Compression Algorithms

Lecture 22
Shafi Goldwasser
Representing Data

Course so far: our goal was to represent data so that it is easiest to manipulate

- Search, edit: add, delete, cut, paste.

- However, there are actually competing interests
  - easily manipulated/processed
  - short for storage and transmission

- Today our goal: represent data in the shortest possible way

- Look for algorithms to achieve this
Algorithms: Data Compression

Today: Lossless Compression: \( D = D' \)
Lossy Compression: \( D' \) close-enough to \( D \) (application dependent)
Data Compression: Lossy or Lossless

• Lossless:
  – Huffman Coding
  -- Lempel-Ziv
  – .gif

• Lossy:
  – .mp3
  – .jpg
Non Adaptive and Adaptive Algorithms

- **Non-Adaptive**: assume we know something about the data source, say the frequency of different characters.

- **Adaptive**: do not know anything, and the algorithms has to build the knowledge up by itself.

- **TODAY**: An example of each, start with non-adaptive.
Set Up

• Input: a sequence of characters
  - e.g. English language characters a,b,c...
  - Text: I am a human being

• Look for **Binary codes**: encode each character as a Binary string, also called a code word

• Output: a concatenation of code words corresponding to the input characters
Example of fixed length code

- **Fixed length code**: each character is assigned a code word of the same length.
- Suppose we have a file of 100k characters and each character is one of 8 letters [a…h]
- Need only 3 bits to represent each character: a:000, b: 001, c:010,…, h:111
- To represent the File requires: 300k bits
- Easy to encode and decode: e.g. baba ≡ 001000001000
Variable Length Code

**Variable Length:** Codewords for different characters may have different length.

Say that you knew something about the document to be compressed: the **frequency** with which each character appears.

**Idea:** use fewer bits for code words corresponding to highly frequent characters & more bits to encode characters which are rare.
Example of variable length code

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

- Frequency: 45 13 12 16 9 5
- VarLenCode: 0 101 100 111 1101 1100
- Variable Length Code Size for example:
  \[45k \times 1 + 13k \times 3 + 12k \times 3 + 9k \times 4 + 5k \times 4 = 224k\], a 25% savings on the 300k of the fixed length code

- But how do we decode?
Prefix (Free) Coding

• Codes where no codeword is a prefix of some other codeword.
• Ex: \(0 ~ 101 ~ 100 ~ 111 ~ 1101 ~ 1100\)

• Decoding (de-compressing) back to the original file is easy now: can scan from left to right, as soon as recognize a code word in the file, peel it off and continue decoding

• 01101100 = \(0101011110011011101\)

• Claim: optimal data compression can be achieved by a prefix-free code
A labeled binary tree where
• A left edge is labeled 0
• A right edge is labeled 1
• To decode a string: follow edges from the root to the character at that leaf.
• Each Leaf: labeled with a character
• Prefix free implies characters at leaves
Examples of Specific Codes as Binary Trees

- Fixed Length Codes
- Variable Length Codes
Optimal Coding

**Problem:** Given alphabet \{a_1 \ldots a_n\} & frequency counts \{f(a_1), \ldots f(a_n)\}, find the Binary prefix code that minimizes the number of bits to represent your data.

Namely, find codeword \(c(a_i) \forall a_i\) which minimizes \(B(C) = \sum f(a_i) L(c(a_i))\) where \(L(c(a_i))\) is the length of the code word \(c(a_i)\)
Equivalent Optimization Problem on Trees

**Codes:** find codeword \( c(a_i) \) for each \( a_i \) that minimizes \( B(C) = \sum f(a_i) L(c(a_i)) \) where \( L(c(ai)) \) is length of the code word \( c(ai) \)

**Trees:** Let \( d(a_i) \) be depth of leaf \( a_i \) in tree \( T \) Find tree \( T \) with \( n \) leaves where each leaf has \( f(a_i) \) associated with it so that it has the minimum weighted path \( B(T) = \sum f(a_i) d(a_i) \)

Obvious since \( d(a_i) = L(c(ai)) \)
Search for Optimal Coding

• Shannon: invented information theory
• Fano and Shannon worked together to find minimal size codes, found good heuristics
• Fano assigned the problem to his class
• Huffman solved it: not knowing that his teacher had struggled with it !!!!
• A lesson to us all.
Huffman Code

Huffman invented an algorithm to construct an optimal prefix code/tree: Huffman Code.

Greedy Algorithm Idea

Step 1: Pick two letters \( x, y \) from alphabet \( A \) with the smallest frequencies and create a sub-tree that has these two characters as leaves.

