Arithmetic Coding

National Chiao Tung University
Chun-Jen Tsai
10/09/2014
Huffman coding is inefficient if the probability model is biased (e.g. \( P_{\text{max}} \gg 0.5 \)). Although extended Huffman coding fixes this issue, it is expensive:

- The codebook size increases exponentially w.r.t. alphabet set size

Key idea:

*Can we assign codewords to a long sequences of symbols without generating codes for all possible sequences of the same length?*

Solution: Arithmetic Coding
Arithmetic Coding Background

- **History**
  - Shannon started using cumulative density function for codeword design
  - Original idea by Elias (Huffman’s classmate) in early 1960s
  - First practical approach published in 1976, by Rissanen (IBM)
  - Made well-known by a paper in Communication of the ACM, by Witten et al. in 1987†

- **Arithmetic coding addresses two issues in Huffman coding:**
  - Integer codeword length problem
  - Adaptive probability model problem

Two-Steps of Coding Messages

- To encode a long message into a single codeword without using a large codebook, we must
  - Step I: use a (hash) function to compute an ID (or tag) for the message. The function should be invertible
  - Step II: Given an ID (tag), assign a codeword for it using simple rules (e.g. maybe something similar to Golomb codes?), hence, there is no need to build a large codebook

- Arithmetic coding is an example of how these two steps can be achieved by using cumulative density function (CDF) as the hash function
CDF for Tag Generation

- Given a source alphabet $\mathcal{A} = \{ a_1, a_2, \ldots, a_m \}$, a random variable $X(a_i) = i$, and a probability model $P. P(X = i) = P(a_i)$. The CDF is defined as:

$$F_X(i) = \sum_{k=1}^{i} P(X = k).$$

- CDF divides $[0, 1)$ into disjoint subintervals:

  - tag for $a_i$ can be any value that belongs to $[ F_X(i-1), F_X(i) )$
Example of Tag Generation

In arithmetic coding, each symbol is mapped to an interval.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>.2</td>
<td>[0, 0.2)</td>
</tr>
<tr>
<td>e</td>
<td>.3</td>
<td>[0.2, 0.5)</td>
</tr>
<tr>
<td>i</td>
<td>.1</td>
<td>[0.5, 0.6)</td>
</tr>
<tr>
<td>o</td>
<td>.2</td>
<td>[0.6, 0.8)</td>
</tr>
<tr>
<td>u</td>
<td>.1</td>
<td>[0.8, 0.9)</td>
</tr>
<tr>
<td>!</td>
<td>.1</td>
<td>[0.9, 1.0)</td>
</tr>
</tbody>
</table>

Message: “eaii!”
Tag Selection for a Message (1/2)

- Since the intervals of messages are disjoint, we can pick any values from the interval as the tag
  - A popular choice is the lower limit of the interval
- Single symbol example: if the mid-point of the interval $[F_X(a_{i-1}), F_X(a_i))$ is used as the tag $T_X(a_i)$ of symbol $a_i$, then

$$T_X(a_i) = \sum_{k=1}^{i-1} P(X = k) + \frac{1}{2} P(X = i)$$

$$= F_X(i-1) + \frac{1}{2} P(X = i).$$

Note that: the function $T_X(a_i)$ is invertible.
Tag Selection for a Message  (2/2)

- To generate a unique tag for a long message, we need an ordering on all message sequences
  - A logical choice of such ordering rule is the lexicographic ordering of the message

- With lexicographical ordering, for all messages of length $m$, we have
  \[ T_X^{(m)}(x_i) = \sum_{y < x_i} P(y) + \frac{1}{2} P(x_i), \]
  where $y < x_i$ means $y$ precedes $x_i$ in the ordering of all messages.

