CS51 - Technical 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

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.

Pseudocode for Huffman:
Encode:
1. Get the relative frequencies of all the bytes in the file
2. Create a leaf node for each byte (would be character in a text file)
3. Take the two nodes with the lowest probabilities from the tree and replace them with a new node that has them as children and a probability equal to the sum of theirs.
4. Repeat step 3 until there is one node with probability 1
5. Assign bits to branches so that leaf nodes with the highest frequencies require the least number of bits to represent.
6. Go through the file replacing bytes with the corresponding sequence of bits from the graph (this will be done in a separate output file).
7. This “coded” version of the original file along with a header containing the corresponding frequencies of bytes make up the encoded file.

Decode:
1. Reconstruct the tree from encode using the frequencies in the header.
2. Go through the encoded file and when you reach a sequence of bits that leads to a leaf node, write the corresponding byte to an output file.
3. Repeat step 2 until you reach the end of the encoded file, and the resulting output is the decoded file.

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

Compression Pseudo-Code (Note: this is probably not the most efficient, but I want to get a working version before I try to increase efficiency):

1. Define dictionary with string keys and int values
2. Generate list of unique symbols
3. Read first two chars
4. If char pair is not in dictionary
    a) Enter pair as key
    b) Enter 1 (or Null symbol yet to be determined) as corresponding value
6. If pair is in dictionary, update value to be an unused symbol
7. Repeat steps 4-6 for whole string (except the next char pair includes the second char from the first char pair - moves one char at a time)
8. Read first two chars of original string
9. If corresponding dictionary value is 1, do nothing
10. If corresponding dictionary value is a symbol, replace pair with the symbol
11. Repeat steps 8-10 for whole string, with the following condition
    a) If char pair was left, move forward one (so the second char is repeated)
    b) If chars were replaced with a symbol, move to next char pair
12. Repeat steps 3-11 until there are no repetitions in the remaining string

Pseudo-Code to create decoding information:

1. Turn dictionary of symbols into tuple list sorted containing the keys and values
2. Iterate through list and create new dictionary from non-1 keys

Pseudo-Code to Decode (super inefficient way, but I'm not sure how to implement the more efficient way – I'll implement this first and then try to do the better way if I have time):

1. Read first char
2. If in dictionary, replace with corresponding value
3. Repeat steps 1-2 for rest of string
4. Repeat steps 1-3 until there are no symbols left in string

http://www.cs.waikato.ac.nz/ml/publications/1997/NM-IHW-Compress97.pdf
http://arxiv.org/pdf/cs/9709102.pdf
Note: Sequitur the way we have it only works on text files (a tf said this may end up being the case) so it is not included in our super compressor or directory compression.

IV. Lempel–Ziv–Markov chain algorithm (LZMA) (*Eamon O'Brien, Willy Xiao,*

Kevin Eskici)

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.

Compression Pseudo-Code

1. Specify the number of bits to be extended (i.e. 8-bits or 12-bits)
2. Read in the first character from the file to be compressed and store it in char
3. Repeat steps 4 through 7 for every remaining character in the file.
4. Read in the next character and store it in char2
5. If the string concatenation of char + char2 is in the Code Table, get the code. Otherwise, output the code for the concatenation of char + char2 and add it to the Code Table.
6. Store code in the Output file in the specified number of bits
7. char = char2
8. Output the last character char
9. Exit

Decompression Pseudo-Code

1. Read the first char (l)
2. Convert it to its original form via the Code Table
3. Output the converted code
4. Repeat steps 5 through 10 for every remaining code in the file
5. Read a char z
6. Convert l + z to its original form
7. Output in character form
8. If l + z is new then store in the Code Table
9. Add l + z first char of entry to the Code Table
10. l = first char of file
11. Exit

Reference: http://guideme.itgo.com/atozofc/ch60.pdf

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

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

Step 3. Implement a super-compressor

Step 4. Implement compression for directories

Step 5. Implement more algorithms, incorporated into the study of Part 2

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.

Instead of reader.read and reader.write we used wrapped all abstractions into a "helpers" module. Here is file i/o info :

```
helpers.start_compress(file_name, alg_name) ->
        (file_in, file_out)
helpers.end_compress(file_in, file_out) ->
        compressed_file

helpers.start_decompress(file_name, alg_name) ->
        (file_in, file_out)
helpers.end_decompress(file_in, file_out) ->
        decompressed file
```

These four handlers allowed python scripts to work on the intermediate “file_in” and “file_out” rather than on the original files. This allows us to write strings of 1’s and 0’s as ascii characters rather than bytes (some of which python does not allow to be written).

These functions also handle ensigning and unsigning the files with the algorithm name that was used to compress it and the original file extension.

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:

study.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.concat_add : (directory tree -> directory tree representation) -- this algorithm will take in a directory tree and somehow represent it in a file.

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

e. Tester

Public:

Takes in a file to test on and a directory to test on. Does unit tests on these and finally returns unit

test.test_all : (file_name, directory_name -> unit)

Private :

Each of these test the individual parts of the project given either a file_name or a directory_name to test :

test.test_algs : (file_name -> unit)
test.test_super : (directory_name -> unit)
test.test_dir : (directory_name -> unit)

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.

Version Control:

We set up a git repository (on code.seas) for version control and file sharing