15-853: Algorithms in the Real World

Data Compression: Lectures 1 and 2

Compression in the Real World

Generic File Compression
- Files: gzip (LZ77), bzip (Burrows-Wheeler), BOA (PPM)
- Archivers: ARC (LZW), PKZip (LZW+)
- File systems: NTFS

Communication
- Fax: ITU-T Group 3 (run-length + Huffman)
- Modems: V.42bis protocol (LZW), MNP5 (run-length+Huffman)
- Virtual Connections

Multimedia
- Images: gif (LZW), jbig (context), jpeg-ls (residual), jpeg (transform+RL+arithmetic)
- Video: Blue-Ray, HDTV (mpeg-4), DVD (mpeg-2)
- Audio: iTunes, iPhone, PlayStation 3 (AAC)

Other structures
- Indexes: google, lycos
- Meshes (for graphics): edgebreaker
- Graphs
- Databases

Compression Outline

Introduction:
- Lossless vs. lossy
- Model and coder
- Benchmarks

Information Theory: Entropy, etc.
Probability Coding: Huffman + Arithmetic Coding
Applications of Probability Coding: PPM + others
Lempel-Ziv Algorithms: LZ77, gzip, compress, ...
Other Lossless Algorithms: Burrows-Wheeler
Lossy algorithms for images: JPEG, MPEG, ...
Compressing graphs and meshes: BBK
**Encoding/Decoding**

Will use "message" in generic sense to mean the data to be compressed.

The encoder and decoder need to understand common compressed format.

---

**Lossless vs. Lossy**

**Lossless:** Input message = Output message  
**Lossy:** Input message \(\approx\) Output message

Lossy does not necessarily mean loss of quality. In fact the output could be "better" than the input.  
- Drop random noise in images (dust on lens)  
- Drop background in music  
- Fix spelling errors in text. Put into better form.  

Writing is the art of lossy text compression.

---

**How much can we compress?**

For lossless compression, assuming all input messages are valid, if even one string is compressed, some other must expand.

---

**Model vs. Coder**

To compress we need a bias on the probability of messages. The model determines this bias.

Example models:  
- Simple: Character counts, repeated strings  
- Complex: Models of a human face
Quality of Compression

Runtime vs. Compression vs. Generality
Several standard corpuses to compare algorithms
\textit{e.g., Calgary Corpus}
2 books, 5 papers, 1 bibliography,
1 collection of news articles, 3 programs,
1 terminal session, 2 object files,
1 geophysical data, 1 bitmap bw image
The \textit{Archive Comparison Test} and the
\textit{Large Text Compression Benchmark} maintain a
comparison of a broad set of compression algorithms.

Comparison of Algorithms

<table>
<thead>
<tr>
<th>Program</th>
<th>Algorithm</th>
<th>Time</th>
<th>BPC</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>RK</td>
<td>LZ + PPM</td>
<td>111+115</td>
<td>1.79</td>
<td>430</td>
</tr>
<tr>
<td>BOA</td>
<td>PPM Var.</td>
<td>94+97</td>
<td>1.91</td>
<td>407</td>
</tr>
<tr>
<td>PPMD</td>
<td>PPM</td>
<td>11+20</td>
<td>2.07</td>
<td>265</td>
</tr>
<tr>
<td>IMP</td>
<td>BW</td>
<td>10+3</td>
<td>2.14</td>
<td>254</td>
</tr>
<tr>
<td>BZIP</td>
<td>BW</td>
<td>20+6</td>
<td>2.19</td>
<td>273</td>
</tr>
<tr>
<td>GZIP</td>
<td>LZ77 Var.</td>
<td>19+5</td>
<td>2.59</td>
<td>318</td>
</tr>
<tr>
<td>LZ77</td>
<td>LZ77</td>
<td>?</td>
<td>3.94</td>
<td>?</td>
</tr>
</tbody>
</table>

Compression Outline

\textbf{Introduction}: Lossy vs. Lossless, Benchmarks, ...

\textbf{Information Theory}:
- Entropy
- Conditional Entropy
- Entropy of the English Language
\textbf{Probability Coding}: Huffman + Arithmetic Coding
\textbf{Applications of Probability Coding}: PPM + others
\textbf{Lempel-Ziv Algorithms}: LZ77, gzip, compress, ...
\textbf{Other Lossless Algorithms}: Burrows-Wheeler
\textbf{Lossy algorithms for images}: JPEG, MPEG, ...
\textbf{Compressing graphs and meshes}: BBK

Information Theory

An interface between modeling and coding
\textbf{Entropy}:
- A measure of information content
\textbf{Conditional Entropy}:
- Information content based on a context
Entropy of the English Language
- How much information does each character in
"typical" English text contain?
Entropy (Shannon 1948)

For a set of messages $S$ with probability $p(s), s \in S$, the self information of $s$ is:

$$i(s) = \log \frac{1}{p(s)} = -\log p(s)$$

Measured in bits if the log is base 2.

