DCC'96 Impressions
Data Compression Conference

March 31 - April 3, 1996 • Snowbird, Utah

The most common and recurring theme: lossless compression of text and images. Talks on lossy compression of images were mostly devoted to quantization and especially encoding of wavelet coefficients, barely even mentioning the wavelet decomposition itself.

1. Weighted Finite Automata (WFA) compression

Although WFA (and fractal-type image compression in general) were definitely *not* the most popular topic of the conference, there have been three presentations (counting my poster) on the subject. Furthermore, *quite* a few people are interested in WFA, want to find out more about it, and even keep reading papers in this area. I saw this interest for myself when talking to those who stopped by at my poster. The WFA compression performance (which is better than that of the traditional wavelet coding, and on par with the most advanced techniques as zerotrees and spatial-oriented trees) is clearly the most appealing factor. WFA is much faster than a traditional (IFS) Fractal image compression. Finally, WFA brings in new, fresh and exciting ideas into somewhat stale area of image compression.

Karel Culik II and V. Valenta introduced new Generalized Finite Automata (GFA) in their paper *Finite Automata based compression of bi-level images* (pp. 280-289 of *Proc*.). It was with great clarity that the paper got across the main idea of using FA in (bi-level) image compression: A bi-level image can be completely described by a set of its black pixels. One can always attach (string) labels to pixels: For example (as used in Culik's GFA) one can consider a complete quadtree (with leaves being pixels), and associate labels `0`, `1`, `2`, and `3` with the four children of every non-terminal node of the quadtree. The path from the root to a leaf (representing a pixel) then reads as the pixel's label. Thus, a bi-level image can be uniquely described by a set of strings (black pixels' labels), or, in other words, by a language with the alphabet `{0,1,2,3}`. Since the language is finite, it is regular, and can be generated/recognized by a finite automaton. The obvious automaton contains as many states as there are strings in the language; this automaton however hardly leads to any compression. We hope though that the language (the set of black pixels) possesses some degree of regularity, which can be captured by an automaton with relatively few states. The idea of representing an image by a language is not new (see refs. [1,2,4,14] in Culik's paper); still Culik was the first to come up with an algorithm for building and compactly storing of an automaton that really leads to a good compression.

Building an FA representing an image is similar to a quadtree image segmentation. At the beginning, we consider an entire image as a state, and mark it as "to be considered". At each iteration step, we pick up a state that is still "to be considered", partition it into four quadrants and check out to see if they can be expressed as a (reduced-resolution) version of existing states. If some extant FA state does indeed look similar to our quadrant, we add an arc from our "being considered" state to that state, and mark the arc with 0,1,2, or 3 depending on the quadrant. If a quadrant of the being-considered state doesn't look like any existing state, we add the quadrant to the set of FA states, mark it as "to be considered" and connect it with a marked arc (as above).

Thus the FA compression algorithm can be expressed as follows (as adapted from the algorithm and discussion in the paper):

Algorithm 1.0 (automaton construction)
Data structures:
  Set of states $S = \varnothing$
  Set of arcs `(from-state,to-state,label,transformation)` $A = \varnothing$
  Stack of to-be-considered-states $U = \varnothing$

Initialization:
  I (the original image) $\rightarrow S$, $I \rightarrow U$

Iteration:
  if U **is** $\varnothing$, **stop**
  $c \leftarrow U$
  for-each quadrant q_i of c (i=0,1,2,3) **do**
    if q_i **is** white, **continue** the loop
    try to find such s in S and p that *transform*(s,p)$==q_i$
    if s is found
      $(c,s,i,p) \rightarrow A$
    else
      $q_i \rightarrow S$, $q_i \rightarrow U$
      $(c,q_i,i,\varnothing) \rightarrow A$
    end-if
  end-for

Here $x \rightarrow Y$ stands for adding (pushing) of x into Y, while $x \leftarrow Y$ means taking (popping) x from Y.

When the algorithm terminates, A represents the completed finite automaton. It is this set that has to be stored/communicated. The automaton is obviously a deterministic one (that is, each state has at most 4 arcs fanning from it, each being marked with a different label). The paper claims that this automaton is minimal. The paper states that while there may exist an equivalent non-deterministic automaton with fewer states, it won't be noticeably smaller than the deterministic one, as far as bi-level images are concerned. Note, a recursive inference algorithm for grayscale images described in [8,9] builds a non-deterministic WFA.

In the simplest formulation, function *transform*(s,p) takes no parameters (that is, $p \equiv \varnothing$) and simply squeezes s to the size of q_i (if necessary). In a more advanced formulation (GFA), the function, besides squeezing, can perform 90-degree rotations, flips and complementations.

As an elementary example, let's consider how algorithm would compress a 2x2 chess-board, Fig. 1:

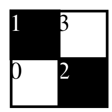

Fig. 1. 2x2 chess-board to compress, with labeled quadrants.

