Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- Why do we use compression?

1) To save space
   - Drives (HD or flash), despite increasing size, always seem to be filled with new programs and data files
     - Programs keep getting larger (and more complex)
     - New formats (ex: 4K video) require a lot of space

2) To save time/bandwidth
   - Most programs and files are now downloaded from the internet
   - Many people may be accessing at the same time
   - Compressed files allow for faster transfer times, and allow more people to use a server at the same time

- Major types of compression

  1) Lossy – some data is irretrievably lost during the compression process

     - Ex: AAC, MP3, JPEG, MPEG-2, MPEG-4, H.265
       - Good for audio and video applications, where the perception of the user is required
       - Gives extremely large amounts of compression, which is useful for large audio and video files
       - If the quality is degraded somewhat, user may not notice or may not care
       - Many sophisticated algorithms are used to determine what data to "lose" and how it will have the least degradation to the overall quality of the file
     - See: http://en.wikipedia.org/wiki/Lossy_compression

2) Lossless – original file is exactly reproduced when compressed file is decompressed
   - Or, $D(C(X)) = X$ where C is the compression and D is the decompression
   - Necessary for files that must be exactly restored
     - Ex: .txt, .exe, .docx, .xlsx, .java, .cpp and others
   - Will also work for audio and video, but will not realize the same compression as lossy
     - However, since it works for all file types, it can be used effectively for archiving all data in a directory or in multiple directories

- Many modern lossless compression techniques have roots in 3 algorithms:
  - Huffman
    - Variable length, 1 codeword-per-character code
  - LZ77
    - Uses a "sliding" window to compress groups of characters at a time
  - LZ78 / LZW
    - Uses a dictionary to store previously seen patterns, also compressing groups of characters at a time
- We will discuss Huffman and LZW in more detail

- What is used in actual compression programs?
  - Here are a few
    - unix compress, gif: LZW
    - pkzip, zip, gzip: LZ77 + Huffman
    - bzip2: Burrows-Wheeler transform (incl. Huffman)
    - 7zip: LZMA (Lempel Zip Markov Algorithm, variant of LZ77)
  - See Wikipedia link for more info:
    - http://en.wikipedia.org/wiki/Lossless_compression

- Background:
  - Huffman works with arbitrary bytes, but the ideas are most easily explained using character data, so we will discuss it in those terms
  - Consider extended ASCII character set:
    - 8 bits per character
    - BLOCK code, since all codewords are the same length
      - 8 bits yield 256 characters
      - In general, block codes give:
        - For K bits, 2^K characters
        - For N characters, $\lceil \log_2 N \rceil$ bits are required
    - Easy to encode and decode

- What if we could use variable length codewords, could we do better than ASCII?
  - Idea is that different characters would use different numbers of bits
  - If all characters have the same frequency of occurrence per character we cannot improve over ASCII [not considering any patterns]
- What if characters had different freqs of occurrence?
  - Ex: In English text, letters like E, A, I, S appear much more frequently than letters like Q, Z, X
  - Can we somehow take advantage of these differences in our encoding?

- First we need to make sure that variable length coding is feasible
  - Decoding an ASCII code is easy – take the next 8 bits and look up in a table
  - Decoding a variable length code is not so obvious
  - In order to decode unambiguously, variable length codes must be prefix-free
    - No codeword is a prefix of any other
    - Consider table on the right
    - Consider now bit string