- Bad news: need $P(y)$ for all $y < x_i$ to compute $T_X(x_i)$!
Recursive Computation of Tags (1/3)

- Assume that we want to code the outcome of rolling a fair die for three times. Let’s compute the upper and lower limits of the message “3-2-2.”
  - For the first outcome “3,” we have
    \[ l^{(1)} = F_X(2), \quad u^{(1)} = F_X(3). \]
  - For the second outcome “2,” we have upper limit
    \[
    F_X^{(2)}(32) = [P(x_1 = 1) + P(x_1 = 2)] + P(x = 31) + P(x = 32)
    = F_X(2) + P(x_1 = 3)P(x_2 = 1) + P(x_1 = 3)P(x_2 = 2)
    = F_X(2) + P(x_1 = 3)F_X(2) = F_X(2) + [F_X(3) - F_X(2)]F_X(2).
    \]
    Thus, \[ u^{(2)} = l^{(1)} + (u^{(1)} - l^{(1)})F_X(2). \]
    Similarly, the lower limit \( F_X^{(2)}(31) \) is \[ l^{(2)} = l^{(1)} + (u^{(1)} - l^{(1)})F_X(1). \]
For the third outcome “2,” we have

\[ l^{(3)} = F_X^{(3)}(321), \quad u^{(3)} = F_X^{(3)}(322). \]

Using the same approach above, we have

\[ F_X^{(3)}(321) = F_X^{(2)}(31) + [F_X^{(2)}(32) - F_X^{(2)}(31)]F_X(1). \]
\[ F_X^{(3)}(322) = F_X^{(2)}(31) + [F_X^{(2)}(32) - F_X^{(2)}(31)]F_X(2). \]

Therefore,

\[ l^{(3)} = l^{(2)} + (u^{(2)} - l^{(2)})F_X(1), \quad \text{and} \]
\[ u^{(3)} = l^{(2)} + (u^{(2)} - l^{(2)})F_X(2). \]
In general, we can show that for any sequence \( x = (x_1, x_2, \ldots, x_n) \),

\[
\begin{align*}
    l^{(n)} &= l^{(n-1)} + (u^{(n-1)} - l^{(n-1)}) F_X(x_{n-1}) \\
    u^{(n)} &= l^{(n-1)} + (u^{(n-1)} - l^{(n-1)}) F_X(x_n).
\end{align*}
\]

If the mid-point is used as the tag, then

\[
T_X(x) = \frac{u^{(n)} + l^{(n)}}{2}.
\]

Note that we only need the CDF of the source alphabet to compute the tag of any long messages!
Deciphering The Tag

- The algorithm to deciphering the tag is quite straightforward:
  1. Initialize $l(0) = 0$, $u(0) = 1$.
  2. For each $k$, $k \geq 1$, find $t^* = (T_X(x) - l^{(k-1)})/(u^{(k-1)} - l^{(k-1)})$.
  3. Find the value of $x_k$ for which $F_X(x_k - 1) \leq t^* \leq F_X(x_k)$.
  4. Update $u^{(k)}$ and $l^{(k)}$.
  5. If there are more symbols, go to step 2.

- In practice, a special “end-of-sequence” symbol is used to signal the end of a sequence.
Example of Decoding Tag

Given $\mathcal{A} = \{1, 2, 3\}$, $F_X(1) = 0.8$, $F_X(2) = 0.82$, $F_X(3) = 1$, $l^{(0)} = 0$, $u^{(0)} = 1$. If the tag is $T_X(x) = 0.772352$, what is $x$?

$$t^* = (0.772352 - 0)/(1 - 0) = 0.772352$$
$$F_X(0) = 0 \leq t^* \leq 0.8 = F_X(1)$$
$$l^{(1)} = 0, u^{(1)} = 0.8.$$  

$$t^* = (0.772352 - 0)/(0.8 - 0) = 0.96544$$
$$F_X(2) = 0.82 \leq t^* \leq 1 = F_X(3)$$
$$l^{(2)} = 0.656, u^{(2)} = 0.8.$$  

$$t^* = (0.772352 - 0.656)/(0.8 - 0.656) = 0.808$$
$$F_X(1) = 0.8 \leq t^* \leq 0.82 = F_X(2)$$
$$l^{(3)} = 0.7712, u^{(3)} = 0.77408.$$  

$$t^* = (0.772352 - 0.7712)/(0.77408 - 0.7712) = 0.4$$
$$F_X(1) = 0 \leq t^* \leq 0.8 = F_X(1)$$

Note:

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_{n-1})$$
$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n)$$
Binary Code for the Tag

- If the **mid-point** of an interval is used as the tag $T_X(x)$, a binary code for $T_X(x)$ is the binary representation of the number truncated to $l(x) = \left\lfloor \log(1/P(x)) \right\rfloor + 1$ bits.

