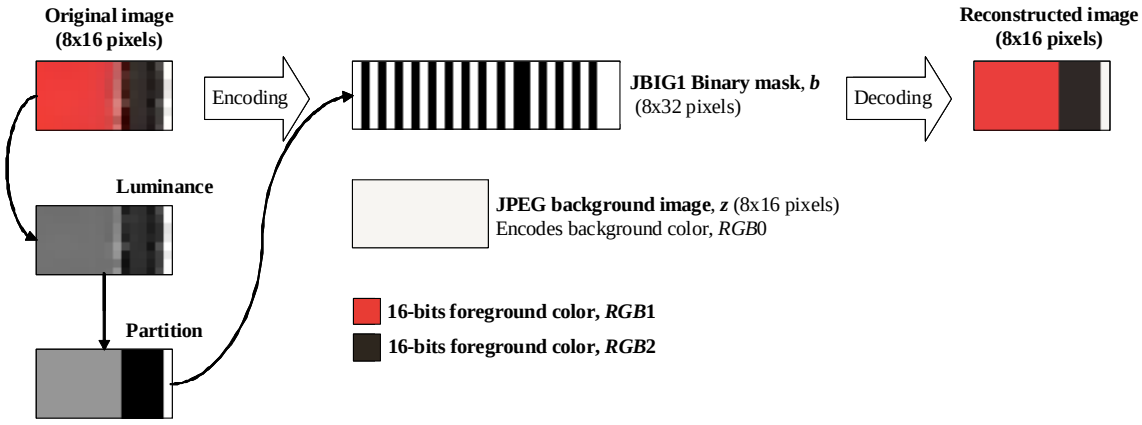


Fig. 2.7. Example of the segmentation and compression of a 3-color block. The 8×16 24-bit color block is represented by one background color, two foreground colors, and a 3-color grayscale mask that indicates if the associated pixel is a foreground pixel or a background pixel. The 3-color grayscale mask is converted into a binary mask. Then, the binary masks from all n-color blocks are formed into a single binary image, which is JBIG1 compressed. The foreground colors from all n-color blocks are packed in raster order, and then quantized to 16 bits. The background color of a 3-color block is used to fill-in the associated block in the background image. Then, the background image is compressed by JPEG Q25.

The background color, $RGB0_k$, is the one that has the largest l_1 norm among the n colors. Each color is given by the average color of the interior pixels in each

group. Note that the boundary pixels between two or more color regions are typically a combination of the colors of each region. In order to obtain an unbiased estimate of the mean color of each group, we do not consider the boundary pixels, except in situations where all the pixels in a group are boundary pixels.

The n-ary masks from 3 and 4-color blocks are $N \times M'$ grayscale blocks of pixels, where each pixel can take one of n possible values ($n = 3,\ 4$). To compress the grayscale masks, we first convert them to binary by using 2 bits per pixel, which produces an $N \times 2M'$ binary block of pixels. Then, the binary masks from all n-color blocks are formed into a single binary image, $\boldsymbol{b}$, and then JBIG1 compressed. The foreground colors are packed in raster order, and then quantized to 16 bits. Finally, the background color is copied to fill-in the associated block $\boldsymbol{z}_k$ in the background image, $\boldsymbol{z}$. Then, $\boldsymbol{z}$ is compressed by JPEG Q25. Fig. 2.7 illustrates an example of the MCC compression of a 3-color block.

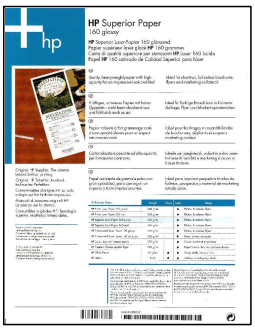
2.3 Experimental Results

For our experiments, we used an image database consisting of 600 scanned document images comprising a mixed of text, pictures, and graphics. These documents were scanned at 300 dpi and 24 bits per pixel (bpp) using four different all-in-one HP models. Then, all the scanned images were descreened to remove halftone noise. The luminance component was descreened using the hardware-friendly descreening (HFD) algorithm proposed by Siddiqui et al. in [35], and the chrominance components were descreened using a 3×3 averaging filter.