At the first iteration, the whole image is the only state that exists, and is being considered. The image is split into 4 quadrants (labelled as indicated on Fig. 1). Quadrants #0 and #3 are completely white, and are not considered any more. Quadrant #1 obviously doesn't look like Fig. 1. So, it is added as a new state, and marked as to be considered. The automaton built so far is

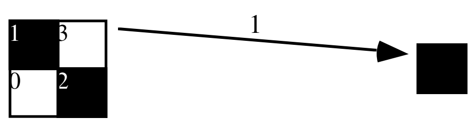

Fig. 2. Automaton at the first iteration, after quadrant #1 was considered

When considering quadrant #2, we notice that it looks exactly as the state we just added. Therefore, we simply make an arc to this state. The first iteration is thus finished, yielding the following automaton:

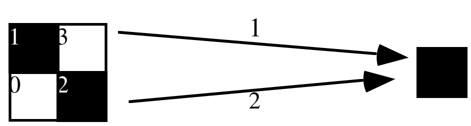

Fig. 3. Automaton after the first iteration.

The stack U of to-be-considered states contains now a single state, corresponding to the black square on the right-hand side of Fig. 3. Again, we split this square into four quadrants. Its quadrant #0 looks exactly the same as the square itself: both are completely black. Differences in size/resolution between the image and its quadrant are disregarded by the algorithm. Thus, we add an arc from the state to itself, and label it. The other quadrants are handled identically. Thus, the second iteration gives us

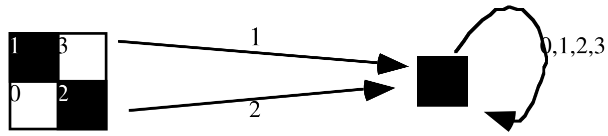

Fig. 4. Automaton after the second (final) iteration.

which is our final result, as none of the states left to be considered (stack U is empty now). Set A of arcs can be stored into a file. Of course, states themselves can be represented merely by their labels. Also note, when the algorithm creates a new state, the corresponding image is a block (square) of the original picture. Therefore, rather than storing this image block separately, we can merely point to a corresponding tile in the original image.

Let's see now how the compressed image can be restored from its FA representation. While the compression algorithm tries to find an automaton given an image (that is, given a language of black pixels), the decompression algorithm must find a language a given automaton recognizes/generates. Since the algorithm 1.0 above constructs a FA as a "language machine", we can put this language-recognizing machine in reverse, to be a "language generator": start from the final state, and follow the arcs in the opposite direction, until we arrive at the initial state. The image associated with this state would be the original picture. However, FA at Fig. 4 does not have any final state. Nor does it contain any mentioning of the original picture's dimensions. The latter is a feature, however: it lets one reconstruct an image at any resolution. Lack of the final state is no problem either: one can always use forward iterations, starting from an arbitrary (not all-white, though) image. We will show now how this can be done.

Suppose we want to restore the compressed chessboard at a resolution of 8x8 pixels. We will start with an arbitrary image (Fig. 5) and plug it into the automaton, Fig. 4:

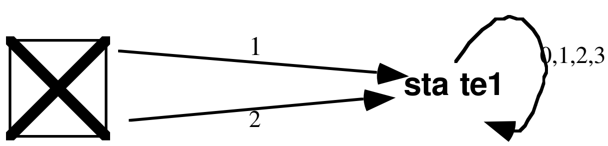

Fig. 5. 0th reconstruction iteration

We will split the image into 4 quadrants, and follow the arcs. Note, quadrants #0 and #3 don't have any corresponding arcs. By our assumption, that means these quadrants are purely white. Thus we obtain

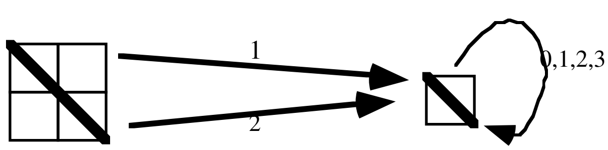

Fig. 6. 1st reconstruction iteration

The other state (the right-hand side of Fig. 6) becomes current now. We split the corresponding image into quadrants. The arrows say each of the four quadrants is identical to the whole image (up to the resolution, of course). Thus we take the whole state image (the right-hand side of Fig. 6), scale it down (to match the size of the quadrants), and arrange in its own quadrants. Also note, because the image on the right-hand side of Fig. 6 is nothing but a "reference" to two quadrants of the image on the left-hand site, changing the former image changes the latter two. Thus, after following the circular arc once, we get

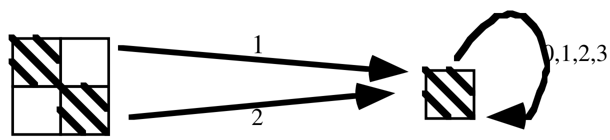

Fig. 7. 2nd reconstruction iteration

Obviously, a few more iterations would make the image on the right completely black, and the image on the left look like a 2x2 chessboard. Thus, although the automaton created by the algorithm 1.0 does not have final states, the iterative reconstruction process would converge after at most `log(N)` steps, `N` being a dimension of the image. Indeed, following an arc implies taking a quadrant of state's image; as a result, the state's image becomes more detailed. However, there is an obvious limit to the extent one can add details to an image: the number of pixels.

