Dictionary Design for Text Image Compression with JBIG2

Yan Ye, *Student Member, IEEE,* and Pamela Cosman, *Senior Member, IEEE*

***Abstract*—The JBIG2 standard for lossy and lossless bilevel image coding is a very flexible encoding strategy based on pattern matching techniques. This paper addresses the problem of compressing text images with JBIG2. For text image compression, JBIG2 allows two encoding strategies: SPM and PM&S. We compare in detail the lossless and lossy coding performance using the SPM-based and PM&S-based JBIG2, including their coding efficiency, reconstructed image quality and system complexity. For the SPM-based JBIG2, we discuss the bit rate tradeoff associated with symbol dictionary design. We propose two symbol dictionary design techniques: the class-based and tree-based techniques. Experiments show that the SPM-based JBIG2 is a more efficient lossless system, leading to 8% higher compression ratios on average. It also provides better control over the reconstructed image quality in lossy compression. However, SPM's advantages come at the price of higher encoder complexity. The proposed class-based and tree-based symbol dictionary designs outperform simpler dictionary formation techniques by 8% for lossless and 16–18% for lossy compression.**

***Index Terms*—Bilevel image coding, JBIG2, soft pattern matching, symbol dictionary, text image compression.**

I. Introduction

THE JBIG2 standard [1], [2] is the new international standard for bilevel image compression. Bilevel images have only one bit-plane, where each pixel takes one of two possible colors. Prior to JBIG2, compression of bilevel images was addressed by facsimile standards such as ITU-T recommendations T.4, T.6, and T.82 (JBIG1) [3], [4]. However, these recommendations provide only for lossless compression of bilevel images. JBIG2 is the first one that also provides for lossy compression. A properly designed JBIG2 encoder not only achieves higher lossless compression ratios than the other existing standards, but also enables very efficient lossy compression with almost unnoticeable information loss.

Another main advantage of JBIG2 is its progressivity. JBIG2-compliant bitstreams can be made progressive in two senses: *quality progressive* and *content progressive*. A quality progressive bitstream first provides a lossy representation of the original image at a relatively low bit rate, then successively refines it to less lossy versions, until finally the reconstructed image becomes strictly lossless. A content progressive bitstream orders the different types of image content, for example, first text, then halftones, and finally general graphics (e.g., line art).

For these reasons, besides the obvious facsimile application, JBIG2 will be useful in a variety of different applications, including document storage and archiving, the World Wide Web, wireless communications, and printing. To accomodate the various application needs, JBIG2 provides a standardized toolkit from which each application can select different parts based on its own system resources and requirements.

A typical JBIG2 encoder will first segment an image into different regions [5] and then use different coding mechanisms for text and for halftones. In this paper, we are concerned with compressing text images, defined as bilevel images which consist mainly of repeated text characters and possibly some general graphic data (e.g., line art) but no halftones. A JBIG2 encoder uses pattern matching techniques to efficiently compress the textual regions. For general graphic data not identified as text, the encoder will use a cleanup coder which is essentially a basic bitmap coder such as specified by JBIG1 or T.6.

To encode a page of text, rather than coding all the pixels of each occurrence of each character, we code the bitmaps of a representative subset and put them into the *symbol dictionary*. We define a *symbol* as one black connected component. We can extract symbols from the input image using a standard segmentation algorithm [6]. Usually one symbol represents an instance of a certain alphanumeric character; however, characters that contain separated parts, e.g., English letter "i," will be extracted as more than one symbol. Based on how the symbol dictionary is formed and how symbol bitmaps are compressed, JBIG2 defines two modes for the compression of text: *pattern matching and substitution* (PM&S) [7] and *soft pattern matching* (SPM) [8].

Viewing the bitmap of each symbol as a binary pattern, we can measure the similarity between two symbols using standard pattern matching techniques. When a certain matching criterion declares a match between two symbol bitmaps that actually represent two different characters, we say a substitution error has occurred. For the PM&S-based JBIG2 system, as we will explain shortly, adopting a pattern matching criterion that is robust against substitution errors is particularly important. In [9], a pattern matching technique is introduced that is robust against character substitutions but very computationally demanding. In the JBIG2 system we built, we adopt the Hamming distance matching criterion. We measure the percentage of different pixels between two symbols (binary patterns). In order to control substitution errors, we set a low mismatch threshold.

Manuscript received June 29, 2000; revised February 22, 2001. This work was supported by the National Science Foundation under Grants MIP-9617366 and MIP-9624729 (CAREER), and by the Center for Wireless Communications, University of California at San Diego, La Jolla. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Touradj Ebrahimi.

The authors are with the Electrical and Computer Engineering Department, University of California at San Diego, La Jolla, CA 92093-0407 USA (e-mail: yye@code.ucsd.edu; pcosman@code.ucsd.edu; http://www.code.ucsd.edu/cosman/).

Publisher Item Identifier S 1057-7149(01)04478-5.

1057–7149/01$10.00 © 2001 IEEE

In Section III, we will discuss how to choose representative patterns to form the dictionary for an SPM-based JBIG2 encoder. Prior to our work, Zhang *et al.* also addressed codebook design for text image compression [10], [11]. Their methods are not directly comparable to ours for two reasons. Firstly, PM&S, as opposed to SPM, is used as their framework. Secondly, their work is not related to JBIG2, and so a number of system components, e.g., location coding, dictionary structure, and index coding, differ significantly from ours.

This paper is organized as follows. In Section II we elaborate on the two text compression modes in JBIG2. In Section III we investigate the problem of symbol dictionary design for SPM-based JBIG2. In Section IV we present our experimental results in terms of compression performance and system complexity. We conclude the paper in Section V.

II. Text Image Compression with JBIG2

In this section we explain in detail the two text coding modes supported by JBIG2: PM&S and SPM. First, we will explain how these two systems work and their respective advantages and disadvantages in lossless and lossy compression. Then, we will explain how JBIG2 organizes the different types of information into segments for text image compression.

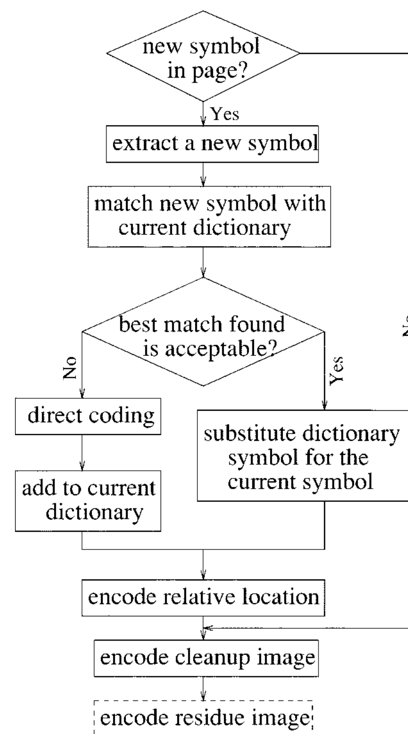

Fig. 1. Flow chart of a typical PM&S system.

A. Pattern Matching and Substitution

Ideally, in the symbol dictionary, we want one and only one symbol bitmap to represent one certain character, such as the letter "b." This way each time there is another instance of the letter "b," we can use a pointer to the "b" in the symbol dictionary to describe the bitmap of the current "b." This leads to the idea of pattern matching and substitution (PM&S) [7]. In PM&S, a pattern matching criterion is first chosen to measure the mismatch between two symbols. To code a new symbol, we look for the dictionary entry which has the smallest mismatch with the symbol to be coded. If that smallest mismatch is less than the preset threshold, we code the symbol by using a pointer to the dictionary entry. If the closest dictionary entry is too far away, we code the new symbol bitmap with a JBIG1 or T.6 type of entropy encoder (called *direct coding*). Then we add the new symbol to the dictionary. Fig. 1 describes the typical coding procedure of a PM&S system.