The full test suite was divided into three categories: text images, photo/picture images, and mixed images, and each category consists of 200 images (100 color images, and 100 mono images). Text images typically contained text and some graphics. Photo/picture images typically contained continuous tone natural scenes and smoothly varying background regions. Mixed images contained both text and photographic and picture content. Fig. 2.8 illustrates a sample of the test images.

colormix300dpi

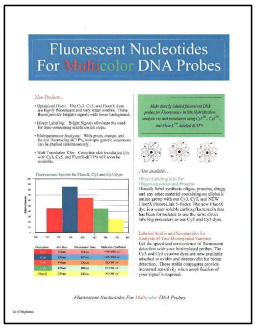
colormix110



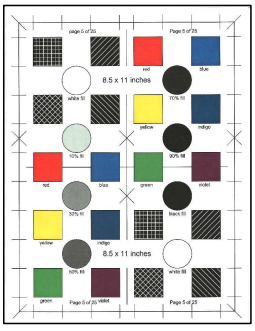
colormix114

colormix117

monomix9

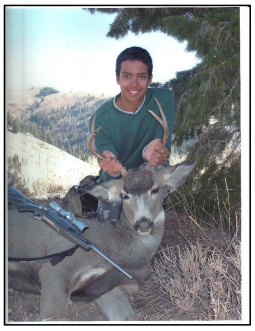


colortext6



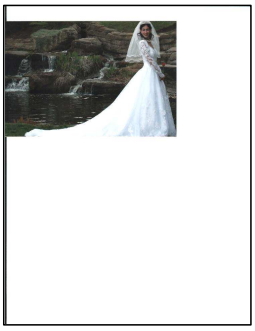
colortext16

monotext12

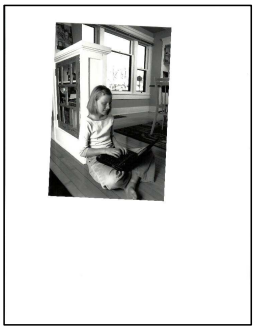


colorpic11

monopic16



colorphoto6



monophoto32

Fig. 2.8. Thumbnails of 12 sample test images. The full test suite consists of 600 images with a variety of content: 100 color mixed images, 100 mono mixed images, 100 color text images, 100 mono text images, 100 color photo/picture images, and 100 mono photo/picture images.

In our experiments, we compare the MCC encoder to the JPEG encoder in terms of bit rate in bits per pixel (bpp). Table 2.1 lists the settings we used for each encoder. These settings were determined from the test images so that the MCC encoder and the standalone JPEG encoder produced similar reconstructed image quality, and were used throughout all our experiments. Note that the standalone JPEG encoder uses the quantization tables provided by the JPEG standard at quality level 50, and that the JPEG encoder included within the MCC framework uses the same tables but at quality level 25. For the Huffman entropy coding, JPEG uses the Huffman tables provided in the JPEG standard.

Table 2.1

Settings for the MCC encoder and the JPEG encoder. These settings were determined from the test images so that the MCC encoder and the JPEG encoder produced similar reconstructed image quality. These settings were used throughout all our experiments.

Encoder	Block classification parameters						JPEG settings		JBIG1 settings	
	T_1	T_2	T_3	T_4	T_5	β	Quality	Subsampling mode	Compression mode	Lines per stripe
MCC	200	200	4000	20	200	500	25	4:2:2	Sequential	8
JPEG	-	-	-	-	-	-	50	4:2:2	-	-

For our first test, we compressed the 12 images illustrated in Fig. 2.8 using both our MCC encoder and the JPEG encoder. Fig. 2.9 shows the associated block classifications, and Fig. 2.10 illustrates samples of the background images that result from the MCC algorithm. Note that the background images consist mainly of smoothly varying background and pictures, and therefore compressed efficiently with JPEG Q25.

Table 2.2 compares the bit rates achieved by MCC and JPEG. The bit rate reduction achieved by MCC relative to that of JPEG varies between 10% for a photo/picture image, and 60% for a typical text image. For these 12 images, MCC achieved a 41% average bit rate reduction relative to that of JPEG.

