CSC310 – Assignment #2

Due: Oct.20, 2005, 9am at the **START** of tutorial
Worth: 8%
Late assignments not accepted.

1 Typical Sequences

- For a source which emits sequences of symbols from the alphabet $\{a_1, a_2, a_3\}$ independently with probabilities $p(a_1) = 0.05$, $p(a_2) = 0.05$ and $p(a_3) = 0.9$, describe the typical sequences in the N^{th} extension of the source.
- Consider a source which emits sequences of symbols from the alphabet $\{a_1, a_2\}$ independently with probabilities $p(a_1) = 0.005$ and $p(a_2) = 0.995$.
- Imagine we use a fixed-length code (all codewords have the same length) to encode blocks of 100 consecutive symbols (i.e. the $100^t h$ extension of the source). We do this by defining all the "typical blocks" of 100 symbols to be *those with 3 or fewer a_1's.* We assign each typical block a codeword, and ignore all other ("atypical") blocks.
- Keeping in mind that all the codewords must be of the same length, find the minimum length required to provide the above typical set of blocks with distinct codewords.
- Calculate the probability of getting an "atypical" block, i.e. one that we ignored when designing the code.

2 Lossless Compression

- In this question you will write computer programs to losslessly compress and re-expand files based on Huffman and Arithmetic coding. You can use whatever programming language you like for this assignment, and you do not have to hand in your code. However, as with many of your other programming courses, you are expected to do all of your coding by yourself; sharing code with other students or copying parts of programs you did not write or do not understand is not permitted and will be considered cheating.
- Your programs will read in arbitrary files one byte at a time and thus our source alphabet will be the set $\{0, 1, \ldots, 255\}$. As the probabilities of these symbols, you will use the relative frequencies of the symbols in the file you are encoding, in other words the counts of the number of times each byte occurs divided by the total number of bytes in the file. (Notice that this means many symbol probabilities will be zero; make sure your programs deal correctly with this.)
- Of course, your decoder will also need to know the symbol probabilities. In a real world system these probabilities would themselves have be encoded (somehow) at the beginning of the compressed file. But for this assignment, you can "cheat" and just write the symbol counts to a separate file (different than your encoding file) which the decoder reads before it starts decoding the compressed (encoded) file.
- Our encoding alphabet will be binary bits $\{0, 1\}$. To keep things simple, when you read/write your encoded files you can represent each bit as either the character '0' or '1', rather than using single bits directly. This means your encoded file will appear 8 times larger than it should be, but you can just divide by 8 when computing encoding lengths, etc. Of course, if you want you can read/write bits directly, but don't waste too much time hacking this up.

- Here's what you need to code up:
  1. Write a Huffman coder which encodes a single symbol (byte) at a time, using the frequency counts in the original file as the probabilities. You will have to build the Huffman tree, figure out the optimal code based on that, and then use that code to compress sequences of bytes into sequences of bits and decompress the other way.
  2. Write a Huffman coder which encodes pairs of symbols (two bytes) at a time, i.e. for the second extension of your source.
  3. Write an Arithmetic coder which encodes and decodes streams of symbols into streams of bits. To keep things simple, you can use double precision floating point representations of the interval boundaries in your code; this will be sufficient given that the arithmetic coding algorithm re-expands the interval whenever it can.
- Here are the experiments you need to run and the results you need to report:
  1. Using your Huffman coder, your extended Huffman coder and your Arithmetic coder, compress the files `file1, file2, file3, file4` from the course website. Decompress the encoded versions of these files and verify that the decoded result exactly matches the original file.
  2. Report the number of bits needed to encode each file (not counting the length of the separate file you used to store the symbol counts).
  3. Compute the average number of bits per symbol and the compression ratio you achieve for each file.
  4. Compute the entropy of each source model (ie the entropy of the set of probabilities you estimated by counting), and state how much worse than the entropy each encoder is on each file.
  5. Report the first, second, last and second-to-last bit for each of the encodings using each encoder.
  6. Create a file which is a single line containing your name in all capital letters followed by your student number, with no spaces or punctuation anywhere and no linefeed or carriage return at the end. (In other words, your file should contain only the symbols A-Z and 0-9.) On your assignment type or print this line, as well as the complete encoding of it using all three programs. It is cheating to ask anyone else to produce this encoding for you.

2.1 Arithmetic Coding – Encoder Pseudocode

- Input symbol probabilities $p_1, p_2, \ldots, p_I$.
- Compute the cumulative probabilities $P[j] = \sum_{i=1}^{j} p_i$ with $P[0] = 0$ and $P[I] = 1$.
- Initialize the interval boundaries u=0,v=1 and the extra bit count c=0.
- Read in the source symbols $s_1, s_2, \ldots, s_N$ in order.
  For each symbol s_n:

  $r = v - u$ // set the interval width
  $v = u + rP[s_n]$ //set the new interval by subdiving
  $u = u + rP[s_n - 1]$ //the existing interval according to this symbol

  while($v \leq 0.5$ or $u \geq 0.5$ or $u \geq 0.25$ and $v \leq 0.75$) {

  if($v \leq 0.5$) { output "0" followed by c "1"; set $c = 0$
  set $u = 2u$; set $v = 2v$ } //expand bottom half of unit interval
  else if($u \geq 0.5$) { output "1" followed by c "0"; set $c = 0$;
  set $u = 2u - 1$; set $v = 2v - 1$ } //expand top half of unit interval
  else if($u \geq 0.25$ and $v \leq 0.75$) { set $c = c + 1$
  set $u = 2u - 0.5$; set $v = 2v - 0.5$; } //expand around middle of unit interval
  }
- Output a few more bits to finish off the encoding:

  if($u < 0.25$) output "0" followed by (c+1) "1"
  else output "1" followed by (c+1) "0"

2.2 Arithmetic Coding – Decoder Pseudocode

- Input symbol probabilities $p_1, p_2, \ldots, p_I$.
- Compute the cumulative probabilities $P[j] = \sum_{i=1}^{j} p_i$ with $P[0] = 0$ and $P[I] = 1$.
- Initialize the interval boundaries u=0,v=1 and the current message $t = 0$.
- Set $m = \text{ceil}[-\log_2(\min_i p_i/4)]$.
- Read in the first m encoding bits $z_1, z_2, \ldots, z_m$.
  Set t to the binary number $0.z_1 z_2 \ldots z_m$, i.e. $t = \sum_{f=1}^{m} z_f 2^{-f}$.
- While there are remaining encoding bits:

```
r = v − u                  // set the interval width
w = (t − l)/r              // find normalized position of t within interval
find i such that P[i − 1] ≤ w < P[i]
output "s_i"
v = u + rP[s_n]            //set the new interval by subdiving
u = u + rP[s_n − 1]        //the existing interval according to this symbol


while(v ≤ 0.5 or u ≥ 0.5 or u ≥ 0.25 and v ≤ 0.75) {

    if(v ≤ 0.5) set u = 2u; set v = 2v                          //expand bottom half of unit interval
    else if(u ≥ 0.5) set u = 2u − 1; set v = 2v − 1; set t = t − 0.5          //expand top half
    else if(u ≥ 0.25 and v ≤ 0.75) set u = 2u − 0.5; set v = 2v − 0.5; set t = t − 0.25     //middle

    t = 2t; if(t ≥ 1) t = t − 1;

    Read in the next encoding bit z_n.
    If z_n is "1", set t = t + 2^(−m).
}
```