Project 2 Description
Huffman Coding
FCM 709

March 30, 2014

1 Introduction

In this project you will define your own format for doing lossless compression on an image file. The two main ingredients to accomplishing this are Huffman coding and the xpm file format.

The book *Algorithms in C* by Sedgewick explains Huffman coding in chapter 22, and gives an implementation for compressing text files. You should be able to adapt the code given there for the purposes of this project, though you will need to make some changes.

I have made a short video to complement the explanation of Huffman coding provided by Sedgewick which can be viewed at `http://tinyurl.com/lkmt57k`. Another video on the xpm format is at the url `http://tinyurl.com/me9t2pn`.

1.1 Huffman Codes

The basic idea of Huffman coding is to compress a message, thought of as a stream of symbols, by assigning binary numbers called *codewords* to the symbols. This assignment is done in such a way that the following are true.

1. Symbols which occur frequently in the stream have short codewords.
2. Symbols which occur infrequently in the stream have long codewords.
3. No codeword (as a binary number) is an initial part of any other codeword.

As an example, consider the phrase

`BABBLING␣BABOON`

The letters occuring in this phrase, together with their frequencies are shown in Table 1.

Huffman coding assigns to the symbols the binary codes seen in Table 2.

A	2
B	5
G	1
I	1
L	1
N	2
O	2
space	1

Table 1: The letters and their frequencies

A	11
B	01
G	0010
I	0000
L	0001
N	100
O	101
space	0011

Table 2: The codewords assigned to the symbols.

A quick look at these tables will verify that the encoding behaves as outlined above. That is, frequently occurring letters have short codes, infrequently occurring letters have longer codes, and no code is an initial part of any other. The last property is usually described by saying that the code is *prefix-free*. The actual encoding is then done simply by replacing letters with their codewords. The result in this case is:

```
01110101000100001000010001101110110110100
```

Note that it is the property of being prefix-free that makes this code unambiguous, since breaks between codewords are not written.

It happens that Huffman codes are optimal in a certain technical sense beyond the scope of the project (see Wikipedia). Roughly speaking, no general binary encoding method produces shorter encodings than Huffman encoding, with minor caveats.

The Sedgewick book gives the exact workings of Huffman coding. The algorithm involves the use of a priority queue and a binary tree (technically a *binary trie*) which is the reason this fits well with the lessons from the 2nd half of the semester.

1.2 XPM format

The acronym xpm stands for X Windows Pixel Map format. It is a kind of raster graphics format that was developed for displaying icons in Linux.

The reason we will work with xpm in this project, as opposed to some other image format are the following.

1. The xpm format is very simple.

2. We can read xpm files with a text editor rather than a hex editor.

3. The format offers no compression, and so we can get real gains by compressing xpm files.

A detailed explanation of xpm is given in the video linked above and the materials referenced below, but let's take a look at a small image (Figure 1) in xpm format (Figure 2). You will see that the xpm format is actually in C syntax and defines an image as an array of strings. The first string

```
"30 30 2 1"
```

says that the image is 30 pixels wide, 30 pixels long, has two colors, and, in the xpm file, one character is used to represent one color. There then follows, on lines 4 and 5 a mapping of the space character to white, and the period character . to black. The colors are expressed as RGB values given in a hexadecimal format. Lines 6 to 35 give the actual bitmap. The lines are each `(num chars per color)*width` characters long (in this case $1 \times 30 = 30$) and there are `height` lines (in this case also 30).

Figure 1: A simple image.

Image files in the xpm format can be read and manipulated using the GIMP image viewer, which is freely available on all platforms. GIMP can also be used to export images to xpm format.

More information about xpm can be found on the corresponding Wikipedia page and in the xpm manual, which is linked from Wikipedia.

2 The project

The goal of this project is to compress xpf files using Huffman coding. For simplicity, we will only compress the bitmap part of the xpm file, which is the part of the file that comes after the color mapping. Rather than compressing characters, you should instead compress colors (or pixels). This involves parsing the bitmap part of the xpm file into n character groupings, where n characters represent a color as explained in the xpm literature. The frequencies of the groupings (which together make a single "symbol") should then be computed, and Huffman coding should be performed in the usual way.