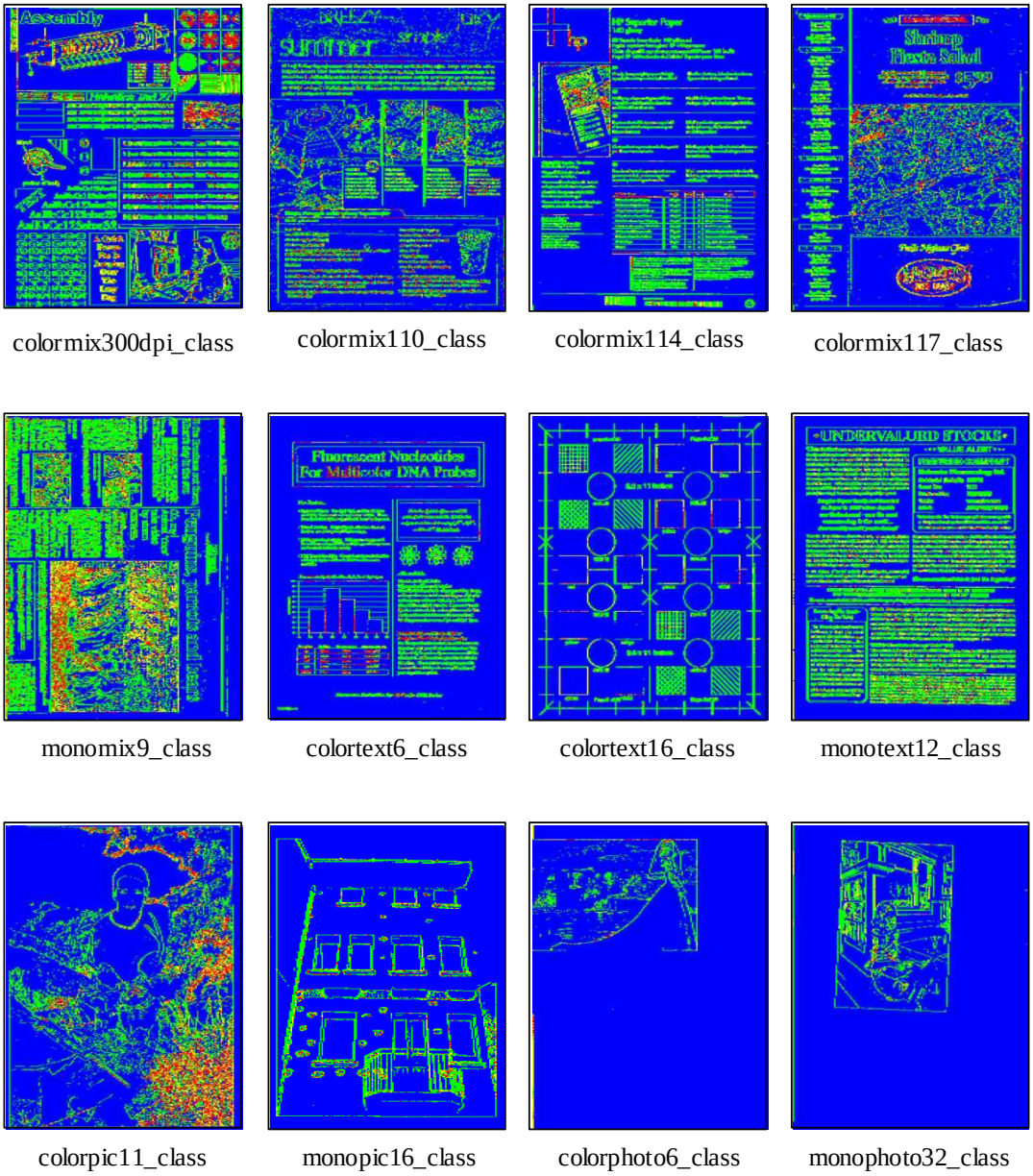


Fig. 2.9. Sample block classifications. Each image is divided into 8×16 blocks of pixels, and blocks are classified in raster order. Blue color blocks indicate background/picture blocks; green color blocks indicate 2-color blocks; yellow color blocks indicate 3-color blocks; and red color blocks indicate 4-color blocks.

colormix300dpi_bkg colormix110_bkg colormix114_bkg colormix117_bkg

monomix9_bkg colortext6_bkg colortext16_bkg monotext12_bkg

colorpic11_bkg monopic16_bkg colorphoto6_bkg monophoto32_bkg

Fig. 2.10. Sample JPEG compressed background images produced by the MCC algorithm. MCC compressed these background images using JPEG Q25 at an average bit rate of 0.214 bpp (112:1 compression average). Note that the background images compress very efficiently with JPEG because they contain mostly smoothly varying regions.

Table 2.2
Comparison between bit rates achieved by our MCC encoder and JPEG at similar quality for 12 sample test images. In average, MCC compresses these test images at 0.402 bpp (60:1 compression), while JPEG compresses them at 0.683 bpp (35:1 compression).

Image	MCC bit rate (bpp)	JPEG bit rate (bpp)	MCC/JPEG bit rate ratio
colormix300dpi	0.448	0.678	0.66
colormix110	0.491	0.893	0.55
colormix114	0.339	0.749	0.45
colormix117	0.491	0.839	0.59
monomix9	0.742	1.218	0.61
colortext6	0.303	0.570	0.53
colortext16	0.263	0.460	0.57
monotext12	0.420	1.061	0.40
colorpic11	0.546	0.706	0.77
monopic16	0.326	0.507	0.64
colorphoto6	0.222	0.247	0.90
monophoto32	0.235	0.265	0.89

Next, we compressed the full test suite composed of 600 images with both our MCC encoder and the JPEG encoder. Table 2.3 and Table 2.4 summarize the average bit rate reduction achieved by MCC relative to that of JPEG for each image category, and Fig. 2.11 compares the average bit rate of each bitstream. Figures 2.12 through 2.18 compare portions of the images compressed using JPEG and MCC. Note that while our MCC algorithm sharpens text and reduces halo around text and vector objects, it achieves a 38% average bit rate reduction relative to that of JPEG. In particular, for lossy compression of mixed documents, MCC achieves an average bit rate which is 43% lower than that of JPEG. For text documents, MCC achieves a 44% bit rate reduction, and for photographic and picture images, MCC achieves a 23% lower average bit rate than that of JPEG. Note that the relative average bit

Table 2.3
Average bit rate reduction achieved by our MCC encoder relative to that of JPEG at similar quality for 600 sample test images. The bit rates achieved by MCC and JPEG are compared for each image type.