- For example, $\mathcal{A} = \{ a_1, a_2, a_3, a_4 \}$ with probabilities { 0.5, 0.25, 0.125, 0.125 }, a binary code for each symbol is as follows:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>$F_X$</th>
<th>$\overline{T}_X$</th>
<th>In Binary</th>
<th>$\left\lfloor \log(1/P(x)) \right\rfloor + 1$</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.500</td>
<td>.2500</td>
<td>.0100</td>
<td>2</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>.750</td>
<td>.6250</td>
<td>.1010</td>
<td>3</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>.875</td>
<td>.8125</td>
<td>.1101</td>
<td>4</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>1.000</td>
<td>.9375</td>
<td>.1111</td>
<td>4</td>
<td>1111</td>
</tr>
</tbody>
</table>

- The binary code for a message is defined recursively!
Unique Decodability of the Code

- Note that the tag $T_X(x)$ uniquely specifies the interval $[F_X(x-1), F_X(x))$, if $\lfloor T_X(x) \rfloor_{l(x)}$ is still in the interval, it is unique. Since $\lfloor T_X(x) \rfloor_{l(x)} > F_X(x-1)$ because $1/2^l(x) < P(x)/2 = T_X(x) - F_X(x-1)$, we know $\lfloor T_X(x) \rfloor_{l(x)}$ is still in the interval.

- To show that the code is uniquely decodable, we can show that the code is a prefix code. This is true because $\lfloor T_X(x) \rfloor_{l(x)}, \lfloor T_X(x) \rfloor_{l(x)} + (1/2^l(x)) \subset [F_X(x-1), F_X(x))$. Therefore, any other code outside the interval $[F_X(x-1), F_X(x))$ will have a different $l(x)$-bit prefix.
The average code length of a source $A^{(m)}$ is:

$$l_{A^{(m)}} = \sum P(x)l(x) = \sum P(x)\left[ \log \frac{1}{P(x)} + 1 \right]$$

$$< \sum P(x)\left[ \log \frac{1}{P(x)} + 1 + 1 \right] = -\sum P(x)\log P(x) + 2\sum P(x)$$

$$= H(X^{(m)}) + 2.$$ 

Recall that for i.i.d. sources, $H(X^{(m)}) = mH(X)$. Thus,

$$H(X) \leq l_{A} \leq H(X) + \frac{2}{m}.$$
Arithmetic Coding Implementation

- Previous formulation for coding works, but we need real numbers with undetermined precision to work
  - Eventually $l^{(n)}$ and $u^{(n)}$ will be close enough to identify the message, but could take long iterations
  - To avoid recording long real numbers, we can sequentially outputs known digits, and rescale the interval as follows:

    $E_1$: $[0, 0.5) \rightarrow [0, 1);$ \quad $E_1(x) = 2x$
    $E_2$: $[0.5, 1) \rightarrow [0, 1);$ \quad $E_2(x) = 2(x - 0.5)$.

- As interval narrows, we have one of three cases
  1. $[l^{(n)}, u^{(n)}] \subset [0, 0.5) \rightarrow$ output 0, then perform $E_1$ rescale
  2. $[l^{(n)}, u^{(n)}] \subset [0.5, 1) \rightarrow$ output 1, then perform $E_2$ rescale
  3. $l^{(n)} \in [0, 0.5), u^{(n)} \in [0.5, 1)$ \rightarrow output undetermined
Implementation Key Points

- **Principle**
  - Scale and shift simultaneously $x$, upper bound, and lower bound will give us the same relative location of the tag.