Use of PM&S allows high compression ratios for text images that have many repeated symbols. However, it is inherently lossy and there are inevitable substitution errors. How frequently these occur depends solely on the pattern matching criterion and threshold selected. If these errors are not acceptable, JBIG2 allows the use of a residual coder which uses *refinement coding* [12]. Refinement coding refers to coding the original image again with an arithmetic coder using context information from both the lossy image transmitted and the already coded part of the original image. The overall compression thus achieved is better than that obtained by coding the original image by JBIG1. We will discuss refinement coding further when we introduce SPM. Besides refining the lossy image to its strictly lossless version, the encoder can also elect to refine it merely to a less lossy version.

B. Soft Pattern Matching

As an alternative to PM&S, JBIG2 allows soft pattern matching (SPM) [8]. Fig. 2 shows the original SPM system block diagram as proposed in [8]. In SPM, if no match is found in the symbol dictionary, the current bitmap is coded with direct coding as in PM&S. But when a matching dictionary entry is found for the current symbol, SPM differs from PM&S in coding the new symbol with refinement coding and adding it to the dictionary.

In Fig. 3, we show the JBIG2-compliant context templates for direct and refinement coding used in our implementation. In refinement coding, to predict the color of the current pixel, the arithmetic coder not only uses knowledge about its causal neighbors, but also draws information from the causal and non-causal neighbors from the corresponding pixel in the reference bitmap. Therefore, refinement coding achieves more accurate prediction and hence higher compression ratios. Naturally, in order to improve the prediction accuracy in refinement coding, the reference bitmap should be a close match to the current one. However, we note that even using a totally mismatched reference symbol merely leads to reduced compression efficiency, not to any substitution errors. The residual coder mentioned earlier uses refinement coding on the entire image level rather than on the symbol level. To transmit the current image (either the lossless one or a less lossy one), the residual coder uses knowledge from both the already coded portion of the current image and the lossy image transmitted to make prediction more accurate.

Without the two dashed boxes in Fig. 2, SPM is a strictly lossless system. To achieve lossy compression with SPM, [8] proposed three preprocessing techniques to introduce information loss in a restricted manner. These techniques are called *speck*

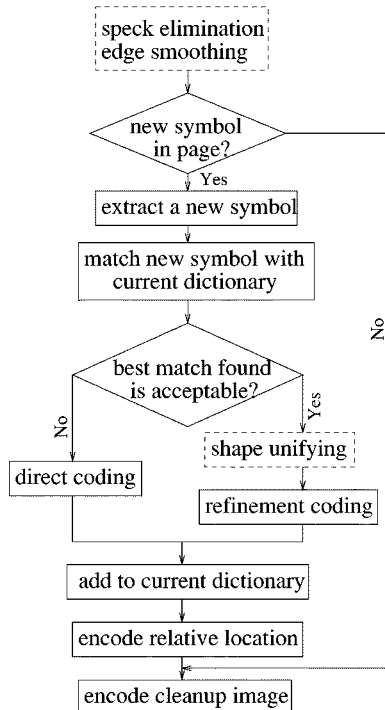

Fig. 2. Flow chart of the original SPM system.

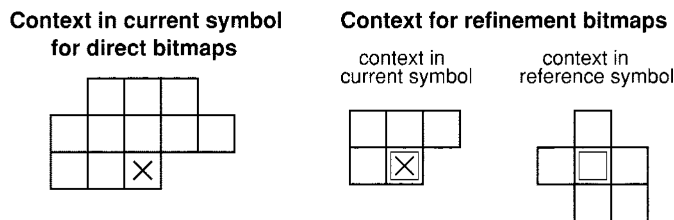

Fig. 3. Contexts used for direct and refinement coding: To code the pixel marked X, the binary arithmetic coder forms a context using the neighboring pixels.

elimination, *edge smoothing*, and *shape unifying*. Speck elimination wipes out very tiny symbols or flying specks (symbols no bigger than 2×2). Edge smoothing fixes jagged edges by flipping protruding single black pixels or indented single white pixels along text edges. Shape unifying (see Fig. 4) tries to make the current symbol's bitmap as similar as possible to its reference bitmap, without introducing "visible" changes. This is achieved by flipping pixels in the current bitmap if they are isolated areas of difference with the reference bitmap. Throughout this paper, we use the term "isolated" to mean a 1×1, 1×2, or 2×1 block of pixels. This modified current bitmap is then refinement coded losslessly.

Shape unifying gives the largest amount of improvement over compressing losslessly. Our experiments show that using only the first two techniques together can improve compression by around 8% over the strictly lossless case, while adding shape unifying provides approximately 40% improvement. Shape unifying is so efficient because isolated differences between the current bitmap and its reference are very detrimental to arithmetic coding efficiency (even more so than clustered differences); shape unifying completely eliminates such events. Furthermore, except in the special case of very tiny symbols, these preprocessing approaches do not introduce substitution errors. The loss of visual information is almost imperceptible. Compared to PM&S where the error image may contain many clustered errors, SPM-based lossy compression generates mostly dispersed errors. The quality of such reconstructed images is better from the point of view of character recognition and visual appearance.

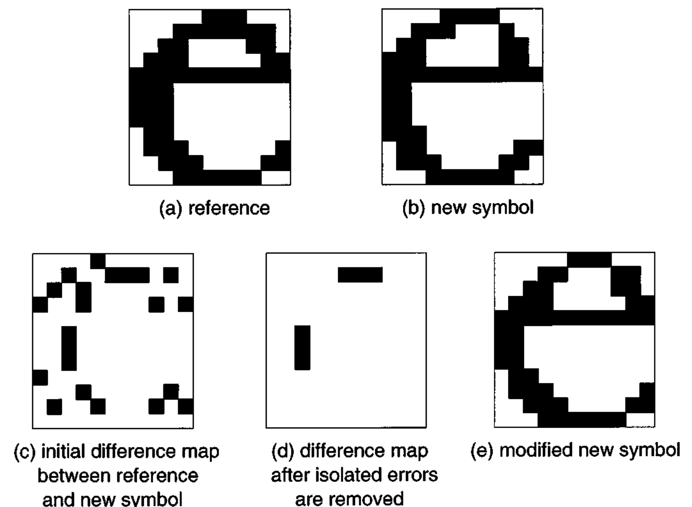

Fig. 4. Example of shape unifying. The modified new symbol (e) is losslessly coded.

If a lossless or less lossy version of the original image is to be requested later, as in the PM&S case, the SPM encoder can also use a residual coder to efficiently transmit the original or the less lossy image using the lossy version already sent.

Notice that speck elimination and edge smoothing are truly preprocessing techniques. Shape unifying, on the other hand, is not strictly a preprocessing procedure since it is not done until after symbols find their best matches in the dictionary, which, for the proposed dictionary design techniques, will not happen until the entire dictionary itself is finally decided. Once the entire symbol set is altered with these three approaches, the page is compressed losslessly; no further loss will be introduced.

Perfect Symbols: For symbols that are matched in the dictionary, PM&S transmits them using only an index pointer. SPM transmits not only the index pointer, but also the refinement coded bitmap. Therefore, PM&S encodes faster. However, in the lossy SPM case, the bitmap coding procedure can be sped up by taking advantage of *perfect symbols*. Perfect symbols are those refinement symbols that become identical to their references after shape unifying. That is, the block in Fig. 4(d) is blank. For perfect symbols, the following two facts hold true:

1) there is no need to send a symbol's bitmap with refinement coding if it is perfect;
2) a dictionary symbol that becomes perfect can be removed from the dictionary.