Image Type	# of Images	Encoding method	Average bit rate (bpp)	Compression ratio average	Average bit rate reduction	Average compression gain
mono text	100	MCC	0.291	82:1	46%	87%
		JPEG	0.544	44:1	0%	0%
color text	100	MCC	0.307	78:1	42%	73%
		JPEG	0.529	45:1	0%	0%
mono mixed	100	MCC	0.383	63:1	44%	78%
		JPEG	0.683	35:1	0%	0%
color mixed	100	MCC	0.367	65:1	42%	72%
		JPEG	0.632	38:1	0%	0%
mono photo/picture	100	MCC	0.324	74:1	22%	28%
		JPEG	0.416	58:1	0%	0%
color photo/picture	100	MCC	0.370	65:1	24%	32%
		JPEG	0.487	49:1	0%	0%

Table 2.4
Average bit rate reduction achieved by our MCC encoder relative to that of JPEG at similar quality for 600 sample test images. In average, MCC reduces the bit rate by 38% relative to that of JPEG.

Image Type	# of Images	Encoding method	Average bit rate (bpp)	Compression ratio average	Average bit rate reduction	Average compression gain
text	200	MCC	0.299	80:1	44%	80%
		JPEG	0.536	45:1	0%	0%
mixed	200	MCC	0.375	64:1	43%	75%
		JPEG	0.658	36:1	0%	0%
photo/picture	200	MCC	0.347	69:1	23%	30%
		JPEG	0.451	53:1	0%	0%

rate reduction achieved by our MCC encoder relative to that of JPEG is particularly significant for text and mixed images.

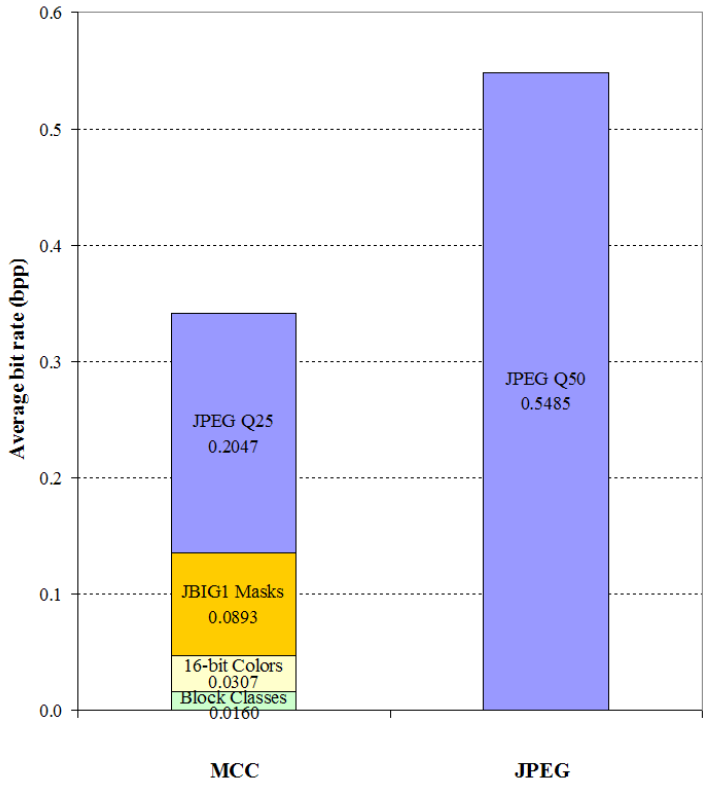
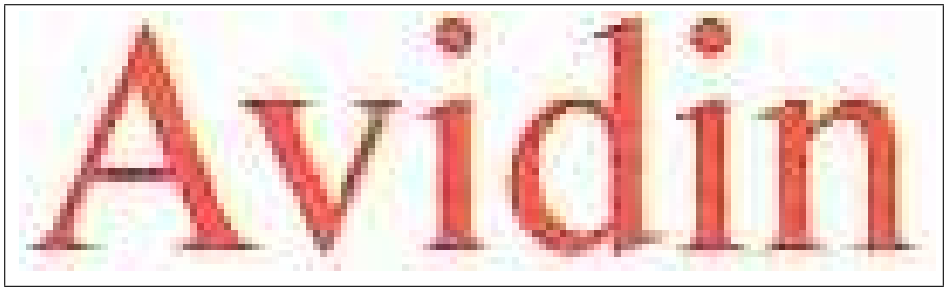
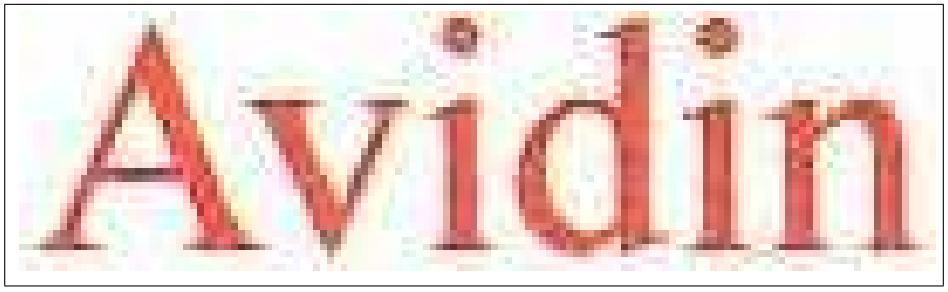