- **Encoder**
  - Once we reach case 1 or 2, we can ignore the other half of $[0,1)$ by sending all the prefix bits so far to the decoder.
  - Rescale tag interval to $[0, 1)$ by using $E_1(x)$ or $E_2(x)$.

- **Decoder**
  - Scale the tag interval in sync with the encoder.
Consider $X(a_i) = i$, encode 1 3 2 1, given the model:

Given $\mathcal{A} = \{1, 2, 3\}$, $F_X(1) = 0.8$, $F_X(2) = 0.82$, $F_X(3) = 1$, $l(0) = 0$, $u(0) = 1$.

<table>
<thead>
<tr>
<th>Input: 1321</th>
<th>Input: *321</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l^{(1)} = l^{(0)} + (u^{(0)} - l^{(0)})F_X(0) = 0$</td>
<td>$l^{(2)} = 0.656$, $u^{(2)} = 0.8$</td>
</tr>
<tr>
<td>$u^{(1)} = l^{(0)} + (u^{(0)} - l^{(0)})F_X(1) = 0.8$</td>
<td>$[l^{(2)}, u^{(2)}] \subset [0.5, 1) \rightarrow \text{Output: } 1$</td>
</tr>
</tbody>
</table>

Output: 1

$E_2$ rescale:

$[l^{(2)}] = 2 \times (0.656 - 0.5) = 0.312$

$u^{(2)} = 2 \times (0.8 - 0.5) = 0.6$

Output: 1
Tag Generation with Scaling (2/3)

Input: **21
\[ l^{(3)} = l^{(2)} + (u^{(2)} - l^{(2)})F_X(1) = 0.5424 \]
\[ u^{(3)} = l^{(2)} + (u^{(2)} - l^{(2)})F_X(2) = 0.54816 \]
\[ [l^{(3)}, u^{(3)}] \subset [0.5, 1) \rightarrow \text{Output: 11} \]

\[ E_1 \text{ rescale:} \]
\[ l^{(3)} = 2 \times (0.5424 - 0.5) = 0.0848 \]
\[ u^{(3)} = 2 \times (0.54816 - 0.5) = 0.09632 \]
\[ [l^{(3)}, u^{(3)}] \subset [0, 0.5) \rightarrow \text{Output: 110} \]

\[ E_1 \text{ rescale:} \]
\[ l^{(3)} = 2 \times 0.1696 = 0.3392 \]
\[ u^{(3)} = 2 \times 0.19264 = 0.38528 \]
\[ [l^{(3)}, u^{(3)}] \subset [0, 0.5) \rightarrow \text{Output: 1100} \]

\[ E_2 \text{ rescale:} \]
\[ l^{(3)} = 2 \times 0.0848 = 0.1696 \]
\[ u^{(3)} = 2 \times 0.09632 = 0.19264 \]
\[ [l^{(3)}, u^{(3)}] \subset [0, 0.5) \rightarrow \text{Output: 1100} \]

\[ \]
The final symbol ‘1’ in the input sequence results in:

End-of-sequence symbol can be a pre-defined value in \([l^{(n)}, u^{(n)}]\). If we pick 0.510 as EOS†, the final output of the sequence is 11000110…0.

Note that 0.110001 = 2^{-1} + 2^{-2} + 2^{-6} = 0.765625.

† The number of bits for the EOS symbol shall be the same as the decoder word-length.
Tag Decoding Example (1/2)

- Assume word length is set to 6, the input sequence is \(110001100000\).