The lower the probability, the higher the information.

Entropy is the weighted average of self information.

$$H(S) = \sum_{s \in S} p(s) \log \frac{1}{p(s)}$$

---

Conditional Entropy

The conditional probability $p(s | c)$ is the probability of $s$ in a context $c$. The conditional self information is

$$i(s | c) = -\log p(s | c)$$

The conditional information can be either more or less than the unconditional information.

The conditional entropy is the weighted average of the conditional self information

$$H(S | C) = \sum_{c \in C} \left( p(c) \sum_{s \in S} p(s | c) \log \frac{1}{p(s | c)} \right)$$

---

Entropy Example

$$p(S) = \{.25, .25, .25, .125, .125\}$$

$$H(S) = 3 \times .25 \log 4 + 2 \times .125 \log 8 = 2.25$$

$$p(S) = \{.5, .125, .125, .125, .125\}$$

$$H(S) = .5 \log 2 + 4 \times .125 \log 8 = 2$$

$$p(S) = \{.75, .0625, .0625, .0625, .0625\}$$

$$H(S) = .75 \log (4/3) + 4 \times .0625 \log 16 = 1.3$$

---

Example of a Markov Chain

\[
\begin{align*}
\text{w} & \quad \text{p(b|w)} = .9 \\
\text{w} & \quad \text{p(w|b)} = .2 \\
\text{b} & \quad \text{p(b|b)} = .8 \\
\end{align*}
\]
Entropy of the English Language

How can we measure the information per character?
- ASCII code = 7
- Entropy = 4.5 (based on character probabilities)
- Huffman codes (average) = 4.7
- Unix Compress = 3.5
- Gzip = 2.6
- Bzip = 1.9
- Entropy = 1.3 (for "text compression test")
  Must be less than 1.3 for English language.

Shannon’s experiment

Asked humans to predict the next character given the whole previous text. He used these as conditional probabilities to estimate the entropy of the English Language.

The number of guesses required for right answer:

<table>
<thead>
<tr>
<th># of guesses</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>&gt;5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probability</td>
<td>.79</td>
<td>.08</td>
<td>.03</td>
<td>.02</td>
<td>.02</td>
<td>.05</td>
</tr>
</tbody>
</table>

From the experiment he predicted

\[ H(\text{English}) = 0.6 - 1.3 \]

Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, ...
Information Theory: Entropy, etc.
Probability Coding:
- Prefix codes and relationship to Entropy
- Huffman codes
- Arithmetic codes
Applications of Probability Coding: PPM + others
Lempel-Ziv Algorithms: LZ77, gzip, compress, ...
Other Lossless Algorithms: Burrows-Wheeler
Lossy algorithms for images: JPEG, MPEG, ...
Compressing graphs and meshes: BBK

Assumptions and Definitions

Communication (or a file) is broken up into pieces called messages.
Each message come from a message set \( S = \{s_1, ..., s_m\} \) with a probability distribution \( p(s) \).
Probabilities must sum to 1. Set can be infinite.
Code \( C(s) \): A mapping from a message set to codewords, each of which is a string of bits
Message sequence: a sequence of messages
Note: Adjacent messages might be of a different types and come from a different probability distributions
Discrete or Blended

We will consider two types of coding:

**Discrete**: each message is a fixed set of bits
- Huffman coding, Shannon-Fano coding
  
  \[
  0100110001011
  \]
  
  message: 1 2 3 4

**Blended**: bits can be "shared" among messages
- Arithmetic coding
  
  \[
  010010111010
  \]
  
  message: 1, 2, 3, and 4

Uniquely Decodable Codes

A **variable length code** assigns a bit string (codeword) of variable length to every message value

\[ a = 1, \ b = 01, \ c = 101, \ d = 011 \]

What if you get the sequence of bits 1011?

Is it \( aba \), \( ca \), or, \( ad \)?

A **uniquely decodable code** is a variable length code in which bit strings can always be uniquely decomposed into its codewords.