Fig. 2.11. MCC and JPEG bit rate comparison. MCC compresses the full test suite at an average bit rate of 0.340 bpp (71:1); JPEG compresses the full test suite at an average bit rate of 0.546 bpp (44:1). By using our MCC encoder, we achieve a 38% average bit rate reduction relative to that of JPEG at similar quality, that is, a 61% improvement in compression average.

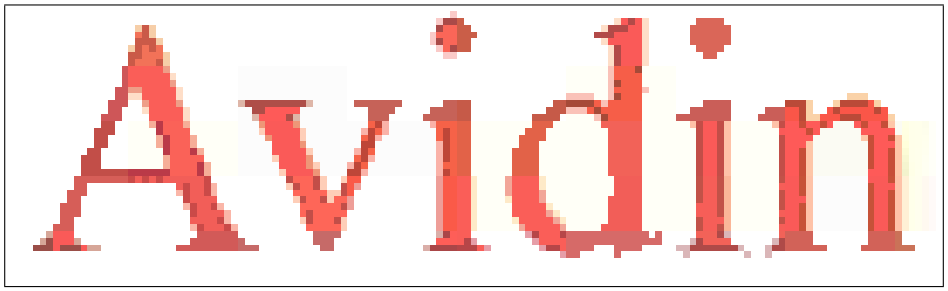
Overall, at similar quality, our MCC encoder achieves lower bit rates than JPEG. We believe that a major advantage of our encoder relative to other MRC-based encoders is that it uses only an 8 row buffer of pixels. In contrast, traditional MRC-based compression algorithms buffer image strips that typically require large memory buffers, and are therefore not easily implemented in imaging pipeline hardware.



(a)

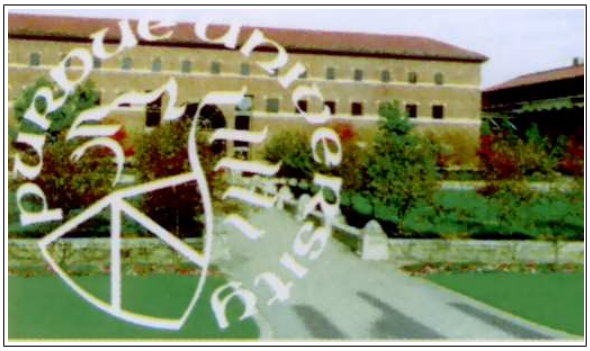


(b)

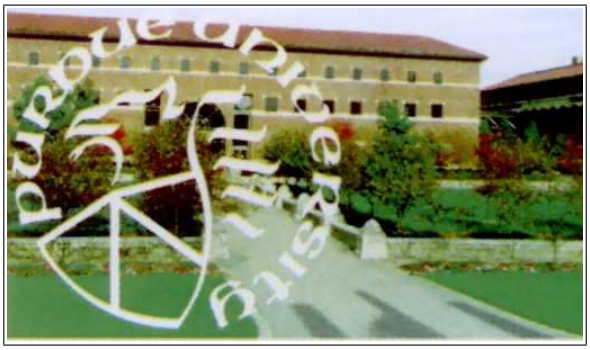


(c)

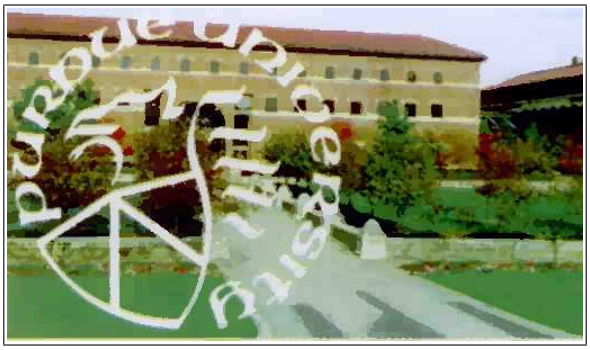
Fig. 2.12. Comparison between images compressed using the JPEG encoder and the MCC encoder. (a) A portion of the original test image "colortext6". (b) A portion of the reconstructed image compressed with JPEG; achieved bit rate is 0.570 bpp (42:1 compression). (c) A portion of the reconstructed image compressed with MCC; achieved bit rate is 0.303 bpp (79:1 compression).



