CS51 - Draft Specification

Team Members

1. Willy Xiao **2.** Kevin Eskici **3.** David Herman **4.** Eamon O'Brien

Brief Overview:

Our team will tackle compression. While different compression algorithms exist for specific data-types and file-types, we will implement compression algorithms that work for any type of file. Then, we will conduct a study to determine which type of algorithm works best for which kinds of files, ultimately culminating in a super-compressor and a directory-compressor.

Features List:

Step 1. Implement 4 distinct compression algorithms (April 21, CP1)

Algorithms:

See this for types of compression:
http://en.wikipedia.org/wiki/Category:Lossless_compression_algorithms

I. Fibonacci Coding (*Willy Xiao*)

Fibonacci coding functions mainly by shortening integers (which can be universalized to 4-byte inputs). This type of encoding should not require a large data-structure. In fact, it's very simple, here's pseudocode to encode an integer "N:"

1. Subtract the largest number in the fibonacci sequence from N
2. Place a bit 1 in the location of that fibonacci number's sequence
3. Repeat with N' until N' becomes 0
4. Append an extra 1 at the end of the sequence.

http://en.wikipedia.org/wiki/Fibonacci_code

*This should be a fairly quick implementation. Once done, if I have a lot of time, I will work on: Golomb-Coding/Golomb-Rice Coding.

II. Huffman (*Kevin Eskici)*

Huffman coding works by constructing a binary tree of bytes based on their representative frequencies in a file. The main idea is that the file can be compressed by

using less space for something that shows up a lot (for example the letter “e” in a text) and with the tradeoff of using more space for those things that are less common (i.e the letter “q”), so it is a type of entropy encoding. A huffman compressed file contains a header of each bytes respective frequencies so that the decompressor can rebuild the tree used to compress it and decode the message using it.

III. Sequitur (*David Herman)*

Functions by forming a hierarchical representation of a single string (however long). Terminal symbols (a, b, c...) are replaced by non-terminal grammatical symbols (numbers, etc.) that represent repeated sequences. The process continues recursively generating a hierarchical representation of the original string. A basic example:

a) abcababc -->
b) AcAAc -->
c) BAB --> where B = Ac and A = ab

http://www.cs.waikato.ac.nz/ml/publications/1997/NM-IHW-Compress97.pdf
http://arxiv.org/pdf/cs/9709102.pdf

Note: I’m pretty sure this is doable but not completely. I’ll start working over the next few days and figure out as soon as possible how complex it is.

IV. Lempel–Ziv–Markov chain algorithm (LZMA) (*Eamon O’Brien*)

LZMA makes use of a unique type of data compression known as dictionary compression. Dictionary compression compresses files by looking for similarities between the file's contents and a dictionary (data structure) that contains a number of strings. When a match is found, the string in the file is replaced with a reference to the string in the data structure. This modified version of the file is then encoded with a range encoder. A range encoder basically converts a file’s contents into one number.

http://www.7-zip.org/sdk.html

Range Encoding:
http://en.wikipedia.org/wiki/Range_encoding

Step 2. Conduct a study defining which algorithms work best for which data-structures; implement a function that takes in a file and returns the best algorithm (April 28, CP2)

Step 3. Implement a super-compressor (April 28, CP2)

Step 4. Implement compression for directories (April 28 - May 5, CP2 - Final)

Step 5. Implement more algorithms, incorporated into the study of Part 2 (May 5, Final)

Draft Technical Spec:

1. Modularization using python modules: http://docs.python.org/2/tutorial/modules.html

2. Signatures for each step in features list created using modules:

0. We will need a module that reads in bytes from a file and outputs bytes into a file.

Public:

reader.read : (file, file_location -> byte)

reader.write : (byte, file, file_location -> file)

Note: This can probably be found in the Python library. Essentially this means that before we start reading/writing, we should all agree to use the same module for it.

a. Compression Algorithm:

Public:

algorithm.compress : (file -> compressed of alg)

algorithm.decompress: (compressed of alg-> file)

Private:

[described in "algorithms" in features list]

b. Function to determine best algorithm for file type

Public:

compare.best_alg : (file -> algorithm)

Private:

Depending on the study, this will require different data structures to determine which algorithm is best for which data-type.

c. Super-Compressor/Decompressor

Public:

super.compress : (file -> compressed of super)

super.decompress : (compressed of super -> file)

Private:

Use: compare.best_alg

Use: algorithm modules in part (a).

super.which_alg : (compressed of super -> algorithm) -- this function will take in a compressed file from the super compressor and determine which algorithm was used to compress it.

d. Compression of directory

Public:

dir.compress: (directory tree -> compressed of directory)

dir.decompress: (compressed of directory -> directory tree)

Private:

Use: super module in part (c)

dir.dir_tree : (directory tree -> directory tree representation) -- this algorithm will take in a directory tree and somehow represent it in a file. Not sure how this will be done yet; it might just be done based on the order of the glued together list of files.

dir.glue : (compressed of super list -> compress of directory) -- this algorithm takes in a list of compressed files from the directory and then glues it together

Note: There should be no problems with this type of modularization. The compare module and the super module will be developed concurrently. This means there should be a prototype of compare.best_alg that just returns a random algorithm.

Python reference:
https://developers.google.com/edu/python/set-up