Prefix Codes

A **prefix code** is a variable length code in which no codeword is a prefix of another word.

\[ a = 0, \ b = 110, \ c = 111, \ d = 10 \]

All prefix codes are uniquely decodable

Prefix Codes: as a tree

Can be viewed as a binary tree with message values at the leaves and 0s or 1s on the edges:

\[
\begin{array}{c}
a \\
\end{array}
\begin{array}{c}
/ \ \\
\end{array}
\begin{array}{c}
1 \\
\end{array}
\begin{array}{c}
/ \ \\
\end{array}
\begin{array}{c}
b \\
\end{array}
\begin{array}{c}
/ \ \\
\end{array}
\begin{array}{c}
c \\
\end{array}
\begin{array}{c}
/ \ \\
\end{array}
\begin{array}{c}
d \\
\end{array}
\]

\[ a = 0, \ b = 110, \ c = 111, \ d = 10 \]
Some Prefix Codes for Integers

<table>
<thead>
<tr>
<th>n</th>
<th>Binary</th>
<th>Unary</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>.010</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>.011</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>.100</td>
<td>1110</td>
<td>11000</td>
</tr>
<tr>
<td>5</td>
<td>.101</td>
<td>11110</td>
<td>11001</td>
</tr>
<tr>
<td>6</td>
<td>.110</td>
<td>111110</td>
<td>11010</td>
</tr>
</tbody>
</table>

Many other fixed prefix codes:
- Golomb, phased-binary, subexponential, ...

Average Length

For a code $C$ with associated probabilities $p(c)$ the average length is defined as

$$l_a(C) = \sum_{c \in C} p(c)l(c)$$

We say that a prefix code $C$ is optimal if for all prefix codes $C'$, $l_a(C) \leq l_a(C')$

$l(c)$ = length of the codeword $c$ (a positive integer)

Relationship to Entropy

Theorem (lower bound): For any probability distribution $p(S)$ with associated uniquely decodable code $C$,

$$H(S) \leq l_a(C)$$

Theorem (upper bound): For any probability distribution $p(S)$ with associated optimal prefix code $C$,

$$l_a(C) \leq H(S) + 1$$

Kraft McMillan Inequality

Theorem (Kraft–McMillan): For any uniquely decodable code $C$,

$$\sum_{c \in C} 2^{-l(c)} \leq 1$$

Also, for any set of lengths $L$ such that

$$\sum_{i=1}^{\mid L \mid} 2^{-l_i} \leq 1$$

there is a prefix code $C$ such that

$$l(c_i) = l_i (i = 1, \ldots, \mid L \mid)$$
Proof of the Upper Bound (Part 1)

Assign each message a length:  \( l(s) = \lceil \log(1/p(s)) \rceil \)

We then have

\[
\sum_{s \in S} 2^{-l(s)} = \sum_{s \in S} 2^{-\lceil \log(1/p(s)) \rceil}
\]

\[
\leq \sum_{s \in S} 2^{-\log(1/p(s))}
\]

\[
= \sum_{s \in S} p(s)
\]

\[
= 1
\]

So by the Kraft-McMillan inequality there is a prefix code with lengths \( l(s) \).

Proof of the Upper Bound (Part 2)

Now we can calculate the average length given \( l(s) \)

\[
l_a(S) = \sum_{s \in S} p(s)l(s)
\]

\[
= \sum_{s \in S} p(s) \lceil \log(1/p(s)) \rceil
\]

\[
\leq \sum_{s \in S} p(s) \left( 1 + \log(1/p(s)) \right)
\]

\[
= 1 + \sum_{s \in S} p(s) \log(1/p(s))
\]

\[
= 1 + H(S)
\]

And we are done.

Another property of optimal codes

**Theorem:** If \( C \) is an optimal prefix code for the probabilities \( \{p_1, \ldots, p_n\} \) then \( p_i > p_j \) implies \( l(c_i) \leq l(c_j) \)

**Proof:** (by contradiction)

Assume \( l(c_i) > l(c_j) \). Consider switching codes \( c_i \) and \( c_j \). If \( l_a \) is the average length of the original code, the length of the new code is

\[
l_a' = l_a + p_j(l(c_i) - l(c_j)) + p_i(l(c_j) - l(c_i))
\]

\[
= l_a + (p_j - p_i)(l(c_i) - l(c_j))
\]

\[
< l_a
\]

This is a contradiction since \( l_a \) is not optimal

Huffman Codes

Invented by Huffman as a class assignment in 1950.