It's instructive to perform the similar restoration iterations on more complex examples, for instance, the ones on Figs. 3-6 in Culik's paper in the Proceedings. They all work, but it would take too much space to write out the examples here. Note that some arcs on Figs. 5 and 6 are in error.

Although the construction/reconstruction process outlined above has certain grace and appeal, it seems that resorting to final states makes a more flexible (and faster convergent) algorithm. For example, let's assume that the initial set of states contains two *a priori* states: pure white and pure black. These states are always final: reaching them means that the corresponding quadrant (from which the arc came) is filled with black/white pixels. Thus the modified construction algorithm would read

Algorithm 1.1 (construction of FA with final states)
Data structures:
Set of states `S = {white-state, black-state}`
Set of arcs `(from-state, to-state, label, transformation)` `A` $= \emptyset$
Stack of to-be-considered-states `U` $= \emptyset$

Initialization:
`I` (the original image) $\rightarrow$ `S`, `I` $\rightarrow$ `U`

Iteration:

if U **is** $\varnothing$, **stop**

$c \leftarrow U$

for-each quadrant q_i of c (i=0,1,2,3) **do**

  if q_i **is** white

    $(c, \text{white-state}, i, \varnothing) \rightarrow A$

  else if q_i **is** black

    $(c, \text{black-state}, i, \varnothing) \rightarrow A$

  else

    try to find such s in S and p that *transform*$(s,p) \approx q_i$

    if s is found

      $(c, s, i, p) \rightarrow A$

    else

      $q_i \rightarrow S$, $q_i \rightarrow U$

      $(c, q_i, i, \varnothing) \rightarrow A$

    end-if

  end-if

end-for

Applying this algorithm to the 2x2 chess-board is not very instructive:

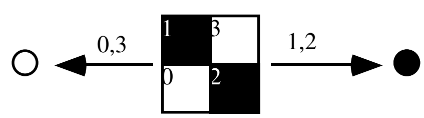

Fig. 8. FA with final states for the 2x2 chess-board

Still, the new algorithm works almost the same way as the old one, Algorithm 1.0. Note that equality check after *transform*(s,p) has been changed to an approximate equality. This means that we are content if a match between blocks (q_i and state s) is good enough. As a measure of quality of the match between two blocks, Culik uses a percentage of pixels where they differ.

As a practical matter, Culik and Valenta found out that it makes sense to use other final states besides pure black and pure white. For example: some commonly occurring black-and-white patterns. It also turns out that dealing with states representing small image blocks in the regular way is not very efficient. Therefore, they created a 256-entry codebook of 8x8 tiles, and used them as final states. Thus if the size of a quadrant q_i under consideration is 8x8, it is forced to point to one of the patterns in the codebook. Also, to speed up the search, it makes sense to tag FA states with a signature (say, the number of the black pixels). In searching for matches, we need to only consider states which signatures are close to that of our block. This is similar to block-classification techniques commonly used in the IFS fractal image compression (see below).

The GFA compression is relatively fast: it takes typically 2-5 secs on Pentium for a 512x512 bitmap. In the worst possible case (when compressing an almost random image) it takes 50 seconds. Comparing to the regular IFS image compression, the new Culik's method is blazingly fast.

The decompression algorithm in presence of final states is similar to the one sketched above. Here's it in the formal notation

Algorithm 2.0 (restoring an image from the FA by finalizing states)
Data structures:

Set of state labels `Sl` = load from the file

Array of images associated with the states `Si` [state-label] =
load-from-file **if** state-label corresponds to a final-state**,**
∅ otherwise

Array of iteration counters: `Cnt` [state-label] [quadrant-label] = `0`

Set of arcs `(from-state,to-state,label,transformation)` `A` =
load from the file

Stack of to-be-considered-states `U` = ∅

Initialization:

`I` (the label of the initial state) → `U`

`Si[I]` = arbitrary image of target's dimensions

Iteration:

if `U` **is** ∅, **stop**

`c` = **peek** from `U`

for-each quadrant q_i of `c` (i=0,1,2,3) **do**

  if q_i **is** marked finished, **continue** the loop

  find `s`,`p` such that an arc (`c`,`s`,`i`,`p`) is in `A`

  if `s` **is** a final state

    fill in q_i with `Si[s]` (using transformation `p`, and expanding
      `Si[s]` if necessary to cover the whole q_i)

    mark q_i as finished

  else if `Si[s]` **is** ∅

    `Si[s]` = q_i

    `s` → `U`

  else if `Cnt[c][i]` **>=** $\log(\mathit{dimension}(q_i))$

    mark q_i as finished

  else

```
            Cnt[c][i] += 1
            fill in qi with Si[s] (using transformation p, and expanding
                Si[s] if necessary to cover the whole qi)
            if s == c
               s → under the top of U
            else
               s → U
            end-if
        end-if
    end-for
    if all qi are marked finished, remove all mentioning of c from U
```