Obviously, 1) saves compression time by skipping the expensive arithmetic coding procedure. Experimental results in Section IV will show that shape unifying can sometimes render

a significant percentage of perfect symbols, thereby compensating for the higher complexity of an SPM encoder. As will be discussed later, 2) can further improve compression by reducing the dictionary size and hence bits spent on index coding.

C. JBIG2 Segments

When a symbol bitmap is extracted from the input image, its location information is obtained at the same time. These different types of information are organized into JBIG2 segments. For a typical page of text, a group of dictionary symbols will be transmitted first in the *symbol dictionary segments* [1]. There are two types of symbol dictionaries: *direct dictionary* and *refinement/aggregate dictionary*. Entries in the direct dictionary are symbols that do not refer to any other symbols. Their bitmaps are Huffman coded or arithmetically coded using direct coding. Entries in the refinement/aggregate dictionary are symbols that refer to some other dictionary symbol (either in the direct or in the previous part of the refinement/aggregate dictionary). Their bitmaps are arithmetically coded using refinement/aggregate coding. Aggregate coding is a special mode of refinement coding that allows the current bitmap to use two or more bitmaps at the same time as its reference. Symbols in the dictionary segments are arranged into classes with the same height; inside one height class symbols are usually sorted by width, making transmittal of size information more efficient.

After the dictionary segments, next are encoded the *text region segments*, in which the makeup of the input page is described to the decoder. Each text region segment consists of narrow horizontal coding strips. In the transpose mode, coding strips are vertical. The symbol instances whose reference corners (e.g., the lower left corner) lie inside the coding strip are encoded together. To describe each symbol instance, several things are transmitted: its vertical offset relative to the top of the current strip, its horizontal offset relative to the previous symbol instance, its dictionary index, a one-bit flag to signal whether its bitmap is refined from the dictionary symbol it refers to, and its refinement coded bitmap if the previous bit is one. We call the refinement coding of bitmaps that happens in the text region segments *embedded coding*. In PM&S, no embedded coding is used.

Since a typical text image can also contain line art that is not identified as text, such general graphics are encoded in the *generic region segments*. Generic region segments are encoded with the cleanup coder, a basic bitmap coder such as specified in JBIG1 or T.6.

If the page image described so far is lossy, and the lossless image or a less lossy one is desired, the encoder transmits the refinement information in *generic refinement region segments* using a residual coder.

In Fig. 5, we show the order in which different types of JBIG2 segments are sent in PM&S and SPM systems. We note that the PM&S system does not need to send a refinement dictionary segment.

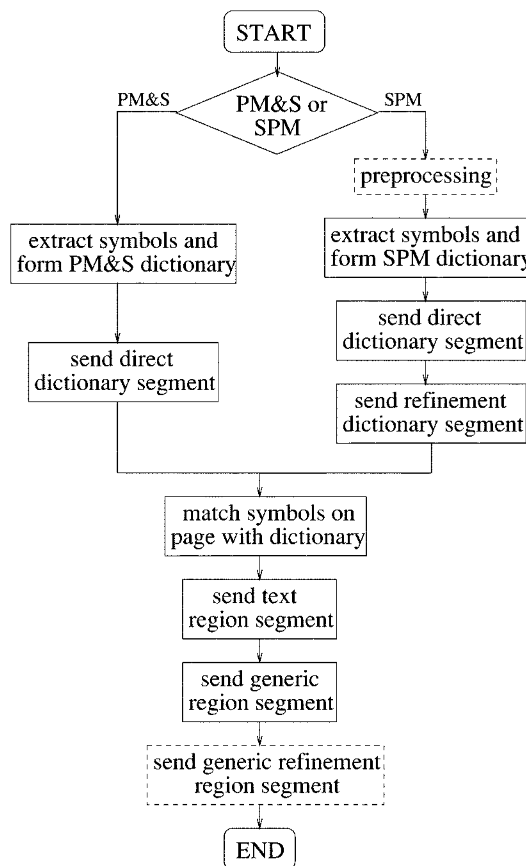

Fig. 5. PM&S- and SPM-based JBIG2 systems.

III. Symbol Dictionary Design for SPM-Based JBIG2

For a PM&S-based JBIG2 encoder, ideally, we want one and only one symbol bitmap in the dictionary to represent one certain character. This way all distinct characters get represented and no bitmap and index resource is wasted representing some characters repeatedly. For an SPM-based JBIG2 encoder, the goal is not as obvious. In SPM, a matching dictionary symbol is not directly substituted for the current symbol; rather, it provides useful prior knowledge to guide the refinement coding of the current bitmap. A more accurate match means less expensive refinement coding. Therefore, the symbol dictionary should be seen as a pool of useful reference information rather than as a collection of distinct characters that occur in the image. An SPM dictionary can have several symbols inside it representing the same character, as long as each provides some useful reference information. Furthermore, sometimes it may be efficient to refine a symbol representing one character from a symbol representing a different character. For example, the letters “c” and “o” are similar enough that the current “c” is better off being coded by refining the previous “o” than by direct coding.

In SPM, the basic bit rate tradeoff associated with symbol dictionary design is as follows. If one puts more symbols into the dictionary, all dictionary symbols have a longer index. Thus *index coding* will require more bits. However, with more dictionary symbols, all symbols will have a broader range of choices for more accurate matches, and hence *refinement coding* can be made less expensive. Therefore, to resolve the tradeoff at a favorable point, our goal is to design a reasonably small dictionary that can still provide comprehensive and accurate reference information. This involves not only choosing a good rep-

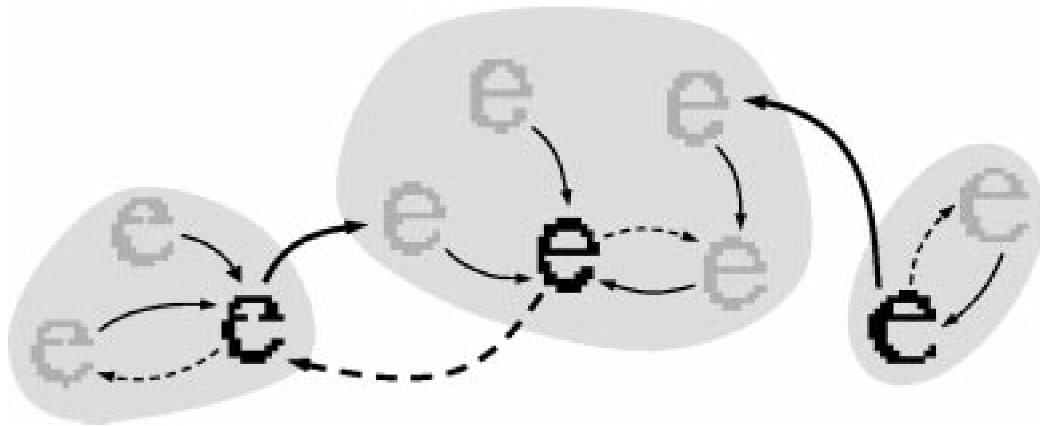

Fig. 6. Super-class merging process. Shaded areas indicate three existing super-classes. Black symbols are super-class leaders. Thin arrows show the reference relationship within each super-class. Removal of the dashed thin arrows breaks the unique loop in each super-class. Thick arrows show how the three leaders are matched at the next round. Removal of the dashed thick arrow breaks the new loop. The three existing super-classes are merged into one new super-class. The new leader is the black "e" in the middle.

resentative small subset from the entire extracted symbol set as the dictionary, but also deciding the reference relationships among dictionary symbols so that we can efficiently compress the dictionary itself. That is, how should we decide the reference relationship between them in such a way that each dictionary symbol can have an accurate reference and thus be compressed efficiently?