0000000011

AD A DA AEB

    - This is ambiguous

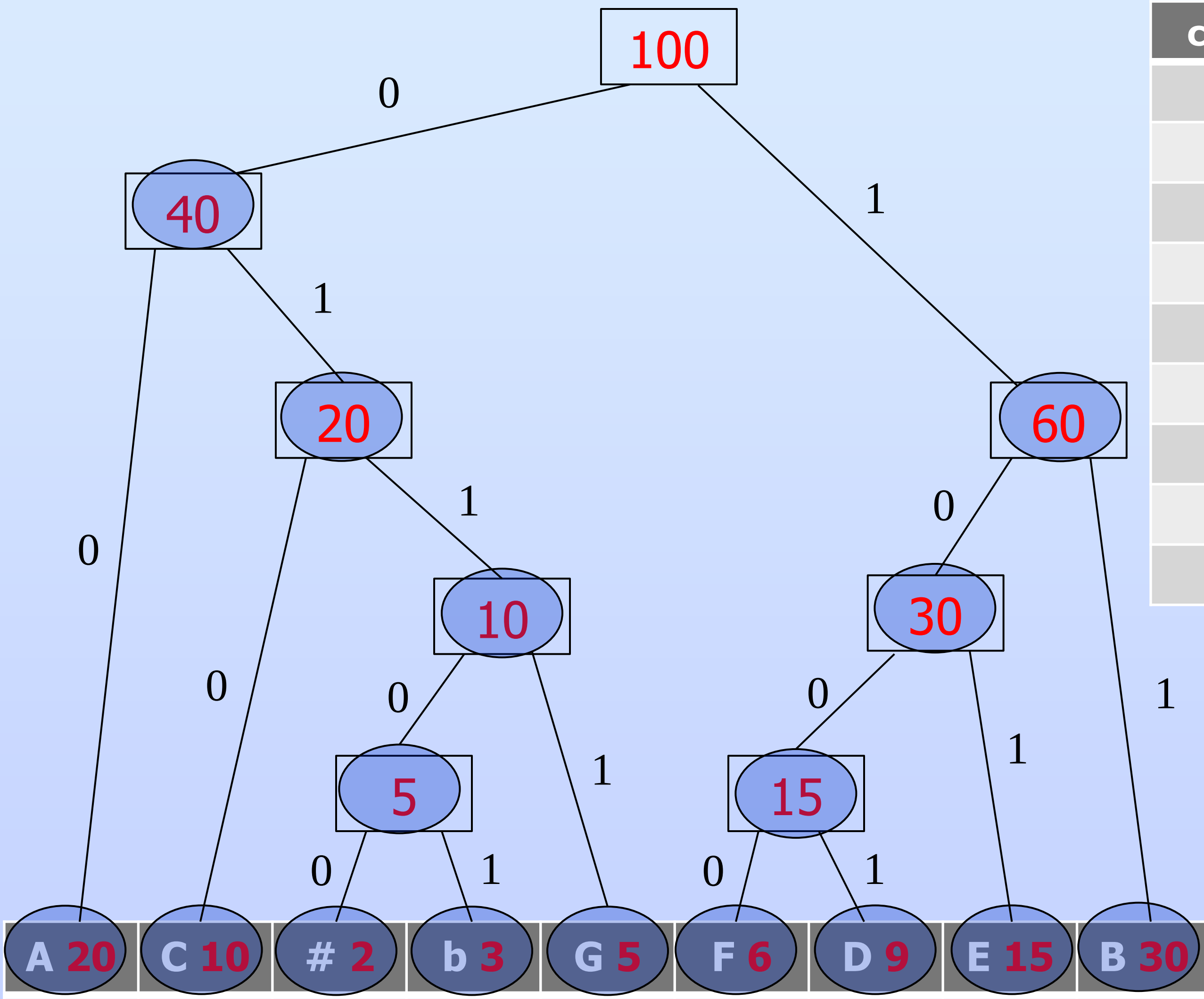
char	code
'A'	00
'B'	11
'C'	001
'D'	000
'E'	0011

  - A given character could be at the end of a codeword OR in the middle of another
- This ambiguity goes away if no codeword is a prefix of any other
  - Now if a character is at the end of one codeword, it cannot be in the middle of another
  - Removes the ambiguity
- Ok, so now how do we compress?
  - Let's use fewer bits for our more common characters, and more bits for our less common characters

- Idea of Huffman Compression:
  - Process the file of characters and build a tree that will represent the codes of all characters
    - Each path from the root to a leaf will represent a character
  - The tree will be built such that the most frequent characters will have shorter paths and the least frequent characters will have longer paths
  - Let's look at the algorithm

- Huffman Algorithm:
  - **Assume we have K characters and that each uncompressed character has some weight associated with it (i.e. frequency)**
  - **Initialize a forest, F, to have K single node trees in it, one tree per character, also storing the character's weight**
  - **while (|F| > 1)**
    - **Find the two trees, T1 and T2, with the smallest weights**
    - **Create a new tree, T, whose weight is the sum of T1 and T2**
    - **Remove T1 and T2 from the F, and add them as left and right children of T**
    - **Add T to F**
  - **Label left edges with 0 and right edges with 1**

Huffman Compression



char	freq
A	20
B	30
C	10
D	9
E	15
F	6
G	5
#	2
b	3

- Huffman Issues:
  1) Is the code correct?
     - Is the code prefix-free?
  2) Does it give good compression?
  3) How to decode?
  4) How to encode?
  5) How to determine weights/frequencies for the initial forest?
  6) How to actually implement?

1) Is the code correct?

- Based on the way the tree is formed, it is clear that the codewords are valid
- Codewords are prefix-free, since each codeword ends at a leaf
  - all original nodes corresponding to the characters end up as leaves

2) Does it give good compression?