To see how this actually works, we will trace the algorithm using a part of an FA from Fig. 6 of the Culik's paper:

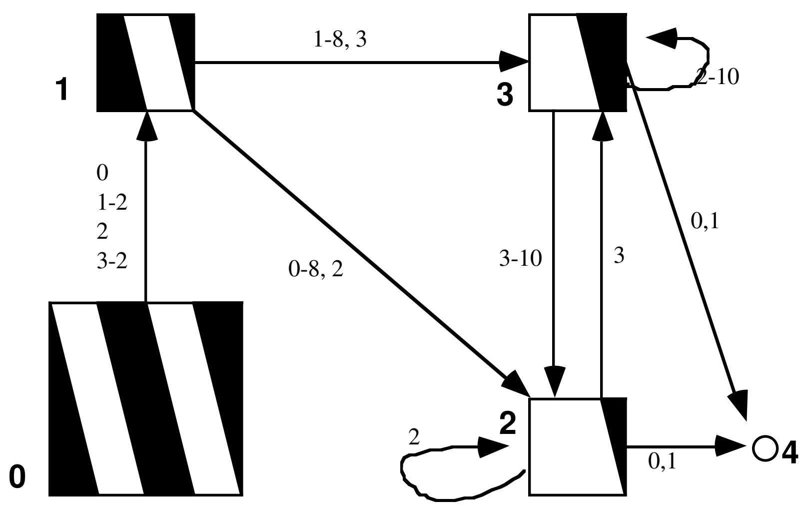

Fig. 9. FA for a diagonal striped image

Note the diagram as shown in *Proc*. has a few typos; Fig. 9 above fixes them. This is a generalized FA, the transform parameters (when non-empty) are specified by their labels after the dash. The complete list of the transformations and their labels is given on Fig. 4 of the Culik's paper. As a matter of fact, label `2` refers to a reflection relative to the center, `8` to complementation, and `10` is a combination of these. The numbers by the states in bold are the states' labels.

The complete tracing of the reconstruction algorithm would take too much space. We have to limit ourselves with a snapshot of the data structures taken somewhat half way through the iterations. The starting image was a criss-crossed one (Fig. 5).

Stack U: **0 1 1 1 1 2 3 2 3 3 2 2 3 3 2 2 3**
Set of state labels `Sl` = {**0,1,2,3,4**}
Final states: **4**
`Cnt`[0] = {**0,1,1,1**}, `Cnt`[1] = {**0,0,1,1**},
`Cnt`[2] = {**f,f,2,2**}, `Cnt`[3] = {**f,f,2,2**}
Array of images associated with the states `Si[`**0**:**3**`]` =

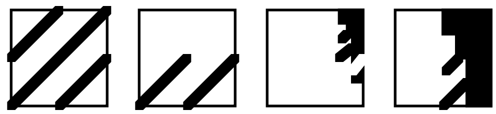

Fig. 10. A snapshot of the reconstruction algorithm

Images corresponding to states **2** and **3** are clearly converging (to those on Fig. 9). Once the corresponding iteration counters reach the threshold, the states become final, immediately leading to the reconstruction of state **1**, and then state **0**, the original image (Fig. 9).

The GFA bi-level image compression can be applied to color images as well, to separate color planes of these images, to be precise. It makes sense to compress color planes "together", that is, to build an automaton for one color plane, and then augment it to get it to represent the other planes.

The other presentation on WFA image compression was by Ullrich Hafner from U. of Würzburg, *Refining image compression with weighted finite automata* (pp. 359-368 of *Proc.*). The most enticing part of the talk was a color picture compressed at 600:1 (contrasted with a 300:1 JPEG compression). The JPEG result looks very blocky, while WFA one is rather smooth and appealing. Unfortunately, the method described in the paper is little more than a mere brute force, a simple and straightforward generalization of the regular (IFS) fractal image compression algorithm, with a rather superficial relation to WFA. Like the IFS fractal image compression, the refined method seeks correspondence (mapping) between range and domain blocks; both sets of blocks being partitionings of the same image. In the IFS method, the mappings are affine transformations of one domain block onto one range block; most commonly, it's merely a scaling down of a domain block (by 50%) followed by a linear adjustment of block's brightness (color). In the refined method, a range block is approximated by a linear combination of at most 8 domain blocks. The domain blocks are chosen from a pool of domain blocks, and can be of arbitrary size: if necessary, a domain block is scaled down and/or clipped to the size of the range block. The domain pool initially contains some 300 "basis" images (like `f(x,y)=1`, `f(x,y)=x+y`, including some DCT basis functions). The algorithm starts with a whole picture considered as a single range block. First, it finds the best approximation of the range block within the available domain pool. The algorithm then splits the range block into two halves, and tries to find (recursively) their best approximations. If the subdivision gives a better representation of the original range block, this block as well as its two halves are added to the domain pool; otherwise backtracking is performed. In any case, parameters of the found best approximation are recorded (and subsequently stored using various elaborate techniques, for example, a quasi-arithmetic coding of the bintree partitioning and matrices participating in linear combinations, with a specially tuned model).

