Home
Syllabus
Lecture Notes
Assignments
Project 2

David Galles
Computer Science
Univerisity of San Francisco

Huffman Coding (Due Monday, March 23rd)

For your second project, you will write a program that compresses and uncompresses files using Huffman coding. To compress a file, your program will follow the following steps:

- Read in the entire input file, and calculate the frequencies of all characters.
- Build a Huffman tree for all characters that appear in the input file (characters that do not appear in the input file should ***not*** appear in your Huffman tree)
- Build a lookup table, which contains the codes for all characters in the input file
- Check to see if the compressed file would be smaller than the original file. If not, stop -- don't do any compression. Print out a message instead that the file cannot be compressed
- If the compressed file will be smaller, create the encoded file:
  - Print out a "Magic Number", which will be used to guard against uncompressing files that we didn't compress
  - Print out the Huffman tree to the output file
  - Use the lookup table to encode the file

To uncompress a file, your program will follow the following steps:

- Read in the "Magic Number", and make sure that it matches the number for the this program (exiting if it does ot match)
- Read in the Huffman tree from the input file
- Decode the input, using the Huffman tree

If your program is called with the ``verbose'' flag (-v), you will also need to print some debugging information to standard out. If your program is called with the ``force'' flag (-f), then the file will be compressed even if the compressed file would be larger than the original file.

File Compression

Reading input files

To read in the input files, you will use the provided TextFile class, which has the following methods:

- `TextFile(String filename, char readOrWrite)` The constructor takes two arguments -- the name of the text file, and a single character. To open a text file for reading, pass in the character 'r' or 'R'. To open a text file for writing, pass in the character 'w' or 'W'
- `public boolean EndOfFile()` The EndOfFile method can only be

called for input files. It returns true if the entire file has been read, and false otherwise.
- `public char readChar()` This method can only be called for input files. The next character (next 8 bits in the input file) is read and returned
- `public void writeChar(char c)` This method can only be called for output files. The character `c` is written to the output file,
- `public void rewind()` This method can only be called for input files. The file is rewound to the beginning (useful for encoding the file, after it has been read in to determine frequency information)
- `public void close()` Close the current file. Call the close method when you are done with the file. If the close method is not called, output files will not be written out correctly

Magic Numbers

You only want to try to uncompress files that you actually compressed yourself. To help ensure this, you will write a "Magic Number" to the first 16 bits of the output file. When uncompressing a file, first read in these 16 bits and make sure that they match the magic number. If not, your program should print out an error message and not try to decompress the file. The "Magic Number" that you should use is 0x4846 (that is, the ASCII characeters HF).

Building Huffman Trees

Huffman trees are built from the leaves up. See the visualizations for examples of building huffman trees. The class notes for this project also have a thorough description of building Huffman trees.

Building Huffman Tables

Once the Huffman tree has been built, we will need to use it to create the codes for each character. We can do this by doing a traversal of the tree, keeping track of the path from the root to the current node. When a leaf is reached, we store the code (that is, path from the root to that leaf) in out code table, at the index of the character stored at the leaf.

Checking File Sizes

Once you have built the tree table, you can compute the sizes of the compressed and uncompressed files.

- The size of the uncompressed file (in bits):
  - (# of characters in the input file) * 8
- Size of the compressed file (in bits)
  - Add up:
    - For each character c in the input file, (frequency of c) * size of the encoding for c
    - Size of the tree (1 bit for each internal node, 9 bits for each leaf

        - An extra 2 bytes (16 bits) for the magic number
        - An extra 4 bytes (32 bits) for header information used in the BinaryFile class
    - Note that the compressed file size needs to be a multiple of 8 bits -- so if the calculated binary file size is 457 bits, the file will actually be 464 bits long

If the compressed file is smaller than the original file (or the code was called with the -f option), go ahead with the compression. Otherwise, do not compress the file (instead, print out a message to standard out that the file was not compressed)

Printing Huffman Files

To assist in printing out compressed files, the class BinaryFile is provided, which has the following methods:

- `public BinaryFile(String filename, char readOrWrite)` The constructor takes two arguments -- the name of the text file, and a single character. To open a text file for reading, pass in the character 'r'. To open a text file for writing, pass in the character 'w'
- `public boolean EndOfFile()` The EndOfFile method can only be called for input files. It returns {\tt true} if the entire file has been read, and {\tt false} otherwise
- `public boolean readBit()` The readBit method can only be called for input files. A single bit is read from the input file.
- `public void writeBit(boolean bit)` The writeBit method can only be called for output files. A single bit is written to the output file.
- `public char readChar()` The readChar method can only becalled for input files. The next 8 bits are read from the input file, and returned as a character
- `public void writeChar(char c)` The writeChar method can only be called for output files. The character c is written to the output file, using 8 bits.
- `public void close()` Close the binary file. This method *must* be called after you are done with the file, or you will get strange behavior. {\em Especially} for output files.