In this section, we will first introduce two simple existing methods for dictionary formation, and then we will elaborate on the two proposed dictionary design techniques, one based on the concept of classes, the other based on minimum spanning trees.

A. Simple Methods for Dictionary Formation

As shown in Fig. 2, the original SPM system proposed by Howard [8] described a sequential way to form the dictionary. As each symbol is extracted from the page, its bitmap is matched with the existing dictionary entries (this match is prescreened by size) and transmitted with direct or refinement coding depending on whether the best match found in the dictionary is acceptable. Then, the new symbol is added to the dictionary.[1] The dictionary size keeps increasing as the page gets processed. By the end of the page, the dictionary contains every extracted symbol. We refer to this as *one-pass* dictionary formation.

The one-pass dictionary inevitably contains *singletons* [6], [13], i.e., dictionary entries that are never referenced by other symbols. They are detrimental to compression efficiency in that they provide no useful reference information, yet dictionary indices are allocated to them, thereby increasing the length of all dictionary indices. Singletons are of two types. *Direct singletons* are symbols that neither refer to other symbols nor are referred to by other symbols; usually they are "strange" symbols in the image, e.g., a company logo in a business letter. Direct singletons can be removed from the direct dictionary and relegated to generic region segments (the cleanup coder) to be coded together with other general graphics. *Refinement singletons* are symbols that refer to other symbols but are not referred to by any one else. They can be removed from the refinement dictionary and relegated to the text region segments for embedded coding. We will refer to the dictionary formation method that excludes direct and refinement singletons as the *singleton exclusion* dictionary. Forming this singleton exclusion dictionary is still essentially one-pass in the sense that each symbol finds its best match only among previous symbols.

[1]Later in [2], the authors described the SPM system as conditionally adding the new symbol into the dictionary after coding its bitmap but did not give more detail. Here, we only consider the simpler case where each symbol is added into the dictionary.

B. Class-Based Symbol Dictionary Design

Because a JBIG2 encoder has access to the entire set of extracted symbols before any actual coding is carried out, each symbol can find its best match among *all* other symbols. Therefore each symbol can potentially find a more accurate reference symbol. In our class-based symbol dictionary design technique [14], we define a class to be a set of symbols for which

1) each symbol in a class has its best match symbol also within the class;
2) there is no way of subdividing the class into subgroups in such a way that each symbol in a subgroup still has its best match in the subgroup.

One way to partition the entire symbol set into classes is to point each symbol to its best match. This way the symbol set is partitioned into small pieces of connected graphs with very similar symbols clustered together. These connected graphs have weighted edges where the weights are the mismatch scores between each pair of symbols. Since class members are very similar, only one dictionary symbol is needed to provide reference information for one class. This dictionary symbol is called the *class representative*. We choose the class representative as the symbol with the smallest average mismatch within the class.

After a dictionary consisting of class representatives has been formed, we follow a similar procedure recursively to efficiently compress the dictionary itself. First, each dictionary symbol is pointed to its best match among other dictionary symbols, so the dictionary symbols are partitioned into small connected graphs. Each graph has exactly one loop inside; we call these connected graphs *super-classes* (see Fig. 6). Each loop is broken by eliminating the edge with the highest

weight (i.e., biggest mismatch score); this generates one super-class leader for each super-class. At the next round, these super-class leaders go out and find their best matches among other super-classes (not just other super-class leaders). In this way super-classes are merged together. For each merge we will have another unique loop within the newly generated super-class, which is broken in the same manner as before. We repeat this "match–merge–break" iteration recursively until no super-class leader can find an acceptable best match among the other super-classes, i.e., the smallest mismatch score is no longer below the preset threshold. At this point the dictionary symbol set is partitioned into disjoint super-classes that are sufficiently dissimilar from each other. We have obtained a suboptimal solution to the reference relationship among the dictionary symbols. The final super-class leaders go into the direct dictionary and other class representatives go into the refinement dictionary. The natural order by which symbols are extracted has been changed, and we will need to reorder the dictionaries so that each symbol comes after its reference symbol.

C. Tree-Based Symbol Dictionary Design

In the dictionary design technique based on minimum spanning trees [15], we first consider all the extracted symbols as a set of vertices. We try to connect each pair of vertices with an edge whose weight is the mismatch score between the corresponding symbols. Because weights can not exceed the preset mismatch threshold, there is no edge connecting symbols that have unacceptable mismatch. Therefore, the entire extracted symbol set is represented by a number of undirected graphs. Strictly speaking, these graphs are not undirected as the mismatch from symbol A to symbol B is not necessarily the same as the mismatch from symbol B to symbol A. However, this difference is small, and so we treat the two scores as equal.

To obtain the reference relationship among all symbols suitable for compression, many edges in the graphs must be eliminated. This is because

1) each symbol needs only one symbol as its reference;
2) symbols can not refer to each other circularly, i.e., the final graphs must be acyclic.

The final matching graphs should be trees spanning all the symbols. Furthermore, for the compression to be efficient, the overall weight (i.e., mismatch) of the trees should be small. The minimal overall weight solution is given by the classic minimum spanning tree (MST) algorithm. We use Kruskal's Algorithm [16]. First, we take out all the edges from the initial graphs, leaving only vertices. Then we put back the next edge with the smallest weight provided that it does not introduce cycle into any of the current graphs. We continue until we can add no more edges into the existing graphs. That gives us the final MSTs. Fig. 7 shows the initial matching graphs and the MSTs constructed. The isolated nodes in the picture represent "strange" symbols that resemble no other symbol; they are direct singletons and will be coded in generic region segments.

After the MSTs are obtained, we need to decide a root node for each MST. See the "a" tree and the "b" tree in Fig. 7. We would like to choose the root that gives the lowest refinement

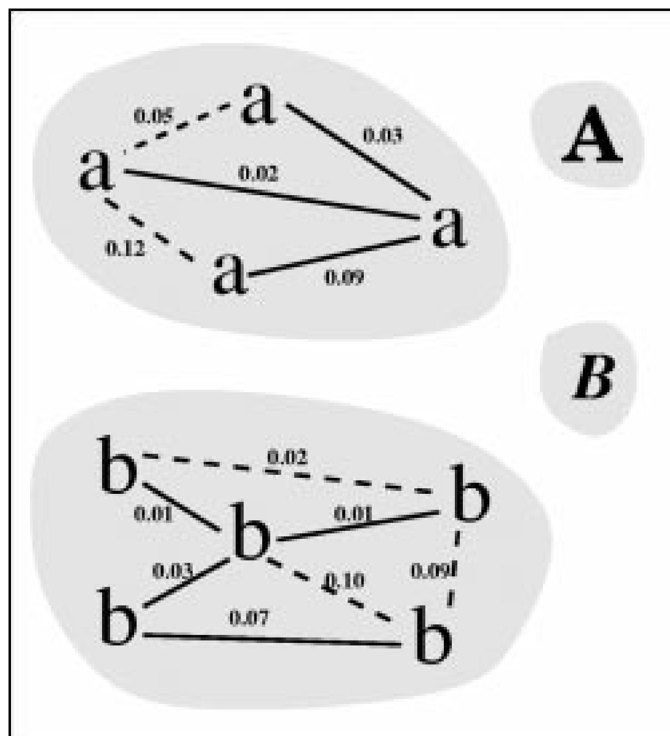

Fig. 7. Minimum spanning trees constructed from original matching graphs. Broken edges existed in the original graphs but are discarded by Kruskal's algorithm.

and index coding cost. For refinement coding, once an MST is constructed, the total amount of mismatch this tree carries (this is defined as the sum of the weights of all the edges) is fixed regardless of the choice of root. Although mismatch score based on Hamming distance does not correspond directly to refinement coding efficiency, experimentally it is observed that this mismatch score is a fairly accurate indicator of the actual refinement compression efficiency. Therefore, refinement coding should not be affected significantly by choosing different nodes as the root.