However limited a one-range-block - one-domain-block (affine) IFS mapping may look like, it has its advantages. It is obvious that a domain block and the range block it is mapped into must have similar "features" (as edges, variance, etc). That makes it possible to limit the number of blocks one has to consider for a mapping by classifying domain/range blocks according to some criteria (e.g., total/vertical/horizontal variance). Only blocks of similar kinds have to be considered for possible matches: see, for example, *Accelerating Fractal Image Compression by Multi-dimensional Nearest Neighbor Search*, presented at DCC'95 (p. 222 of *DCC'95 Proc*.). It is much more difficult to use this type of search acceleration in Hafner's method. He has little choice but consider all possible doubles, triples (up to octuples) of domain blocks (and their clipped/scaled versions) in search for the best linear combination. He mentioned in a conversation that a typical low-quality compression requires a pool of up to 2000 domain blocks. It takes some 10-15 minutes on a DEC AlphaServer 2100 Model 4/200 (190 MHz, 512 MB RAM) to perform the compression. A high-quality setting may take *hours*. On the whole, the refined WFA method is a brute force fairly straightforward generalization of the IFS fractal compression, with little elegance.

In a conversation with Dr. Culik and Ullrich Hafner by my poster, it became clear that WFA are more powerful than IFS. For example, WFA can represent a range block as a (linear) combination of several domain blocks. Ullrich Hafner has mentioned that there are other proofs of representing IFS by WFA (in addition to mine). However, the general case WFA cannot be reduced to IFS (it's kind of obvious from my paper, too). Note that my fat-pixel interpretation of WFA is quite capable of representing linear combinations of domain blocks. *It'll be good to show how* ***exactly*** *my projection matrices map to a WFA state diagram*. Note, it seems also possible to deal with GFA in my matrix notation: one has to consider "multiplications by a scalar" as some kind of "shorthand" to specify flips, translations, and other GFA block operations.

Dr. Culik said that a new edition of Barnsley's *Fractals Everywhere* book contains now a more general definition of IFS (which is still called IFS).

Dr. Culik said that the ‘Iterated Systems’ has bought his WFA implementation for compressing grayscale/color images, and is going to file a patent on his behalf. He will only get his name on the patent. The patent doesn’t cover GFA (for bi-level images).

Some thoughts as to the further development of WFA:

- reducing blockiness artifacts showing up occasionally on reconstructed images (at low-quality compression settings): one can perform one or two steps of a 2D wavelet decomposition of the picture and then proceed with the WFA compression of the baseband (treating it like a wavelet-downsampled image). As to the highbands, they can be either completely disregarded, or compressed with WFA as well (at least, high-low and low-high bands), starting with the automaton already constructed for the baseband. Throwing away the highbands means “smooth” upsampling of the picture, which could improve its appearance. One can be more selective in disregarding the highbands information, and keep “significant” parts of the bands. The significance can be estimated from the segmentation of the baseband (as given by the WFA), in a manner similar to that described in paper *Quadtree-guided wavelet image coding* (see below);

- there may be other ways of labelling pixels, that is, mapping a set of pixels to a set of strings. For example, one can consider zigzag or serpentine labeling;

- one can deduce a grammar for a language of black pixels representing an image, and encode the grammar similar to that for a structured text (see Craig Nevill-Manning’s paper elsewhere in this review). As a matter of fact, the mere construction of a GFA follows rather closely an algorithm for grammar deduction described in that paper.

2. Lossless coding of text

There was an interesting talk, *The Entropy of English using PPM-based model* (pp. 53-62 of *Proc.*) on the “real” entropy of English and if modern text compression methods come close. The real entropy of English is generally considered to be around 1.5 bit per character (bpc): Shannon estimated it as 1.3 - 0.6 bpc (having real people guess the next letter of some text, which is being given them letter-by-letter), modern cryptographical n-gram analysis gives 1.5 bpc. The best text compression scheme (PPMC) encodes an English text at 2.4 bpc.