Used in many, if not most, compression algorithms gzip, bzip, jpeg (as option), fax compression, ...

**Properties:**
- Generates optimal prefix codes
- Cheap to generate codes
- Cheap to encode and decode
- \( l_a = H \) if probabilities are powers of 2
**Huffman Codes**

**Huffman Algorithm:**
Start with a forest of trees each consisting of a single vertex corresponding to a message $s$ and with weight $p(s)$.

Repeat until one tree left:
- Select two trees with minimum weight roots $p_1$ and $p_2$
- Join into single tree by adding root with weight $p_1 + p_2$

**Example**

$p(a) = .1$, $p(b) = .2$, $p(c) = .2$, $p(d) = .5$

```
   a(.1)  b(.2)  c(.2)  d(.5)
  /   \    /    \
 (.3)  b(.2)\  c(.2)
  \       /\
 (.5)  c(.2)
     \   / \
  a(.1)
```

Step 1

Step 2

Step 3

$a=000$, $b=001$, $c=01$, $d=1$

**Encoding and Decoding**

**Encoding:** Start at leaf of Huffman tree and follow path to the root. Reverse order of bits and send.

**Decoding:** Start at root of Huffman tree and take branch for each bit received. When at leaf can output message and return to root.

There are even faster methods that can process 8 or 32 bits at a time.

**Huffman codes are “optimal”**

**Theorem:** The Huffman algorithm generates an optimal prefix code.

**Proof outline:**
Induction on the number of messages $n$.
Consider a message set $S$ with $n+1$ messages
1. Can make it so least probable messages of $S$ are neighbors in the Huffman tree
2. Replace the two messages with one message with probability $p(m_1) + p(m_2)$ making $S'$
3. Show that if $S'$ is optimal, then $S$ is optimal
4. $S'$ is optimal by induction
Problem with Huffman Coding

Consider a message with probability .999. The self information of this message is

\[-\log(.999) = .00144\]

If we were to send a 1000 such message we might hope to use 1000*.0014 = 1.44 bits.
Using Huffman codes we require at least one bit per message, so we would require 1000 bits.

Arithmetic Coding: Introduction

Allows “blending” of bits in a message sequence.
Only requires 3 bits for the example
Can bound total bits required based on sum of self information:

\[ I < 2 + \sum_{i=1}^{n} s_i \]

Used in PPM, JPEG/MPEG (as option), DMM
More expensive than Huffman coding, but integer implementation is not too bad.

Arithmetic Coding: message intervals

Assign each probability distribution to an interval range from 0 (inclusive) to 1 (exclusive).

\[
f(i) = \sum_{j=1}^{i} p(j)
\]

e.g.

<table>
<thead>
<tr>
<th>1.0</th>
<th>0.7</th>
<th>0.2</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>c  = .3</td>
<td>b  = .5</td>
<td>a  = .2</td>
<td></td>
</tr>
</tbody>
</table>

The interval for a particular message will be called the message interval (e.g for b the interval is [.2,.7])

Arithmetic Coding: sequence intervals

Code a message sequence by composing intervals.
For example: bac

The final interval is [.27,.3)
We call this the sequence interval
Arithmetic Coding: sequence intervals

To code a sequence of messages with probabilities $p_i$ ($i = 1..n$) use the following:

- $l_i = f_i$ bottom of interval
- $s_i = s_{i-1}p_i$ size of interval

Each message narrows the interval by a factor of $p_i$.

Final interval size: $s_n = \prod_{i=1}^{n} p_i$

Warning

Three types of interval:
- **message interval**: interval for a single message
- **sequence interval**: composition of message intervals
- **code interval**: interval for a specific code used to represent a sequence interval (discussed later)

Uniquely defining an interval

**Important property**: The sequence intervals for distinct message sequences of length $n$ will never overlap.

**Therefore**: specifying any number in the final interval uniquely determines the sequence.

Decoding is similar to encoding, but on each step need to determine what the message value is and then reduce interval.

Arithmetic Coding: Decoding Example

Decoding the number .49, knowing the message is of length 3:

The message is bbc.
Representing Fractions

Binary fractional representation:

- \(0.75\) = .11
- \(1/3\) = .0101
- \(11/16\) = .1011

So how about just using the smallest binary fractional representation in the sequence interval.

- e.g. \([0, .33) = .01\]
- \([.33, .66) = .1\]
- \([.66, 1) = .11\]

But what if you receive a 1?

Should we wait for another 1?