Step 2: Set frequency \( f(z) = f(x) + f(y) \). Remove \( x, y \) and add \( z \) creating a new alphabet

\( A = A \cup \{z\} - \{x, y\} \). Obviously \( |A| \) is now smaller.

Repeat until \( A \) is empty.
Example of the Algorithm

1. Take the characters and their frequencies, and sort this list by increasing frequency

E: 10, T: 7, O: 5, A: 3 →
A: 3, O: 5, T: 7, E: 10
Huffman Codes

2. All the characters are vertices of the tree:

- A: 3
- O: 5
- T: 7
- E: 10
3. Take the smallest (first) 2 vertices from the list and make them children of a new vertex having the sum of their frequencies.
4. Insert the new vertex into the sorted list of vertices waiting to be put into the tree

List of remaining vertices:

T: 7
E: 10

New list, with the new vertex inserted:

T: 7
Z: 8
E: 10

New
Huffman Codes

5. Take the first 2 vertices from the list and make them children of a new vertex having the sum of their frequencies

- **T**: 7
- **Z**: 8
- **A**: 3
- **O**: 5

New vertex with Frequency $7 + 8 = 15$
6. Insert the new vertex into the sorted list of vertices waiting to be put into the tree

List of remaining vertices:

E: 10
Z2: 15

New list, with the new vertex inserted:
7. Take the first 2 vertices from the list and make them children of a vertex having the sum of their frequencies.

New vertex with Frequency $10 + 15 = 25$
Huffman Codes

- Left branch is 0
- Right branch is 1

Huffman code:
- E: 0
- T: 10
- A: 110
- O: 111
Implementation Details, Complexity

• Stat with forest of 1-node trees representing each a in A.

• Merge them greedily, using a priority queue, sorted by the smallest frequency

• Pseudo code : section 16.3 CLR

• **Runtime**: Let n be the size of the alphabet. Then, to create the tree $O(n \log n)$, since each priority queue operation takes $O(\log n)$ and we have n operations.
Optimality?
Claim: Hoffman's Algorithm produces a tree T with \( \min B(T) \)

- **Proof** by induction on \( |A| \). When \( |A| = 2 \), the optimal tree clearly has two leaves, corresponding to strings 0 and 1, as the algorithm constructs.

- Suppose \( |A| > 2 \). The first greedy choice the algorithm does is make the two lowest-frequency characters (call them \( x \) and \( y \)) into leaf nodes, create a new node \( z \) with frequency \( f(z) = f(x) + f(y) \) and apply the algorithm to the new smaller \( A' = A - \{x,y\} \cup \{z\} \). Now \( A' \) is smaller, \( |A'| = |A| - 1 \), so by induction can assume \( T' \) for \( A' \) will be optimal.

- **Induction step**: Must show that \( T \) obtained by adding \( x \) & \( y \) as children to \( z \) in \( T' \) is optimal.
Claim: Hoffman’s Algorithm produces a tree T with min B(T) (2)

Claim: Adding x and y as L and R children to z in optimal T’ yields optimal T

Proof: Suppose not. Say ∃ tree T s.t. B(T)<B(T). We will then show a contradiction to the cost of T’ being optimal.

Case 1: x and y are siblings in T. Remove them and make their parent a leaf to get T’ for A”. Then get B(T’) < B(T’) by the following calculation.

\[ B(T') = B(T) + f(z)d(z) - f(x)d(x) - f(y)d(y) = B(T) + (f(x) + f(y))d(x) - f(x)d(x) - f(y)d(y) = B(T) - (f(x) + f(y)). \]

Similarly, \( B(T') = B(T) - (f(x) + f(y)) \).

So, B(T)<B(T) implies B(T’) < B(T’) contradiction!
Claim: Hoffman’s Algorithm produces a tree $T$ with min $B(T)$ (2)

- **Case 2:** $x$ and $y$ are not sibling leaves in the better tree $T$. Then can show that there exists another tree $T$ where $x$ and $y$ are sibling leaves, and do the argument of case 1 for $T$
  - Lemma: Optimal Huffman Tree must be a Full Binary Tree (i.e. each internal node has 2 children)
  - Proof: if not, get rid of it and replace by child, that would decrease the total cost of encoding

- **Idea:** Take sibling leaves $a$ and $b$ at maximum depth, replace $a$ with $x$ and $b$ with $y$. Show by calculation (CLR sec 16) that the weight of the new tree could not increase.
Huffman Codes Summary

- Reduce size of data by 20%-90% in general
- If no characters occur more frequently than others, then no advantage over ASCII
- **Encoding**: Given the characters & their frequencies, perform the algorithm and generate a code. Write the file using the code
- **Decoding**:  
  - *Given the Huffman tree*, figure out what each character is (possible because of prefix property)
In Practice

• Both the .mp3 and .jpg file formats use Huffman coding at one stage of the compression
• Alternative method that achieves higher compression but is slower is patented by IBM, making Huffman Codes attractive
Are we done with compression...?
Is Huffman Coding always Optimal

Huffman Coding is optimal under some assumptions

1. The compression is lossless.

When lossy compression is permitted, as for video, other algorithms can achieve much greater compression, and this is a very active area of research because people want to be able to send video and audio over the Web.
Is Huffman Coding always Optimal?