The paper shows that after some tweaking, PPM-based methods can compress English text (e.g., King James Bible) up to 1.46 bpc. The tweaks:

- using only 26 letters of the alphabet and a space, thus disregarding upper/lower case-ness and non-letters. Shannon used the same approximation. This alone gives 10% improvement;

- training a model. Normally encoding starts with an empty model, which gets "filled in" and adjusted as the compression progresses. The paper proposes to take thus adapted model after the end of encoding of some training text, and use the model to compress some other text. If a PPM model is trained on a few chapters of a book (or a few works of some author) and then used to compress other chapters of the same book (other books of the same author), the compression improves by nearly 47%. Even if the model is trained on an unrelated text (e.g., the Brown corpus), the improvement is still significant. The only requirement is that the amount of the training text should be large (at least 10^6 chars);

- replacing frequent digrams (like `th`, `er`, `gh`, etc). by a single symbol;

- context of a PPM model is fixed at 5.

The conclusion: PPM-based methods are actually quite good and rather close to the lower bound.

A very interesting paper *Compressing Semi-Structured text using hierarchical phrase identification* by Craig Nevill-Manning, Ian Witten, and Dan Olsen (pp. 63-72 of the *Proc*.) describes a text compressor sequitur and its applications (to compress a genealogical database of the LDS Church).

Sequitur "discovers" a context-free grammar for a text, and encodes the text as a set of grammar productions plus the "residuals" (including seed symbols for the grammar). Sequitur actually performs quite well as a general text compressor: it gives 2.64 bpc on the Calgary corpus, which is almost as good as PPM-based methods, and slightly better than gzip.

It's interesting that a grammar-based compression is capable of handling situations where a symbol (word, etc.) being predicted does not immediately follow its predictor. This is quite common when a structured text is mixed with a "free" text, as in forms.

I talked to Craig Nevill-Manning (after his presentation, and on a few other occasions). He said that grammar productions (that is, rules, like `B->abA`, `A->cd`) are encoded "inplace". That is, a non-terminal symbol is encoded as a backward pointer to a string defining it. For example, a text string

```
cdabcdklcdfabcd
```

is compressed to `ABklAfB`, and transmitted as cd ab ptr-back-to(cd) (which is A) kl ptr-back-to(cd) f ptr-back-to(cdA) (which is B)

This is very similar to a regular LZ-type compression. In fact, Sequitur is an LZ-compressor, with some modifications as to arranging a dictionary and keeping strings in the dictionary for a very long time. Many LZ methods (e.g., LZ78) add new phrases to the dictionary by growing old phrases by one symbol. Sequitur adds new phrases as complex combinations of old strings (non-terminals) and symbols (terminals). This leads to smaller and richer dictionaries, which means that entry indices and "backpointers" are shorter. Also, Sequitur stands out in interpretation of the LZ dictionary, and applying it to learning.

I mentioned to Craig "productions with exceptions/probabilities", to efficiently encode a text which is very regular, but has a few "typos". He agreed that making "exceptions" to grammar productions may make sense, but he hasn't tried it.

There were some other presentations on modifying LZ compression. For example, a "context sorting" (pp. 160-169), which is actually a straightforward implementation of LZ, with an index into a sorted context playing the role of a backward pointer. This method performed just like gzip, on average. However, because of sorting, it is very slow (even in decoding).

A very clear (and clearly presented) paper *On the implementation of minimum-redundancy prefix codes* by A. Moffat and A. Turpin (pp. 170-179 of *Proc.*) is devoted to designing of an optimal coder (especially, a very fast decoder) for compressing a source with a humongous alphabet (of the order of 10^6 symbols). The method is to be used for a text retrieval system employing a word-based compression. Note that this compression technique is static: probabilities (weights) of symbols in the alphabet are estimated *a priori* and *not* adjusted during encoding. The method is essentially a Huffman optimal prefix codec, with a modification: given an ordered list of symbol probabilities, the Huffman algorithm calculates codeword lengths (rather than codes themselves). This can be done "inplace" in $O(n)$ time. Once the codewords' lengths are figured out, the code itself is easy to make: take a binary counter and truncate it to the necessary size (see p. 173 for details). The method is very fast indeed: it decodes 100 Mbytes/min.

I talked to some guy at GATech (who uses compression for communications). He was experimenting with LZ77 algorithms, specifically, with techniques for pruning the dictionary once it's filled up (and still a new entry has to be made). Regularly, some kind of LRU (least recently used) algorithm is employed. He tried to pick a victim "at random". That is, once the dictionary is filled up, scan it from the very beginning and find the first leaf (that hasn't grown since its addition to the dictionary) and recycle that entry. If another entry needs to be added, keep scanning the dictionary for leafs (from the position of the previous victim). He found out that although this method is simpler than LRU, it outperforms LRU most of the time.

