Lab #8: Source Coding

Goal: Using MATLAB, develop and evaluate a method for compactly encoding images of scanned text.

Instructions:

1. Complete the activities for Wednesday’s lab (see second section below).

2. Prepare a brief written description of your encoding algorithm (see outline in the third section below) and submit your write up at the start of Friday’s lecture.

In the lab (2pm – 5pm, Wed., April 11, 2007)

Start by going through the usual steps to initialize the 6.082 environment, copy over the Lab 8 files from /mit/6.082/Labs/Lab8, and start up MATLAB.

During this week’s lab your task is to develop and evaluate an algorithm for compactly encoding images of scanned text – this is the type of encoding used by fax machines to transmit page images from the sending machine to the receiving machine. For the post-lab activity we’ll ask you to write up a brief description of your algorithm and summarize its performance when encoding the test image.

Obviously we could transmit a black-and-white image pixel-by-pixel using one bit to encode each pixel. Our goal is to develop a lossless encoding that transmits fewer bits but still enables the original image to be reconstructed by the receiver.

Step 1: examine a test image of scanned text

Fire up MATLAB, change to the lab 8 directory and load up the test image along with some other useful pre-computed information:

\[
\text{load lab8.mat}
\]

The black-and-white test image is available in a 500-by-500 array of pixels called \textit{lorem_img}. Note that in MATLAB pixels are encoded by their brightness, so black has a pixel value of 0, and white a pixel value of 1. You can examine the test image using MATLAB’s image manipulation tools:
imshow(lorem_img)

In the window that pops up, click on the magnification tool in the toolbar (it’s the magnifying glass with the “+”) and then click on the image itself to zoom in a couple of times. Think about transmitting the image row-by-row: each row would consist of alternating runs of white pixels and black pixels. What’s your sense of the distribution of run lengths? Does it differ between white runs and black runs? Perhaps we can compress the image by using run-length encoding where we send the lengths of the alternating white and black runs instead of sending the pixel pattern directly.

For example, consider the following MATLAB representation of a 4x7 bit image:

```
1 1 0 0 1 1 1
1 1 0 0 1 1 1
1 1 1 0 0 1 1
1 1 1 1 0 0 1
1 1 1 1 1 1 1
```

which can represented as a sequence of run lengths: [2 2 6 6 2 8]. If the receiver knows that runs alternate between white and black (with the first run being white) and that the width of the image is 7, it can easily reconstruct the original bit pattern. It’s not clear that it would take fewer bits to transmit the lengths than the original image – that’ll depend on how clever we are when we encode the lengths! If all run lengths are equally probable then a fixed-length encoding for the lengths (e.g., using 8 bits to transmit lengths between 0 and 255) is the best we can do. But if some run length values are more probable than others, we can use a variable-length Huffman code to send the sequence of run lengths using fewer bits.

We’ve done some pre-processing of the image for you and left the results in various MATLAB variables initialized by the `load` command you gave above:

```
runs
```

a vector constructed by processing the image left-to-right, top-to-bottom and counting the length of each run of pixels, alternating between runs of white pixels and runs of black pixels (just like the example above). To ensure this has been done correctly, we built a utility that reconstructs an image from a vector of its run lengths; try the MATLAB command `imshow(runs2img(runs,500))` which expands the run lengths into alternating sequences of white and black pixels and then formats the result as a rectangular array of width 500, showing the result.

```
h_runs, h_white, h_black
```

These 32-element vectors are histograms of the run lengths. The first element is the count of the runs of length 1, the second element is the count of runs of lengths 2, etc. The 32nd element is the count of runs with a length of 32 or greater. h_runs
is the histogram for all the runs, \texttt{h\_white} is the histogram for runs of white pixels and \texttt{h\_black} is the histogram for runs of black pixels.

**Step 2:** examine distribution of run lengths

It’s useful to look at the distribution of run lengths – if some run lengths are much more probable than others then we can try using a variable-length Huffman code to encode run lengths. That means we’ll use just a few bits to encode common run lengths, hopefully resulting in an encoding that uses fewer bits overall.

Look at the contents of the three histogram vectors (\texttt{h\_runs}, \texttt{h\_white}, \texttt{h\_black}). Note that the distributions of white and black run lengths are noticeably different, so we’ll end up with shorter encodings if we use a different encoding scheme for white run lengths vs. black run lengths.