However, index coding cost does directly relate to the choice of root. Since the leaves of the trees are refinement singletons which will not get assigned dictionary indices, more leaves means lower index coding cost. For each MST, we choose arbitrarily any node with degree greater than 1 as its root (a node's degree is the number of connecting edges it has). This is because the following theorem is true (proof is trivial).

Theorem 1: For a tree with n nodes, m of which are degree-1 nodes, $m \leq n$, it will have $m-1$ leaf nodes if any degree-1 node is chosen as the root, and have m leaf nodes otherwise.

MSTs give us the optimal reference relationship between extracted symbols. After we have chosen the tree roots, we construct the symbol dictionary and calculate its size. Suppose altogether N symbols are extracted (i.e., the undirected graphs have N vertices). Using Kruskal's algorithm, T trees are constructed. Out of the T trees, $T - T_1$ trees have only one node, and T_1 have more than one node. For the T_1 trees with more than one node, assume we choose their roots as just described and get L leaf nodes in total. The symbols represented by the $T - T_1$ single-node trees are direct singletons. The L leaf nodes from

the T_1 multinode trees are refinement singletons. These singletons do not go into the dictionary. All the T_1 tree roots go into the direct dictionary; the remaining $N-T-L$ internal nodes go into the refinement dictionary.

D. Changing the Dictionary Size

Using "good" dictionaries (reasonably efficient suboptimal dictionaries) with different sizes, we can analyze the bit tradeoff problem, and try to resolve it at a favorable point for a particular page image. In this section, we explain how to modify sizes for the class-based and tree-based symbol dictionaries.

1) Class-Based Design: The mismatch threshold determines the number of classes and therefore the class-based dictionary size. The size can be modified by merging similar classes before putting their representatives into the dictionary. To do this we find the two existing classes that are the most similar, i.e., their representatives have the smallest mismatch. If these two classes are sufficiently similar, we merge them into a new one. We then choose the new representative as the symbol with the lowest average mismatch within the new class. We repeat this until a desired dictionary size is reached or until we can find no similar classes.

2) Tree-Based Design: The initial dictionary constructed from the MSTs has T_1 direct dictionary symbols and $N-T-L$ refinement dictionary symbols. The direct dictionary size cannot be changed unless we use a different mismatch threshold. The refinement dictionary size, however, is reducible by increasing the number of leaf nodes. To do this, we consider each internal node as a leaf candidate. To change an internal node to leaf, we need to relocate all its children to other internal nodes. For each child, its next best parent is chosen as the internal node that gives the next lowest mismatch and is not this child's offspring (otherwise we will introduce a cycle).

Therefore, for each internal node, we can define its *cost as leaf* as the sum of all the mismatch differences its children will suffer by being relocated to their new parents. Sometimes we may not find a new parent with an acceptable mismatch. In this case the child can not relocate; its parent's cost as leaf is infinite.

Fig. 8 illustrates the tree modification process. Starting from the initial MSTs, we calculate every internal node's cost. We choose the node with the lowest cost as leaf, relocate its children to their new parents, and make this node a new leaf. The dictionary size decreases by one. By doing so, the parent–offspring relationship for some nodes has changed and we need to recalculate their cost as leaf. We continue this greedy process until some target dictionary size is reached, or until no more valid leaf candidates can be found and the minimum dictionary size is attained. This allows us to obtain symbol dictionaries with almost arbitrary sizes.

IV. Experimental Results

In this section we show our experimental results on text image compression using JBIG2. Our encoder is fully JBIG2-compliant. Our test images are from two sources.

1) Two CCITT images that are mainly textual: f01_200 and f04_200. Their resolution is 200 dpi, size 1728×2339 pixels;

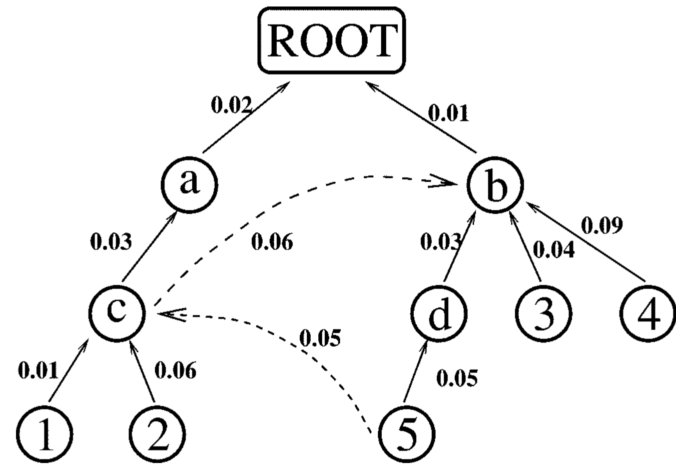

Fig. 8. Tree modification process. Solids arrows show the current tree. Lettered nodes are internal nodes. Numbered nodes are leaf nodes. The cost as leaf is 0.03 for "a" and 0 for "d." Node "d" is first turned into a leaf. Its child "5" goes to "c." Then "a" is turned into a leaf. Its child "c" goes to "b." Note that "c" takes its children "1," "2," and "5" together with it.

2) Ten images selected from the University of Washington Document Image Database I [17]. This database contains about 980 scanned document images. The 10 images we selected are mostly streak-free, not obviously skewed, from various sources, and contain mainly text, little line art and no halftones. All of them have 300 dpi resolution. Eight of the images have the same size 2592 × 3300 pixels, while N03H has size 2480 × 3508 and S012 2536 × 3308.

A. PM&S versus SPM

In this section, we compare the lossless and lossy coding performance of the PM&S-based and SPM-based JBIG2 systems. For the pattern matching criterion, we used the percentage of mismatch based on Hamming distance in both systems. For the PM&S system, our experiments show that a mismatch threshold of 0.1 (meaning that at most 10% of pixels have different colors) leads to rare substitution errors. Therefore, unless otherwise stated, we preset the mismatch threshold to 0.1 for PM&S. In the SPM system, the mismatch threshold is not directly related to the number of substitution errors, and our experiments show that the overall compression differs by less than 1% with different thresholds between 0.1 and 0.2. The optimal value occurs around 0.15; therefore 0.15 is used as the mismatch threshold for our SPM results.

1) Lossless Compression: Table I lists the total lossless bit rates and corresponding compression ratios on the 12 test images for the PM&S- and SPM-based JBIG2. For SPM, singleton exclusion dictionaries are used to obtain the bit rates shown in this table. (The more sophisticated SPM symbol dictionary design methods give further compression gains at the expense of greater complexity.)