(a)



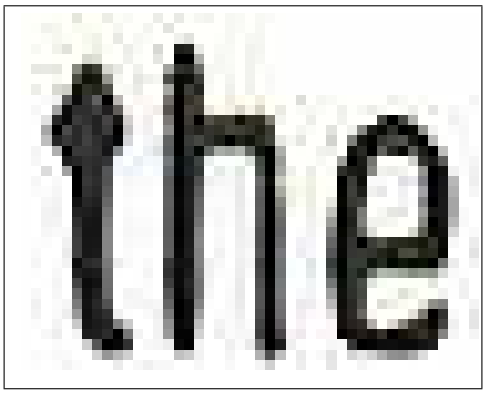
(b)



(c)

Fig. 2.13. Comparison between images compressed using the JPEG encoder and the MCC encoder. (a) A portion of the original test image "colormix300dpi". (b) A portion of the reconstructed image compressed with JPEG; achieved bit rate is 0.678 bpp (35:1 compression). (c) A portion of the reconstructed image compressed with MCC; achieved bit rate is 0.448 bpp (54:1 compression).

(a)



(b)

(c)

Fig. 2.14. Comparison between images compressed using the JPEG encoder and the MCC encoder. (a) A portion of the original test image "monotext12". (b) A portion of the reconstructed image compressed with JPEG; achieved bit rate is 1.061 bpp (23:1 compression). (c) A portion of the reconstructed image compressed with; achieved bit rate is 0.420 bpp (57:1 compression).

(a)

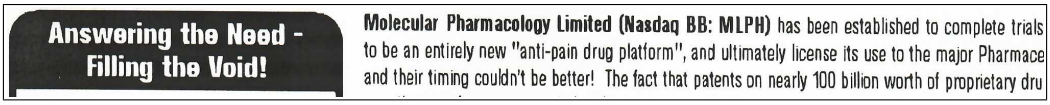
(b)

(c)

Fig. 2.15. Comparison between images compressed using the JPEG encoder and the MCC encoder. (a) A portion of the original test image "colortext6". (b) A portion of the reconstructed image compressed with JPEG; achieved bit rate is 0.570 bpp (42:1 compression). (c) A portion of the reconstructed image compressed with MCC; achieved bit rate is 0.303 bpp (79:1 compression).

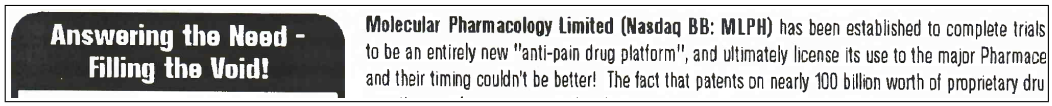
(a) (b) (c)

Fig. 2.16. Comparison between images compressed using the JPEG encoder and the MCC encoder. (a) A portion of the original test image "colormix110". (b) A portion of the reconstructed image compressed with JPEG; achieved bit rate is 0.893 bpp (27:1 compression). (c) A portion of the reconstructed image compressed with MCC; achieved bit rate is 0.491 bpp (49:1 compression).



(a)

(b)



(c)

Fig. 2.17. Comparison between images compressed using the JPEG encoder and the MCC encoder. (a) A portion of the original test image "monotext12". (b) A portion of the reconstructed image compressed with JPEG; achieved bit rate is 1.061 bpp (23:1 compression). (c) A portion of the reconstructed image compressed with; achieved bit rate is 0.420 bpp (57:1 compression).