3. Lossless or near-lossless image compression

The routine for lossless image compression – linear prediction of the next pixel(s), probability estimation of the prediction errors, followed by an entropy coding (mostly arithmetic coding) – has been established a while ago, and hasn't noticeably changed. The pixel predictors have quite evolved though, they can detect and predict edges (and other common features of the image).

The most characteristic example: LOCO-I, developed at HP Labs, which is destined to become the next standard for lossless image compression: *LOCO-I: A low complexity, context based, lossless image compression algorithm* (pp. 140-149 of the *Proc*.). The predictor is simple, but can handle vertical and diagonal edges. The real sophistication of the method is in estimating distribution parameters of prediction errors. These parameters are later used to design the optimal Huffman or Coulomb-Rice coder. As table 1 on p. 148 shows, LOCO-I achieves (on average) 4 bpp (that is, 2:1 compression) in lossless compression of "standard pictures", thus outperforming JBIG by a sizable margin.

The next paper, *An Algorithmic Study on Lossless Image compression* by Xiaolin Wu (pp. 150-159) describes more advanced linear predictors, which could detect and predict sharp/smooth/weak horizontal and vertical edges.

Compression of (bi-level) images of text has some specifics: segmenting picture into connected components of black pixels (marks), linear-prediction (in coding marks), "soft-matching" marks, and recording mark's position relative to the previous marks(s).

There has been an invited talk on (near) lossless image codec standards currently under development by ISO/IEC/JCT1 /Standards Committee 29/Working Group 1. The new standard would supplant a "lossless mode" of the JPEG standard. LOCO-I (see above) was picked up as the base-line algorithm for a low-complexity version of the standard. Arithmetic-coding-based techniques (currently under development) will be included as an extension. The current base-line technique (based on LOCO) outperforms lossless JPEG (with Huffman coding) by 30% on average, with a significantly higher improvement for compound documents. That's why the new standard is being developed. All contenders for a near lossless standard work similarly to the lossless algorithms; however, they do an additional step of quantization of prediction errors. They also use a slightly different prediction error distribution model. In the result, reconstructed image pixels are guaranteed to differ from the original picture ones by no more than a specified amount (generally ±1, ±5 and ±7 pixel values).

Note a paper *Lossy Compression of noisy cardiac image sequences* (p. 43), which handles medical (X-ray) imagery in a particular way: regions of the heart and the organs are not quantized in order to preserve the diagnostic information in these images. There were a few other papers where compression is married to (and facilitated by) a classification.

4. Wavelet coding

Although Wavelet decomposition was used in almost all lossy compression schemes presented at the conference, the wavelet decomposition procedure was barely even mentioned. That is, it has become a routine not worthy expounding on. The attention has clearly shifted from the wavelet decomposition per se (including wavelet filters design) to quantizing, and especially, efficient coding of the decomposition results.

The wavelet image compression has clearly reached saturation. That is, all "good" methods give very close results. A poster *Device Selective Quantization for Reversible Wavelets* (from RICOH California Research Center) showed some very typical examples of a (CREW system) wavelet compression of a 512x512 color picture with lots of details (there was a bike wheel with spokes). Even compressing the image at 160:1 gave quite good reconstructed picture, although defects (especially ringing) were quite noticeable and annoying.

A paper *Optimal Bit Allocation for Biorthogonal wavelet coding* (pp. 387-395) makes a case for applying different thresholding/ quantization to different bands of the wavelet transform, when bi-orthogonal wavelet filters are used. Since bi-orthogonal filters are not energy-conserving, quantization errors in different bands enter the total with some weight, which is proportional to $\mathrm{tr}(G^T G)$, where G is the reconstruction matrix.

Note a surprising (and surprisingly good) rebirth of the old quadtree image segmentation, described in *Quadtree-guided wavelet image coding* by Chia-Yuan Teng and David L. Neuhoff from U. Mich. (pp. 406-415 of *Proc.*). The standard segmentation algorithm subdivides a quadtree node if the corresponding image block (square) is not uniform enough. In a refined formulation, one attempts to predict pixels within an image block, using some linear predictor. The corresponding quadtree node is subdivided only if the prediction is not satisfactory. Surprisingly, the method gives rather good results [5]. In the present paper, the authors modify the method by first performing one or two stages of a wavelet decomposition (they do *not* build the whole wavelet pyramid). The predictive quadtree method is applied then to the low-low band (the base-band, or downsampled image). The authors assume that if some part of the base-band requires a fine subdivision, it contains plenty of detail. Therefore, the corresponding parts of the wavelet bands are expected to be significant, too. The rest of the wavelet bands is assumed insignificant (and simply zeroed out). The method's performance turns out better than even that of Said/Pearlman's [4], which is the best of the wavelet-based codecs. Also, in contrast to typical quadtree-segmentation schemes, the reconstructed image doesn't look blocky. See the table below for PSNR comparisons.