From Table I we see that the SPM-based JBIG2 achieves more efficient lossless coding. The last row gives the total number of bytes needed for the two systems to compress the 12 test images losslessly. The total number of uncompressed bytes is 11 700 164. The compression ratios (CRs) are 23.5 for PM&S and 25.6 for SPM, respectively. This amounts to about

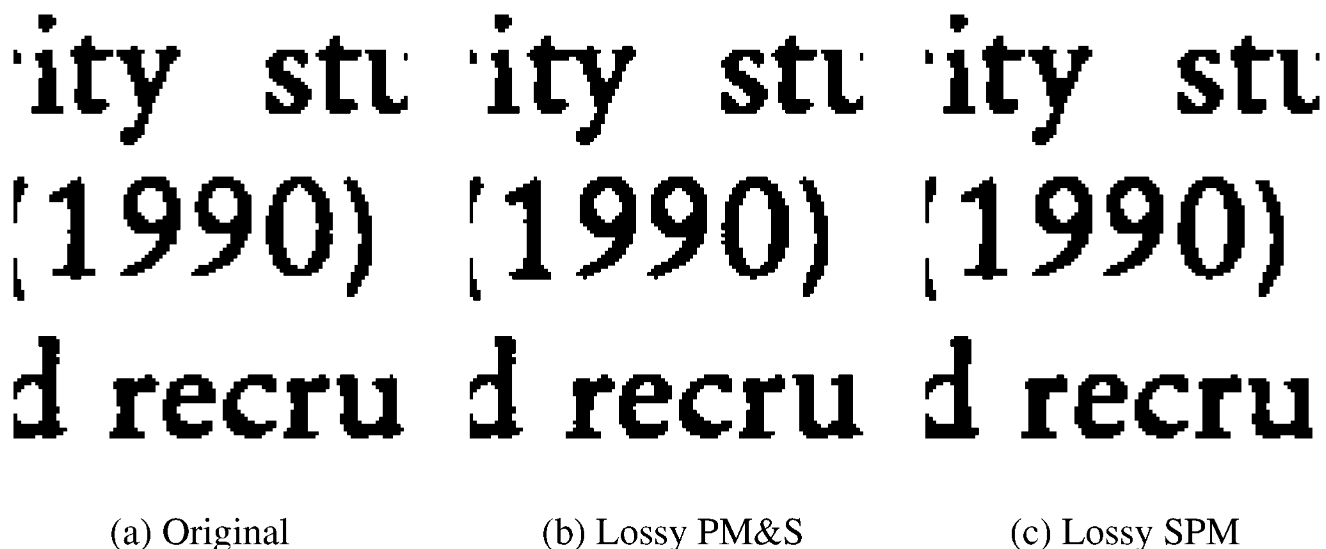

Fig. 9. Original and reconstructed images from lossy PM&S and SPM. Test image is S012.

TABLE I
COMPRESSED FILE SIZES (IN BYTES) FOR LOSSLESS PM&S AND SPM JBIG2 SYSTEMS

	PM&S		SPM (SINGL. EXCL.)	
	CODED SIZE	CR	CODED SIZE	CR
f01_200	10,762	46.9	10,484	48.2
f04_200	35,412	14.3	33,504	15.1
IG0H	47,262	22.6	43,635	24.5
J00O	50,709	21.1	45,393	23.6
N03F	38,527	27.8	35,029	30.5
N03H	34,343	31.7	31,794	34.2
N03M	56,441	18.9	51,601	20.7
N046	28,634	37.3	25,792	41.5
N04D	35,927	29.8	33,290	32.1
N04H	32,695	32.7	29,763	35.9
N057	55,039	19.4	50,095	21.3
S012	71,134	14.7	67,330	15.6
TOTAL	496,885	23.5	457,710	25.6

TABLE II
LOSSLESS COMPRESSION RATIOS FROM OUR ENCODER AND UBC/IMAGE POWER JBIG2 ENCODER

	PM&S		SPM	
	TOTAL BYTES	CR	TOTAL BYTES	CR
OURS	496,885	23.5	457,710	25.6
UBC/IMAGE POWER	566,369	20.7	498,097	23.5

8% better compression for SPM over PM&S. SPM is more efficient because it successively refines the symbol bitmaps on-the-fly by referring them to the best reference bitmaps it can find at the current moment. The PM&S encoder takes care of refinement coding only after all symbols have been transmitted at a lossy level. It then uses the residual coder to bring up the image quality. This is equivalent to refinement coding a group of symbols based on only one reference. Therefore, PM&S provides each refinement bitmap with less accurate reference information. Even though SPM usually generates a larger symbol dictionary and hence has higher index coding cost, it achieves better overall compression. A larger dictionary is also one reason why SPM has longer encoding time as we will show soon.

In Table II, we compare our lossless encoder with another JBIG2-compliant encoder by University of British Columbia and Image Power, Inc. This encoder is available at http://spmg.ece.ubc.ca/jbig2/ as binary executables for several OS platforms. Currently it has only lossless mode. Although detailed information about the UBC/Image Power encoder has not been made publicly available, we attempted to set the encoding parameters to values that correspond to our encoder. For example, the two systems use the same context templates for direct and refinement coding; they both have the same coding strip size of eight pixels; and they both treat symbols smaller than 2×2 as specks and code them with cleanup coding, etc. We see that our encoder compresses more efficiently.

2) Lossy Compression: To achieve lossy compression using SPM-based JBIG2, the encoder preprocesses the input image using the three techniques described in Section II-B. The encoder then encodes the preprocessed image losslessly. This leads to compression approximately 40% more efficient than the strictly lossless case, while guaranteeing the reconstructed image to be different from the original mostly at isolated pixel positions. For PM&S, using the Hamming distance matching criterion, we would not have this nice property because the encoder will indiscriminately introduce clustered errors as well as isolated ones. This can be seen by combining the results in Table III which lists the compressed file sizes in bytes and the percentages of flipped pixels and Table IV which lists the percentages of single-pixel errors, double-pixel errors and clustered errors (containing three or more connected error pixels). To obtain a valid comparison in Table III, we fine tune the mismatch threshold used in the PM&S system to make the compressed file sizes very close to those generated from SPM. From Tables III and IV we see that although the reconstructed PM&S images contain fewer error pixels, the percentages of clustered errors are significantly higher in PM&S than in SPM (27% compared to 4%). Isolated errors are not only less visually perceptible, but also less likely to cause character substitutions. The enlarged portions of the original image and the reconstructed ones from PM&S and SPM are compared in Fig. 9. Visually the SPM image looks closer to the original image. It also looks more appealing because edge smoothing removed the protruding or indenting single pixels, making the

TABLE III
COMPRESSED FILE SIZES (IN BYTES) AND PERCENTAGES OF FLIPPED PIXELS FOR LOSSY PM&S AND SPM SYSTEMS

	PM&S		SPM	
	CODED SIZE	% FLIPS	CODED SIZE	% FLIPS
f01_200	6,748	0.17	6,760	0.25
f04_200	18,980	0.72	18,947	1.22
IG0H	26,362	0.56	26,291	0.72
J00O	23,530	0.63	23,633	0.92
N03F	21,052	0.53	21,032	0.59
N03H	19,339	0.40	19,070	0.54
N03M	31,665	0.71	31,148	0.87
N046	17,338	0.35	17,301	0.37
N04D	19,174	0.48	19,036	0.57
N04H	19,437	0.36	19,522	0.43
N057	30,811	0.69	30,579	0.81
S012	38,341	0.89	38,147	1.13
TOTAL	272,777	6.49	271,466	8.42

TABLE IV
RECONSTRUCTED IMAGE QUALITY FOR PM&S AND SPM. ALL NUMBERS ARE IN PERCENTAGES

	PM&S			SPM		
	SINGLE	DOUBLE	CLUSTERED	SINGLE	DOUBLE	CLUSTERED
f01_200	61	19	20	68	28	3
f04_200	70	17	12	67	30	3
IG0H	55	21	24	62	35	3
J00O	65	17	18	67	30	3
N03F	57	21	22	55	40	5
N03H	38	18	44	59	35	6
N03M	48	24	29	56	37	6
N046	37	21	42	55	39	6
N04D	47	22	30	58	36	6
N04H	45	23	32	57	38	5
N057	49	22	29	59	36	5
S012	56	21	23	63	34	3
AVERAGE	52	21	27	61	35	4

TABLE V
EXECUTION TIME (IN SECONDS) AND PEAK MEMORY USAGE (IN MEGABYTES) FOR THE LOSSLESS AND LOSSY PM&S AND SPM SYSTEMS

	LOSSLESS CODING		LOSSY CODING	
	PM&S	SPM	PM&S	SPM
RUNNING TIME (SEC)	40.29	125.31	55.06	118.55
MEMORY USAGE (MB)	17.1	17.8	17.3	17.6

text edges more continuous and smooth. The superiority of the lossy SPM system comes at the cost of encoding time and system complexity. The complexity gap between PM&S and SPM is even bigger than in the lossless case because lossy PM&S does not need the residual coder, while lossy SPM has to perform the three preprocessing steps on the whole image.