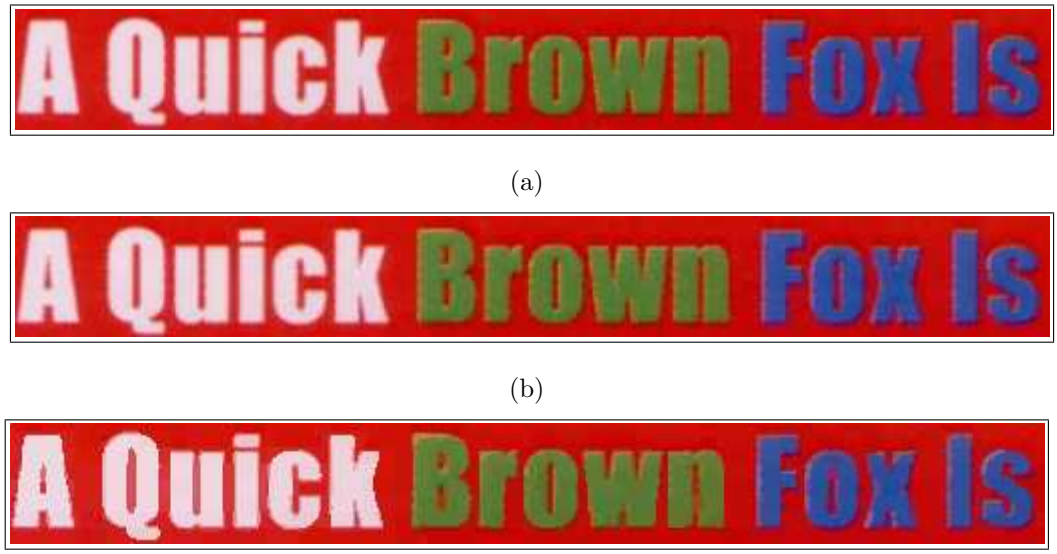


Fig. 2.18. Comparison between images compressed using the JPEG encoder and the MCC encoder. (a) A portion of the original test image "colormix300dpi". (b) A portion of the reconstructed image compressed with JPEG; achieved bit rate is 0.678 bpp (35:1 compression). (c) A portion of the reconstructed image compressed with MCC; achieved bit rate is 0.448 bpp (54:1 compression).

2.4 Conclusion

The mixed content compression (MCC) algorithm developed in this research was primarily designed to provide an efficient and cost-effective solution for compression of scanned compound document images. MCC allows for an easy implementation in imaging pipeline hardware by using only an 8 row buffer of pixels. MCC uses the JPEG encoder to effectively compress the background and picture content of a document image, and the JBIG1 encoder to compress the remaining text and line graphics in the image, which contain high-resolution transitions that are essential to text and graphics quality. To separate the text and graphics from the image, MCC uses a simple MSE block classification algorithm to allow a hardware efficient implementation.

Results show that our proposed MCC algorithm sharpens text and reduces halo around text and vector objects, and can reduce the average bit rate by between 22% and 47% relative to that of JPEG. Hence, by using MCC, we can increase the compression ratio average by between 29% and 88% relative to that of JPEG, while achieving similar or better quality. Our MCC algorithm is particularly suitable for text images and mixed images, which contain both text and picture content. We believe that MCC can achieve lower bit rates than JPEG by first removing the text and graphics from the original scanned image, and then encoding them separately using JBIG1. MCC can then use a JPEG encoder with high quantization step sizes to compress the background and picture content, thus making up for the overhead introduced by encoding text and vector objects with JBIG1.

LIST OF REFERENCES

LIST OF REFERENCES

[1] *Mixed Raster Content (MRC)*, ITU-T Recommendation T.44 (2005).

[2] *Information technology - Lossy/lossless coding of bi-level images*, ITU-T Recommendation T.88 — ISO/IEC 14492:2001 (2000).

[3] *Standardization of Group 3 facsimile terminals for document transmission*, ITU-T Recommendation T.4 (1980).

[4] *Facsimile coding schemes and coding control functions for Group 4 facsimile apparatus*, ITU-T Recommendation T.6 (1988).

[5] *Information technology Coded representation of picture and audio information Progressive bi-level image compression*, ITU-T Recommendation T.82 — ISO/IEC 11544:1993 (1993).

[6] *Application profile for Recommendation T.82 - Progressive bi-level image compression (JBIG coding scheme) for facsimile apparatus*, ITU-T Recommendation T.85 (1995).

[7] *Application profiles for Recommendation T.88 - Lossy/lossless coding of bi-level images (JBIG2) for facsimile*, ITU-T Recommendation T.89 (2001).

[8] P. Howard, "Lossless and Lossy Compression of Text Images by Soft Pattern Matching," in *Proc. of the 1996 IEEE Data Compression Conference* (1996), pp. 210–219.

[9] I.H. Witten, A. Moffat, and T.C. Bell, *Managing Gigabytes: Compressing and indexing documents and images* (Morgan Kaufmann, 1999), 3rd ed.

[10] *Xerox Proposal for JBIG2 Coding*, (1996).

[11] Y. Ye, D. Schilling, P. Cosman, and H. Koh, "Symbol dictionary design for the JBIG2 standard," in *Proc. of the 2000 IEEE Data Compression Conference* (2000), pp. 33–42.

[12] Y. Ye and P. Cosman, "Dictionary design for text image compression with JBIG2," *IEEE Trans. on Image Processing* **10**(6), 818–828 (2001).