For instance, suppose an xpm image uses two characters to represent a color, and has the mappings

$\$f \mapsto \#00FF00$ (green)
$b. \mapsto \#FFFF00$ (yellow)

so that `$f` stands for a green pixel and `b.` stands for a yellow pixel. Then if the image is 5 pixels wide, a typical line in the bitmap part of the xpm file will look like the following.

```
"$f$fb.b.$f",
```

Then we would say that the frequency of the symbol `$f` on this line is 3 and the frequency of `b.` on this line is 2. The frequency of the colors over the entirety of the bitmap portion of the xpm file is what should be used for Huffman coding, and in this case it is the character *pairs* that should be encoded.

The resulting encoding should be written to a file. Rather than writing `0` and `1` characters in ASCII (which would not afford much of a space savings) it will be better to do binary output so that actual bit values can be written.[1] There are special functions (`fwrite` and `fread`) which are designed for IO with blocks of binary data. You don't have to use them, but you can. Here is some sample syntax.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     FILE* fp = fopen("hey.bin","wb");
6     char a = 'a';
7     fwrite(&a,1,1,fp);
8 }
```

You should see the literature online regarding the `fwrite` command to completely understand how it works. Note that on line 5 of the code listing, the output format is "wb" rather than "w" indicating that writing will be done in a binary format. To read binary files it might be helpful to view them in a hex editor, such as `bless` on Linux. See Figure 3 for a screenshot of the output of the above program.

The screenshot in the figure shows that the file contains one byte whose hexadecimal value is 61, which is correct.

It should be interesting to see how much shorter (in bytes) your compressed file is than the original file. For the purposes of testing your work you may want to write some code that reads in your binary file and recovers the original bitmap.

If you would like to do something more ambitious, such as producing a compressed file that contains all of the information necessary for decompression, you are welcome to pursue that and I would strongly advise it. At the moment I am reluctant to make it a requirement because I haven't implemented the project. I will have more advice in a couple of weeks when I have my version finished.

[1] Actually, the smallest amount of memory that can be written to a file is 1 byte. Dealing with this will present a minor technical problem when outputting the Huffman code.

```
/* XPM */
static char * smile_xpm[] = {
"30 30 2 1",
"        c #FFFFFF",
".       c #000000",
"                              ",
"                              ",
"                              ",
"                              ",
"                              ",
"                              ",
"          ...                 ",
"        .....         ...     ",
"        .....       .....     ",
"        .....       .....     ",
"          ...         ....    ",
"                        ..    ",
"                              ",
"                              ",
"                              ",
"                              ",
"                              ",
"                        .     ",
"          .                ...",
"        ...              .... ",
"        ...            ....   ",
"          ...          ....   ",
"            ....   ......     ",
"              ..........      ",
"                .......       ",
"                  ...         ",
"                              ",
"                              ",
"                              ",
"                              ",
"                              ",
"                              "};
```

Figure 2: The smile icon in xpm format.

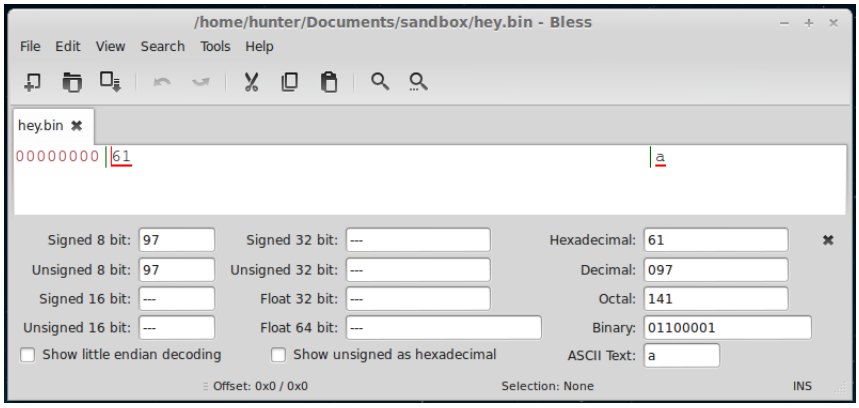

Figure 3: The output of the program listing shown in bless