3) System Complexity: As shown in Figs. 1 and 2, PM&S and SPM share very similar structures. Building blocks such as symbol extraction, location and other numerical data coding, and cleanup page coding are basically the same. The big difference is that PM&S does not do refinement bitmap coding. For lossless coding, the PM&S system needs the residual coder at the very end. For lossy coding, the SPM system needs to preprocess the input image. In real applications, when using PM&S for lossy coding, extreme care should be taken in selecting a proper pattern matching criterion so that the system suffers minimal character substitution. Usually this means a more sophisticated pattern matching technique that takes longer to compute [9]. However, in our experiments, we use the same Hamming distance matching criterion for both systems. We set the mismatch threshold to be low for the PM&S encoder to suffer rare substitution errors.

Our experiments are done on a Pentium Pro 200 MHz, running Red Hat Linux 6.0, with 64 MB physical memory. Such a system is far from state-of-the-art, however, we can still obtain a valid relative comparison between the PM&S and SPM systems. Our code was not optimized for speed or memory efficiency.

We use execution time (Unix "time") and peak memory usage ("top") to measure the system complexity. Table V shows the execution time (in seconds) and peak memory consumption (in megabytes) of the two systems, in lossless and lossy modes, averaged over the 12 images. The timing results vary slightly each time the program is run; therefore we run each test image three times and take the average.

Singleton exclusion SPM is about three times slower than PM&S in lossless coding and two times slower in lossy coding. PM&S encodes only direct bitmaps, which, in our experiments, account for an average of only 12% of all extracted symbols; lossless SPM has to encode all extracted symbols and lossy SPM has to encode most of them (about 75% of all the symbols, see Table VI). Also, PM&S takes less time to match the symbols with the dictionary because it retains a smaller dictionary with only direct symbols; SPM adds every symbol into the dictionary, making it grow much faster.

PM&S needs about 2–4% less physical memory than SPM. Both systems need a page buffer, which, depending on the image size, is about 8 MB in our tests. Both systems also need the same number of arithmetic coders and the same coding strip buffer used for text region segments. Storing the dictionary only takes up a small percentage of the total memory; therefore, a smaller dictionary in PM&S only leads to marginal savings.

Finally, in Table VI, we list the percentages of perfect symbols generated in lossy SPM. This is a measure of how much encoding time is saved by not coding the bitmaps of the perfect symbols. While the percentage is image-dependent and varies a great deal (from 3% to 57%), we see on average 25% of the symbols are perfect. Omitting the arithmetic coding procedure for one quarter of all the symbols saves a nontrivial amount of encoding time. Note that we made no attempt to optimize our encoder toward generating more perfect symbols. Each symbol refers to its closest match in the dictionary, regardless of whether this reference will make it a perfect symbol or not.

B. Different Symbol Dictionaries for SPM-Based JBIG2

In this section we first investigate the effects on coding efficiency of using different symbol dictionaries for SPM-based JBIG2. We then change the dictionary sizes to obtain the dictionary size versus compression tradeoff curves.

1) Lossless and Lossy Compression Ratios: Table VII compares the total number of bytes required to encode the test images using the four symbol dictionary designs discussed in Section III: the one-pass, singleton exclusion, class-based and tree-based dictionaries. The percentages of improvement over the simplest one-pass design are shown for each of the other three

TABLE VI
PERCENTAGE OF TOTAL PERFECT SYMBOLS GENERATED BY LOSSY SPM SYSTEM

f01	f04	IG0H	J00O	N03F	N03H	N03M	N046	N04D	N04H	N057	S012	AVE.
47	57	25	46	5	24	13	3	16	15	17	33	25

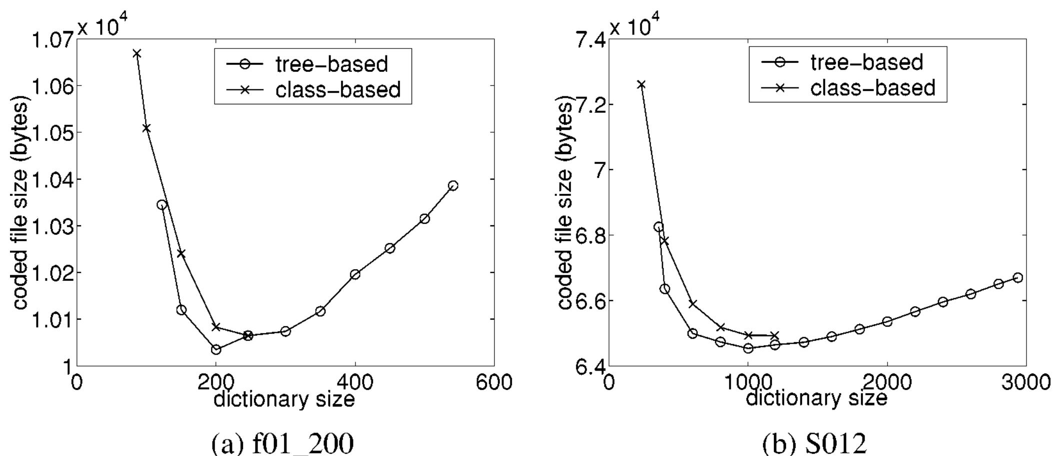

Fig. 10. Dictionary size versus compression tradeoff curves on two test images.

TABLE VII
TOTAL COMPRESSED FILE SIZES FOR LOSSLESS AND LOSSY SPM SYSTEMS WITH THE FOUR DICTIONARIES

	ONE-PASS	SINGL. EXCL.		CLASS-BASED		TREE-BASED	
		BYTES	IMP.	BYTES	IMP.	BYTES	IMP.
LOSSLESS	479,714	457,710	4.6	442,185	7.8	440,197	8.2
LOSSY	303,648	271,466	10.6	250,322	17.6	255,421	15.9

TABLE VIII
AVERAGE DICTIONARY SIZES AND REFINEMENT COMPRESSION RATIOS (RCRS) FOR THE THREE DICTIONARIES

	SINGL. EXCL.		CLASS-BASED		TREE-BASED	
	SIZE	RCR	SIZE	RCR	SIZE	RCR
LOSSLESS	1365	5.65	644	5.62	644	5.64
LOSSY	1080	12.18	536	13.00	529	12.87

designs. The one-pass dictionary gives the poorest compression. By adopting the singleton exclusion dictionary, on average 5% and 11% of improvement can be achieved for lossless and lossy compression, respectively. More sophisticated dictionary designs lead to greater improvements. In the lossless case, the class-based and the tree-based dictionaries both achieve around 8% of improvement on average. In the lossy case, the class-based dictionary is the most efficient, achieving 18% of improvement; the tree-based design is not as efficient as the class-based design but still very comparable, improving the compression by 16% on average.