[13] Y. Ye and P. Cosman, "JBIG2 symbol dictionary design based on minimum spanning trees," in *Proc. of the First Intl. Conf. on Image and Graphics* (2000), pp. 54–57.

[14] Y. Ye and P. Cosman, "Fast and memory efficient JBIG2 encoder," in *Proc. of 2001 IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing* (2001), vol. 3, pp. 1753–1756.

[15] Y. Ye and P. Cosman, "Fast and memory efficient text image compression with JBIG2," *IEEE Trans. on Image Processing* **12**(8), 944–956 (2003).

[16] M. Figuera, J. Yi, and C.A. Bouman, "A new approach to JBIG2 binary image compression," in *Proc. of SPIE - The International Society for Optical Engineering* (2007), vol. 6493, *Color Imaging XII: Processing, Hardcopy, and Applications*, pp. 649305.

[17] R. N. Ascher and G. Nagy, "Means for Achieving a High Degree of Compaction on Scan-digitized Printed Text," *IEEE Trans. on Computers* **C-23**(11), 1174–1179 (1974).

[18] K. Mohiuddin, J. Rissanen, and R. Arps, "Lossless Binary Image Compression Based on Pattern Matching," in *International Conference on Computers, Systems and Signal Processing* (1984), pp. 447–451.

[19] Y. Ye, *Text Image Compression Based on Pattern Matching*, PhD dissertation, University of California, San Diego, USA (2002).

[20] L.A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Systems Journal* **5**(2), 78–101 (1966).

[21] A.V. Aho, P.J. Denning, and J.D. Ullman, "Principles of optimal page replacement," *Journal of the Association for Computing Machinery* **18**(1), 80–93 (1971).

[22] S.J. Inglis, *Lossless Document Image Compression*, PhD dissertation, The University of Waikato, Hamilton, New Zealand, (1999).

[23] Q. Zhang and J.M. Danskin, "Bitmap reconstruction for document image compression," in *Proc. SPIE Multimedia Storage and Archiving Systems* (1996), vol. 2916, pp. 188–199.

[24] Q. Zhang, J.M. Danskin, and N.E. Young, "A codebook generation algorithm for document image compression," in *Proc. of the 1997 IEEE Data Compression Conference* (1997), pp. 300–309.

[25] M.J. Holt, "A fast binary template matching algorithm for document image data compression," in *Proc. 4th Int. Conf. on Pattern Recognition* (Cambridge, UK, 1988), vol. 301, pp. 230–239.

[26] W.K. Pratt, P.J. Capitant, W.H. Chen, E.R. Hamilton, and R.H. Wallis, "Combined symbol matching facsimile data compression system," in *Proc. of the IEEE* (1980), vol. 68, pp. 786–796.

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

[28] E. Haneda, J. Yi, and C. Bouman, "Segmentation for MRC compression," in *Proc. of SPIE* (2007), vol. 6493.

[29] M. Nelson and J. Gailly, *The Data Compression Book* (M&T Books, 1995), 2nd ed.

[30] *Portable Document Format: Changes from Version 1.3 to 1.4*, Adobe Systems Incorporated, 2001.

[31] *JBIG2 Primer - JBIG2: The Compression Connection*, CVISION Technologies Website, 2008.
`http://www.cvisiontech.com/jbig2_primer_compression_connection.html`

[32] *Information Technology - Digital Compression and Coding of Continuous-Tone Still Images - Requirements and Guidelines*, ITU-T Recommendation T.81 — ISO/IEC 10918-1 (1992).

[33] D. Mukherjee, C. Chrysafis, and A. Said, "Low Complexity Guaranteed Fit Compound Document Compression," in *Proc. of the 2002 IEEE International Conference on Image Processing* (2002), vol. 1, pp. I-225–I-228.

[34] H. Cheng and C. A. Bouman, "Document Compression Using Rate-Distortion Optimized Segmentation," *Journal of Electronic Imaging* **10**(2), 460–474 (2001).

[35] H. Siddiqui, M. Boutin, and C. A. Bouman, "Hardware-Friendly Descreening," To appear in the *Journal of Electronic Imaging*.

VITA

VITA

Maribel Figuera was born in Lleida, Spain. She received her B.S. in Telecommunications Engineering in 2001 from Universitat Politècnica de Catalunya (UPC, Barcelona) and her M.S. and Ph.D. degrees in Electrical & Computer Engineering in 2007 and 2008, respectively, from Purdue University. She has been with Microsoft Corporation since July 2008. She is a member of the Institute of Electrical and Electronics Engineers (IEEE), the Society of Women Engineers (SWE), and the Eta Kappa Nu (HKN) Electrical and Computer Engineering Honor Society. Her research interests include image and video processing, image and video compression, electronic imaging systems, multimedia systems, signal processing, and communications.