Note on quantization of wavelet coefficients: in the paper above, the authors increase the quantization step (and a quality threshold) by 30%, as they go down the tree. John Villasenor's paper on compression of seismic data used a uniformly-spaced scalar quantizer with steps chosen as a function of band's variance. The quantizer step centered at zero was wider by approximately 20% than the others (it improves the performance from a rate-distortion standpoint, he says).

The paper on the quadtree-guided wavelet decomposition suggests the following idea: use the combined magnitude of c^2, c^3, c^4 (maybe more) wavelet coefficients to predict significance of wavelet coefficients at the 0th and the 1st levels of the decomposition. Here c^k refers to a coefficient at level k, where k=0 stands for the bottom (the finest resolution) level. To be more precise,

$$\text{if} \quad \sum_{k=2,3,4} \sum_{l=all-bands} \left| c_{l;ij}^{k} \right| < threshold \ \text{then set} \ c_{l;ij}^{k} = 0, \quad k = 0,1$$

It makes sense to do that right before regular quantization/zerotree discovery steps. Note, this approach seems to be simpler (and more natural) than the hybrid quadtree-wavelet method above.

PSNR (dB) for a few compression schemes of a monochrome Lenna image (512x512)

Method	Bits per pixel (compression ratio)			
	1 bpp (8:1)	0.5 bpp (16:1)	0.25 bpp (32:1)	0.17 bpp (47:1)
JPEG	37.7	34.7	30.5	27.3

Plain VQ	32.5	30.5		
SVQ-DCT [6]	39.00	35.88	32.55	
LVQ-SUB [7]	39.97	35.61		
S/C-SUB(D) [8]	38.53	35.32	32.19	
Qtree + Wavelet [1]		36.5	33.6	32.1
Zerotrees [2]		34.0		
Emb zerotrees [3]		36.3	33.2	
spatial trees [4]		36.8	33.7	
Qtree pred [5]		35.6	32.6	31.0

[1] C.-Y.Teng and D.L.Neuhoff, *Quadtree-guided wavelet image coding*, Proc. DCC'96, pp. 406-415.

[2] A.S.Lewis and G.Knowles, *Image compression using the 2-D wavelet transform*, IEEE Trans. Image Proc., pp. 244-250, April 1992.

[3] J.M.Shapiro, *Embedded image coding using zerotrees of wavelet coefficients*, IEEE Trans. on Signal Processing, 41(12), 3445-3462.

[4] A. Said and W.A.Pearlman, I*mage compression using the spatial-orientation tree*, IEEE Int. Symp. on Circuits and Systems, pp. 279-282, May 1993.

[5] C.-Y.Teng and D.L.Neuhoff, *A new quadtree predictive image coder*, IEEE Int. Conf. on Image Proc., pp. II73-II76, Oct. 1995.

[6] N.Farvardin, F. Camurat, and R.Laroia, *An application of fixed-rate scalar-vector quantization in image coding*, Proc. 1994 Int. Conf. on Image Processing, vol I, pp. 598-602, Austin, TX, Nov. 1994

[7] Z. Mohdyusof and T.R.Fischer, *Subband image coding using a fixed-rate lattice vector quantizer*, Proc. 1995 IEEE Int. Conf. on Image Processing, to appear

[8] N.Tanabe and N.Farvardin, *Subband Image coding using entropy-coded quantization over noisy channels*, IEEE J. Sel. Areas Commun., vol. 10, pp.926-942, June 1992.

5. Low-bit rate video standards

An invited mid-day talk by John Villasenor from UCLA.

The current standard is H.261 (a part of H.320 suite, which besides video, deals with audio, packet structure, multiplexing, etc). Emerging replacement is H.263 (a part of H.324 suite), which performs 40-50% better.

There is a special standard AV32M for a wireless low-bit rate audio-visual communication.

H.263 differs from H.261 in:

- $^1/_2$ pixel motion-estimation (using interpolation)
- more resolution options
- can use 4 motion-vectors within a single macroblock
- has an option of using an arithmetic coding
- can encode two frames at the same time

AV32M is H.263 + G.273 (audio codec) + an "adaptation" layer that defines multiplexing of packets and error correction.

MPEG4 has many conceptual differences from the previous MPEGs: it moves away from video in terms of frames to video in terms of object planes (composition of moving objects), has special modes to code texture, shapes, and motion. Because MPEG4 is so innovative, it's slow in discussing/adoption. Chances are MPEG4 would never become a standard.

Midday talk: teaching Computer Science to kids

This was a talk by T.Bell and I.Witten previewing their new book. A bit controversial thesis: computer science is not a programming; teaching computer science can be done can be done very early, and without any computer at all (which has many advantages, especially for schools on a low budget).

Note a very cute public-key cryptography for kids, to encode a yes/no answer (parity). The method is based on graphs, deriving its cryptographical strength from the fact that it is very difficult to find a dominant set of a graph.