Table VIII compares the sizes and corresponding refinement compression ratios (RCRs) obtained from the three symbol dictionaries. RCR is defined as the number of bits used to store the raw bitmaps of all refinement symbols divided by the number of bits spent on refinement coding these bitmaps. The one-pass dictionary is no longer listed for comparison because of its inferior performance. In lossless coding, the class-based and tree-based dictionaries achieve very close RCRs (5.62 and 5.64 compared to 5.65) with only half as many symbols as in the singleton exclusion dictionary (644 compared to 1365). For lossy coding we have similar results. Therefore, these two dictionaries are providing about the same amount of useful reference information even though they are much smaller. This explains the compression improvements obtained in Table VII.

2) Bit Rate Tradeoff Curves: Fig. 10 shows overall bit rate as a function of dictionary size for two test images, f01_200 and S012. Image f01_200 contains only 1088 symbols which is the least among all images, while image S012 contains 5653 symbols, the most among all. Starting with big dictionaries, we follow the procedures described in Section III-D to shrink their sizes. For the class-based design, we start from the initial classes and try to merge the most similar ones subject to the mismatch threshold. For the tree-based design, the biggest size corresponds to the initial MSTs. We relocate tree nodes till we get down to the smallest possible size for which all the tree branches still have weights below the mismatch threshold. The images are compressed using these dictionaries of different sizes. We plot the size versus bit rate curves in Fig. 10. The tree-based curves have rather broad flat floors. For both images tested, the overall compression stays nearly constant ($\leq$0.5% off the lowest values) over about 50% of the entire size ranges. So the compression is not very sensitive to the particular dictionary size chosen as long as it is within a reasonable range. However, the compression can be hurt by 4–6% if the dictionary is too big or too small. The class-based curves sit above the tree-based ones and cover narrower size ranges. They also shoot up faster at the left end, leading to 6–12% rate loss when the dictionary is too small. Both designs achieve the best compression at a size very close to the initial size of the class-based design.

V. CONCLUSION

In this paper, we investigate compression of text images using the JBIG2 standard. First we compare the PM&S-based and SPM-based JBIG2 systems, in terms of their coding efficiency, reconstructed image quality in lossy compression, and system complexity. The SPM system achieves on average 8% more efficient compression in the lossless case. In the lossy case, at

comparable bit rates, the SPM system also has a better control over the reconstructed image quality, making clustered errors very unlikely to occur. However, the SPM advantages come at the cost of higher system complexity: 2–4% higher memory consumption and two to three times longer encoding time. We also examine the bit rate tradeoff problem associated with SPM symbol dictionary design. We propose two new dictionary design techniques: the class-based and tree-based design. We test the proposed techniques on a set of 12 text images and find them to outperform the simplest one-pass dictionary formation by an average of 8% in lossless compression and 16–18% in lossy compression. We also propose methods to change the class- and tree-based dictionary sizes, and plot the bit rate as a function of dictionary size. Over a very broad size range (50% of the entire size range), the overall compression is not very sensitive to dictionary size, but it can be hurt by 4–6% using symbol dictionaries which are too small or too big. Finally, the tradeoff curves for the two dictionaries show that the initial class-based dictionary size falls into the optimal dictionary size range. For future work we will look at extending the proposed dictionary design methods into multi-page text image coding systems.

ACKNOWLEDGMENT

The authors would like to thank Dr. R. Arps, Prof. H. H. Koh, and D. Schilling for their helpful discussions.

REFERENCES

[1] *JBIG2 Final Draft International Standard*, ISO/IEC JTC1/SC29/WG1 N1545, Dec. 1999.

[2] P. Howard, F. Kossentini, B. Martins, S. Forchhammer, and W. Rucklidge, "The emerging JBIG2 standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8, pp. 838–848, Nov. 1998.

[3] R. B. Arps and T. K. Truong, "Comparison of international standards for lossless still image compression," *Proc. IEEE*, vol. 82, pp. 889–899, June 1994.

[4] R. Hunter and A. H. Robinson, "International digital facsimile coding standards," *Proc. IEEE*, vol. 68, pp. 854–867, July 1980.

[5] D. Tompkins and F. Kossentini, "A fast segmentation algorithm for bi-level image compression using JBIG2," in *Proc. 1999 IEEE Int. Conf. Image Processing*, Kobe, Japan, Oct. 1999, pp. 224–228.

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

[7] R. N. Ascher and G. Nagy, "Means for achieving a high degree of compaction on scan-digitized printed text," *IEEE Trans. Comput.*, vol. C-23, pp. 1174–1179, Nov. 1974.

[8] P. Howard, "Lossless and lossy compression of text images by soft pattern matching," in *Proc. 1996 IEEE Data Compression Conf.*, J. A. Storer and M. Cohn, Eds., Snowbird, UT, March 1996, pp. 210–219.

[9] S. Inglis and I. H. Witten, "Compression-based template matching," in *Proc. 1994 IEEE Data Compression Conf.*, J. A. Storer and M. Cohn, Eds., Snowbird, UT, March 1994, pp. 106–115.

[10] Q. Zhang and J. M. Danskin, "Bitmap reconstruction for document image compression," *Proc. SPIE*, vol. 2916, pp. 188–199, Nov. 1996.

[11] Q. Zhang, J. M. Danskin, and N. E. Young, "A codebook generation algorithm for document image compression," in *Proc. 1997 IEEE Data Compression Conf.*, Snowbird, UT, Mar. 1997, pp. 300–309.

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

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

[14] Y. Ye, D. Schilling, P. Cosman, and H. H. Koh, "Symbol dictionary design for the JBIG2 standard," in *Proc. 2000 IEEE Data Compression Conf.*, Snowbird, UT, Mar. 2000, pp. 33–42.

[15] Y. Ye and P. Cosman, "JBIG2 symbol dictionary design based on minimum spanning trees," in *Proc. 1st Int. Conf. Image Graphics (ICIG)*, Tianjin, China, Aug. 2000, pp. 54–57.

[16] R. Gould, *Graph Theory*. Redwood City, CA: Benjamin Cummings, 1988, ch. 3, pp. 68–72.

[17] E. S. Askilsrud, R. M. Haralick, and I. T. Phillips, "A quick guide to UW English document image database I, v. 1.0.," Intelligent Syst. Lab., Univ. Washington, CD-ROM, August 1993.

[18] C. Constantinescu and R. Arps, "Fast residue coding for lossless textual image compression," in *Proc. 1997 IEEE Data Compression Conf.*, Snowbird, UT, Mar. 1997, pp. 397–406.

Yan Ye (S'99) received the B.S. and M.S. degrees in electrical engineering from the University of Science and Technology of China, Hefei, in 1994 and 1997, respectively. She is currently pursuing the Ph.D. degree in electrical engineering at the University of California at San Diego, La Jolla, where she is a Graduate Student Researcher in the Information Coding Laboratory.

Her research interests include data compression and digital image processing.

Pamela Cosman (S'88–M'93–SM'00) received the B.S. degree with honors in electrical engineering from the California Institute of Technology, Pasadena, in 1987, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1989 and 1993, respectively.

She was an NSF Postdoctoral Fellow with Stanford University and a Visiting Professor with the University of Minnesota, Minneapolis, from 1993 to 1995. Since July of 1995, she has been on the faculty of the Department of Electrical and Computer Engineering, University of California at San Diego (UCSD), La Jolla, where she is currently an Associate Professor. Her research interests are in the areas of data compression and image processing.

Dr. Cosman is the recipient of the ECE Departmental Graduate Teaching Award, UCSD (1996), a Career Award from the National Science Foundation (1996–1999), and a Powell Faculty Fellowship (1997–1998). She is an Associate Editor of the IEEE COMMUNICATIONS LETTERS, and was a Guest Editor of the June 2000 special issue on "Error-resilient image and video coding" of the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS. She was the Technical Program Chair of the 1998 Information Theory Workshop, San Diego, CA. She is a member of Tau Beta Pi and Sigma Xi.