Representing an Interval

Can view binary fractional numbers as intervals by considering all completions. e.g.

- \(\text{min} = 0.11\)
- \(\text{max} = 0.1\)
- \(\text{interval} = [0.75, 1.00]\)
- \(\text{min} = 0.101\)
- \(\text{max} = 0.1011\)
- \(\text{interval} = [0.625, 0.75]\)

We will call this the code interval.

Code Intervals: example

\([0, .33) = .01\]
\([.33, .66) = .1\]
\([.66, 1) = .11\]

0
1
.01...
.11...

Note that if code intervals overlap then one code is a prefix of the other.

Lemma: If a set of code intervals do not overlap then the corresponding codes form a prefix code.

Selecting the Code Interval

To find a prefix code find a binary fractional number whose code interval is contained in the sequence interval.

- Sequence Interval
- \(\text{Code Interval (.101)}\)

Can use the fraction \(l + s/2\) truncated to \(\lfloor -\log(s/2) \rfloor + 1 + \lceil -\log(s) \rceil\) bits.
Selecting a code interval: example

\[
[0, .33) = .001 \quad [.33, .66) = .100 \quad [.66, 1) = .110
\]

\[
\begin{array}{c|c}
0 & 0.001 \\
.33 & 0.100 \\
.66 & 0.110 \\
1 & >.110
\end{array}
\]

\[l + \frac{s}{2} = .5 = .1000\ldots\]

truncated to \[1 + \lceil -\log s \rceil = 1 + \lceil -\log (.33) \rceil = 3\] bits is .100

Is this the best we can do for \([0, .33)\)?

RealArith Encoding and Decoding

**RealArithEncode:**
Determine \(l\) and \(s\) using original recurrences
Code using \(l + \frac{s}{2}\) truncated to \(1 + \lceil -\log s \rceil\) bits

**RealArithDecode:**
Read bits as needed so code interval falls within a message interval, and then narrow sequence interval.
Repeat until \(n\) messages have been decoded.

Bound on Length

**Theorem:** For \(n\) messages with self information \(\{s_1, \ldots, s_n\}\), RealArithEncode will generate at most
\[
2 + \sum_{i=1}^{n} s_i \text{ bits.}
\]

**Proof:**
\[
1 + \lceil -\log s \rceil = 1 + \lceil \frac{\sum_{i=1}^{n} P_i}{s} \rceil
\]
\[
= 1 + \sum_{i=1}^{n} \lceil -\log P_i \rceil
\]
\[
= 1 + \sum_{i=1}^{n} s_i
\]
\[
< 2 + \sum_{i=1}^{n} s_i
\]

Integer Arithmetic Coding

Problem with RealArithCode is that operations on arbitrary precision real numbers is expensive.

**Key Ideas of integer version:**
Keep integers in range \([0, R)\) where \(R=2^k\)
Use rounding to generate integer sequence interval
Whenever sequence interval falls into top, bottom or middle half, expand the interval by factor of 2
This integer Algorithm is an approximation or the real algorithm.
**Integer Arithmetic Coding**

The probability distribution as integers

Probabilities as counts:
- e.g. \( c(a) = 11 \), \( c(b) = 7 \), \( c(c) = 30 \)

\( T \) is the sum of counts
- e.g. 48 \((11+7+30)\)

Partial sums \( f \) as before:
- e.g. \( f(a) = 0 \), \( f(b) = 11 \), \( f(c) = 18 \)

Require that \( R > 4T \) so that probabilities do not get rounded to zero

- \( a = 11 \)
- \( b = 7 \)
- \( c = 30 \)
- \( T = 48 \)
- \( R = 256 \)

**Integer Arithmetic (contracting)**

\[
\begin{align*}
l_1 &= 0, \quad s_1 = R \\
s_i &= u_i - l_i + 1 \\
u_i &= l_i + \left[ s_i \cdot (f_i + s_i) / T \right] - 1 \\
l_i &= l_i + \left[ s_i \cdot f_i / T \right]
\end{align*}
\]

**Integer Arithmetic (scaling)**

If \( l \geq R/2 \) then (in top half)
- Output 1 followed by \( m \) 0s
- \( m = 0 \)
- Scale message interval by expanding by 2

If \( u < R/2 \) then (in bottom half)
- Output 0 followed by \( m \) 1s
- \( m = 0 \)
- Scale message interval by expanding by 2

If \( l \geq R/4 \) and \( u < 3R/4 \) then (in middle half)
- Increment \( m \)
- Scale message interval by expanding by 2