**Step 3:** create a Huffman code for black run lengths

Using probability information from the \texttt{h\_black} histogram, create a variable-length Huffman code to be used when encoding black run lengths. For reasons explained in Step 4 below, you should include an encoding for a run length of 0; assume its use will be infrequent (i.e., that it has a very small count in the histogram). So your Huffman code should provide encodings for run lengths from 0 to 10 (the range of possible run lengths from test image).

Let’s find out if using your Huffman code will result in a shorter encoding. First, let’s see what size encoding we’d get if we just encoded the run lengths in binary. From the histogram we see that black run lengths vary between 1 (very common) to 10 (very rare). Using a fixed-length encoding, we could choose 4 bits to encode the length of each run.

What is the expected number of bits from encoding a run length using your Huffman code? You can compute this by multiplying the length of the encoding for each possible run length times the probability of a run of that length, and summing the results. Is your expected number of bits for encoding a run noticeably smaller than 4 bits? [Hint: it should be!]

**Step 4:** create a Huffman code for white run lengths

You can repeat the procedure of step 3 for white run lengths, remembering to include an encoding for runs of length 0. There’s an extra complication however: Huffman encoding works well for the high probability short runs. But we do have a noticeable number of long white runs, some very long indeed. It would be incredibly tedious to construct an encoding for runs of every possible length. So you should create a modified Huffman code which includes an extra “escape code” that switches over to a separate fixed-length encoding for runs longer than some upper limit. In this example, the fixed-length encoding could be the length of the run.
expressed as an 8-bit binary integer (least significant bit first). For example, if we wanted to encode a run length of, say, 169, we’d get “eee…eee10010101” where “eee…eee” is the binary encoding for your escape code.

Even using the escape code, we can’t encode white runs longer than 255. A simple “trick” is to encode the longest run you can, followed by a 0-length run of black, and then repeat this process until the long white run has been encoded. The same technique will work in step 3 for runs of black longer than 10. This is why we included a code for 0-length runs in both our encodings.

Looking at the histogram of white run lengths, I’d recommend switching to an escaped fixed-length encoding for run lengths bigger than 10 or 11. Count up how many run lengths exceed your chosen upper limit and use the result when computing the probability for the escape code during the construction of your Huffman code.

**Step 5.** Use your Huffman codes to encode the test image

Create a MATLAB script that encodes the test image by using your Huffman codes to encode each of the run lengths in the `runs` vector, alternating between encoding white runs and black runs. A sample script that implements a simple Huffman code can be found in `lab8encode.m`—you can modify it to use your Huffman codes. When the script completes, it will print out the total length in bits of the encoded image.

**Step 6.** Write a decoder and use it to display the results of Step 5

Create a MATLAB script that decodes the encoding you created in Step 5, i.e., that converts your binary encoding into a list of integer run lengths. Display the resulting image by using the MATLAB command `imshow(runs2img(decode,500))` where `decode` is the vector that holds the decoded run lengths. `lab8decode.m` is a sample script that decodes and displays the results produced by `lab8encode.m`—you can modify it to use your Huffman codes.

This completes the lab. Don’t forget to work on your lab write up using the outline given below.

---

**Post-lab (due at the start of lecture, 2pm, Fri., April 13, 2007)**

Please write up a brief description of your encoding algorithm using the following outline:

1. Show your Huffman tree for encoding black run lengths, including the probability information you calculated from the histograms. Using your Huffman code, what is the expected length in bits of encoding a black run length? Explain how you
calculated this result. What sort of improvement in encoding length do we get by using the Huffman code vs. a fixed-length code?

2. Show your Huffman tree for encoding white run lengths, including the probability information you calculated from the histograms. Describe at what point you switch to a fixed-length encoding, and how that fixed-length encoding works. Using your Huffman code, what is the expected length in bits of encoding a white run length (including an estimate for the fixed length codes used for long run lengths)? Explain how you calculated this result.

3. If we send the original 500x500 image pixel-by-pixel it will take 250,000 bits. How many bits does it take using your encoding scheme? What compression ratio did you achieve?

4. Attach a printout of your both your encoding and decoding script.

Turn in your write up at the beginning of lecture on Friday.