- For a block code of N different characters, $\lceil \log_2 N \rceil$ bits are needed per character
  - Thus for a file containing M ASCII characters, 8M bits are needed

- Given Huffman codes $\{C_0, C_1, \ldots C_{N-1}\}$ for the N characters in the alphabet, each of length $|C_i|$
- Given frequencies $\{F_0, F_1, \ldots F_{N-1}\}$ in the file
  - Where sum of all frequencies = M
- The total bits required for the file is:
  - Sum from 0 to N-1 of $(|C_i| * F_i)$
    - i.e. sum of (#bits) * (freq) for each character
  - Overall total bits depends on differences in frequencies
    - The more extreme the differences, the better the compression
    - If frequencies are all the same, no compression
      - > Will actually increase the size a bit – discuss
- Let's see the result for our example

- Consider the alphabet from our example
  - If we use a block code we will need $\lceil \lg_2 9 \rceil =$ 4 bits per codeword
    - – For 100 chars we will need 400 bits
  - With Huffman we will need

    (2)*(20) + (2)*(30) + (3)*(10) +
    (4)*(9) + (3)*(15) + (4)*(6) +
    (4)*(5) + (5)*(2) + (5)*(3) =
    280 bits
  - If we think in terms of average bits
    - – (280)/100 = 2.8 bits per char
  - This is better than block code
    - – 4 bits per char

char	freq	code
A	20	00
B	30	11
C	10	010
D	9	1001
E	15	101
F	6	1000
G	5	0111
#	2	01100
b	3	01[illegible]1

3) How to decode?

- This is fairly straightforward, given that we have the Huffman tree available

```
start at root of tree and first bit of file
while not at end of file
   if current bit is a 0, go left in tree
   else go right in tree // bit is a 1
   if we are at a leaf
      output character
      go to root
   read next bit of file
```

  - Each character is a path from the root to a leaf
  - If we are not at the root when end of file is reached, there was an error in the file

- Trace on your own and we will also do one together

4) How to encode?

- This is trickier, since we are starting with characters and outputting codewords
  - Using the tree we would have to start at a leaf (first finding the correct leaf) then move up to the root (requiring parent pointers in nodes), finally reversing the resulting bit pattern
    - A lot of overhead
  - Instead, let's process the tree once (using an inorder traversal) to build an encoding TABLE
    - As we trace each codeword in the tree, put into the table
    - Result is table like in Slide 18
    - We will partially trace this in our synchronous lecture

5) How to determine weights/frequencies?

- 2-pass algorithm
  - Process the original file once to count the frequencies, then build the tree/code and process the file again, this time compressing
  - Ensures that each Huffman tree will be optimal for each file
  - However, to decode, the tree/freq information must be stored in the file
    - Likely in the front of the file, so decompress first reads tree info, then uses that to decompress the rest of the file
    - Adds extra space to file, reducing overall compression quality

Huffman Compression

  - Overhead especially reduces quality for smaller files, since the tree/freq info may add a significant percentage to the file size
  - Thus larger files have a slightly higher potential for compression with Huffman than do smaller ones
    - However, just because a file is large does NOT mean it will compress well
    - The most important factor in the compression remains the relative frequencies of the characters

- Using a static Huffman tree
  - Process a lot of "sample" files, and build a single tree that will be used for all files
  - Saves overhead of tree information, but generally is NOT a very good approach

Huffman Compression

- – There are many different file types that have very different frequency characteristics
  - Ex: .cpp file vs. .txt containing an English essay
  - .cpp file will have many ;, {, }, (,)
  - .txt file will have many a,e,i,o,u,., etc.
  - A tree that works well for one file may work poorly for another (perhaps even expanding it)

- **Adaptive single-pass** algorithm
  - Builds tree as it is encoding the file, thereby not requiring tree information to be separately stored
    - – Processes file only one time
  - We will not look at the details of this algorithm, but the LZW algorithm we will discuss next is also adaptive
    - – See: http://en.wikipedia.org/wiki/Adaptive_Huffman_coding

6) How to actually implement?

- Huffman idea / algorithm is fairly simple
- Implementing is a bit trickier, but is easier if modularized
- Consider what is needed:
  - Algorithm / Data Structures to generate and store the Huffman tree
    - Will store the tree using a dynamic binary tree of nodes
    - In order to choose the nodes to merge, we need
      - A priority queue
      - So nodes must be Comparable
      - How can we do this efficiently?

- Data structure to store the encoding info
  - Ex: Could use an array of (binary) strings indexed on the characters in the alphabet
- Algorithms to compress / expand
  - Must iterate through file and access encoding array to compress
  - Must iterate through compressed file and Huffman tree to expand
  - Must handle special situations (ex: writing tree to file, recognizing end of file, etc)
- Code to read / write binary data
  - Clearly needed, but we will defer examining this until we cover LZW compression

- See Huffman.java