2. Assumed know all the frequencies \( f(c) \) of each character appearing. How do we get this information?

- Make two passes over the data, first to compute the \( f(c) \), second to encode the file. May be much more expensive than passing over the data once for large files residing on disk or tape.

- Assume fractions \( f(c)/n \) of each character in the file are similar to files you’ve compressed before. E.g. assume all Java programs, or PowerPoint files \( \approx \) same fractions of characters appearing.

- Estimate the fractions \( f(c)/n \) on the fly as you process the file: an Huffman coding adaptive
Is Huffman Coding always Optimal?

3. Assumed we know the set of characters (the alphabet) appearing in the file. Obvious?

- Many choices: For example, the alphabet could be the English alphabet characters, or the key words and variables names appearing in a program.

- Alphabet Matters: suppose we have a file consisting of $n$ strings $aaaa$ and $n$ strings $bbbb$ concatenated in some order. If we choose the alphabet $\{a,b\}$ then $8n$ bits are needed to encode the file. If we choose the alphabet $\{aaaa, bbbb\}$ then only $2n$ bits are needed.
Picking the Correct Alphabet as You Compress

• Correct alphabet is crucial in practical compression algorithms.
• Both the UNIX compress and GNU gzip algorithms use a greedy algorithm due to Lempel and Ziv

• Lempel-Ziv Compression:
  • One pass: compute a good alphabet (of `phrases`) in one pass while compressing.
  • Adaptive Compression Algorithm: build the source knowledge by themselves
Lempel Ziv Idea

A is a dictionary of phrases, initially $A[0] = \epsilon$

LZ breaks the text it sees into `phrases` as it goes along, storing new phrases in the dictionary $A$ till it fills up. How?

- As new text is encountered it breaks it into the \textbf{longest matching past phrase} + \textbf{1 new character to append}. Stores this as new phrase in $A$, if $A$ not full
- Phrases represented as: $(\text{loc of past longest phrase}, \text{end char})$
- File: list of such pairs
Example

$F = 0 \ 00 \ 1 \ 11 \ 10 \ 101 \ 1010 \ 00$ (to be compressed)

Build dictionary $A[1...6]$

- $A(1) = A(0)0 = 0$
- $A(2) = A(1)0 = 00$
- $A(3) = A(0)1 = 1$
- $A(4) = A(3)1 = 11$
- $A(5) = A(3)0 = 10$
- $A(6)0 = 1010$

Compressed $F = (0,0), (1,0), (0,1), (3,1), (3,0), (5,1)(6,0),(1,0)$
Remarks

• In this small example no compression is obtained, but if $A$ were large, and the same long bit strings appeared frequently, compression would be substantial. The gzip manager claims that source code and English text is typically compressed 60%-70%.

• LZ Encodes blocks of varying lengths into blocks of fixed size whereas Huffman encoded blocks of the same length into blocks of varying length.
Lempel Ziv Encoding Algorithm

- **Input:** file $F$, alphabet size $N$
- **Initialize** $A = \{A[0]=\varepsilon \}$, $i = 0$ (points to next place in file $f$ to start encoding)

Repeat till $i > |F|$

1. find $A(k)$ for $1 \leq k \leq N$ in $A[1 \ldots N]$ that matches as many bits $f_if_{i+1}f_{i+2} \ldots$ as possible
2. Let $p$ be the number of bits in $A(k)$
3. if $A$ is full upto $l<N$, add to it $(A(l+1), f_{i+p})$ \( (f_{i+p} \text{ is the first bit unmatched by } A(k)) \)
4. output $(k , f_{i+p})$
5. $i = i + p + 1$
Lempel Ziv: What can you prove?

- There are no optimality guarantees for this algorithm. It can perform badly if the nature of the file changes substantially after A is filled up.

- If assume text in the file source obeys a Markov chain model, can prove average case guarantees: HARD PROOFS.
Compression: Current Research

• Sub-linear time approximation algorithms to estimate how compressible is a file

• Clustering by compression:
  – Want to cluster similar document sources
  – Two objects are close if we can compress one given information on the other.
  – Yields: Automatic Clustering without any background knowledge of the data
  – Genomics applications