To print a Huffman tree to the output file, we merely do a preorder traversal of the tree, printing out all of the nodes in the tree. We will need to encode which nodes are leaves, and which nodes are
interior nodes. We can do this by:

- Printing out a single bit with value 1 for each internal node.
- Printing out a single bit with value 0, followed by an 8-bit character value for each leaf.

The BinaryFile class has methods writeBit and writeChare to assist you. You may use some other method of your choice for serializing trees if you wish, but make sure that your method does not require more space!

Encoding File

Once the Huffman codes have been created, and the Huffman tree (and Magic Number) have been written to the output file, we only need to go through the input file again, character by character, writing out the appropriate code for each character. *Don't forget to close the output file when you are done!*

File Decompression

First, we need to make sure that the magic number matches. If it does, we can go head and do the decompression. If not, then we will print out a message to standard out and exit.

Reading Huffman Tree

To read in the Huffman tree, we do a preorder traversal of the tree -- guided by the input file -- creating nodes as we go.

Decoding File

Once the tree has been built, decoding files is easy. Start from the root of the tree, follow the appropriate child based on the next bit read in from the input file until a leaf is reached, and then print out the character stored at that leaf.

Command Line Arguments

Java allows the user to pass in command line arguments. The input parameter to the main function is an array of strings. If a Java main program has the prototype:

```
public static void main(String args[])
```

and the program is called with the command

```
% java MyProgram arg1 arg2 arg3
```

Then `args.length == 3, args[0] = "arg1", args[1] = "arg2",` and `args[2] = "arg3".`

Your program should expect to be called as follows:

```
% java Huffman (-c|-u) [-v] [-f]  infile outfile
```

where:

- `(-c|-u)` stands for either "-c" (for compress), or "-u"(for uncompress)
- `[-v]` stands for an optional "-v" flag (for verbose)
- `[-f]` stands for an optional "-f" flag, that forces compression even if

the compressed file will be larger than the original file
- `infile` is the input file
- `outfile` is the output file

The flags -f and -v can be in either order. So, the following would all be legal:

- `java Huffman -c test test.huff`
- `java Huffman -c -v myTestFile myCompressedFile`
- `java Huffman -c -f -v test test.huff`
- `java Huffman -u -f test1.huff test2`
- `java Huffman -u -f -v test1.huff test2`

Verbose Output

If a file is compressed with the "-v" option, you should print the following to standard output (using System.out.print(ln)):

- The frequency of each character in the input file (print the ASCII values of the characters, instead of the characters themselves, to make this more readable for binary files)
- The Huffman tree (see class notes on printing trees for pointers on how this can be done)
- The Huffman codes for each character that has a code (characters which do not appear in the input file will not have codes. Again, print the ASCII values of characters instead of the characters themselves)
- The size of the uncompressed file and the size of the compressed file

If a file is uncompressed with the "-v" option, you should print out following to standard output (using System.out.print(ln)):

- The Huffman tree (see class notes on printing trees for pointers on how this can be done)

Due Date

This project is due at Midnight on Monday, March 23rd. The project may be turned in after Monday, but by Wednesday March 25th at Midnight for 75% credit. Projects turned in after Midnight on March 25th will receive no credit.

Program Submission & Environment

You need to submit an electronic version of your code. To submit electronically, submit the file Huffman.java (as well as all other source files that your program needs to run, including the provided files for file I/O) to the subversion repository:

`https://www.cs.usfca.edu/svn/<username>/cs245/Project2`

You may develop your code in any environment that you like, but *it needs to run under linux in the labs!* While I recommend developing under linux, you may develop in Windows if you prefer, as long as your program runs under linux. To compile and run your program in linux, create a directory that contains all of the necessary .java files. Then compile all the files with the command

```
% javac *.java
```

You can then run you program with the command:

```
% java Huffman -c <input file> <output file>
```

Collaboration

It is OK for you to discuss solutions to this program with your classmates. However, no collaboration should *ever* involve looking at one of your classmate's source programs! It is usually extremely easy to determine that someone has copied a program, even when the individual doing the copying has changed identifier names and comments.

Supporing Files

- BinaryFile.java
- TextFile.java
- Assert.java (Used by BinaryFile and TextFile clases)
- Javadocs for Provided Classes