Input tag: \(110001100000\)
Output: 1

\[t^* = \frac{(0.765625 - 0)}{(0.8 - 0)} = 0.9579\]
\[F_X(2) = 0.82 \leq t^* \leq 1 = F_X(3)\]
Output: 13
\[l^{(2)} = 0 + (0.8 - 0)\times F_X(2) = 0.656,\]
\[u^{(2)} = 0 + (0.8 - 0)\times F_X(3) = 0.8\]

\(E_2\) rescale:
\[l^{(2)} = 2\times(0.656 - 0.5) = 0.312\]
\[u^{(2)} = 2\times(0.8 - 0.5) = 0.6\]
Update tag: \(*10001100000\)

Input tag: \(*10001100000\)

\[t^* = \frac{(0.546875 - 0.312)}{(0.6 - 0.312)} = 0.8155\]
\[F_X(1) = 0.8 \leq t^* \leq 0.82 = F_X(2)\]
Output: 132
\[l^{(3)} = 0.5424, u^{(3)} = 0.54816\]

\(E_2\) rescale:
\[l^{(3)} = 2\times(0.5424 - 0.5) = 0.0848\]
\[u^{(3)} = 2\times(0.54816 - 0.5) = 0.09632\]
Update tag: \(**0001100000\)
Tag Decoding Example (2/2)

\[ E_1 \text{ rescale:} \]
\[ l^{(3)} = 2 \times 0.0848 = 0.1696 \]
\[ u^{(3)} = 2 \times 0.09632 = 0.19264 \]
Update tag: **001**\underline{100000}

\[ E_1 \text{ rescale:} \]
\[ l^{(3)} = 2 \times 0.1696 = 0.3392 \]
\[ u^{(3)} = 2 \times 0.19264 = 0.38528 \]
Update tag: ****0\underline{1}0000

\[ E_1 \text{ rescale:} \]
\[ l^{(3)} = 2 \times 0.3392 = 0.6784 \]
\[ u^{(3)} = 2 \times 0.38528 = 0.77056 \]
Update tag: *****1\underline{00000}

\[ E_2 \text{ rescale:} \]
\[ l^{(3)} = 2 \times (0.6784 - 0.5) = 0.3568 \]
\[ u^{(3)} = 2 \times (0.77056 - 0.5) = 0.54112 \]
Update tag: *****\underline{00000}

Now, since the final pattern 100000 is the EOS symbol, we do not have anymore input bits.

The final digit is 1 because the final interval is in
\[ F_x(0) = 0 \leq l^{(3)} \leq u^{(3)} \leq 0.8 = F_x(1) \]
Output: 1321
Rescaling in Case 3

- If the limits of the interval contains 0.5, i.e., $l^{(n)} \in [0.25, 0.5), u^{(n)} \in [0.5, 0.75)$, we can perform rescaling by $E_3$: $[0.25, 0.75) \rightarrow [0, 1); E_3(x) = 2(x - 0.25)$.

- If we decide to perform $E_3$ rescaling, what output do we produce for an $E_3$ rescale operation?
  - Recall that, for $E_1$, 0 is sent, and for $E_2$, 1 is sent
  - For $E_3$, it depends on the non-$E_3$ rescale operation after it. That is, we can keep count of consecutive $E_3$ rescales and issue the same number of zeros/ones after the first encounter of $E_2/E_1$ rescale operation. For example, $E_3E_3E_3E_2 \rightarrow 1000$. Only used to properly rescale the intervals at the decoder!
Integer Implementation

- Assume that the interval limits are represented using integer word length of \( n \), thus
  
  \[ [0.0, 1.0) \rightarrow [00\ldots0, 11\ldots1), \text{ and } 0.5 \rightarrow 10\ldots0. \]

- Furthermore, if symbol \( j \) occurs \( n_j \) times in a total of \( n_{\text{total}} \) symbols, then the CDF can be estimated by
  
  \[ F_X(k) = \frac{CC(k)}{n_{\text{total}}} \]

  where \( CC(k) \) is the cumulative count defined by

  \[ CC(k) = \sum_{i=1}^{k} n_i. \]

  Thus, interval limits are:

  \[
  l^{(n)} = l^{(n-1)} + \left\lfloor (u^{(n-1)} - l^{(n-1)} + 1) \times CC(x_n) / n_{\text{total}} \right\rfloor
  \]

  \[
  u^{(n)} = l^{(n-1)} + \left\lfloor (u^{(n-1)} - l^{(n-1)} + 1) \times CC(x_n) / n_{\text{total}} \right\rfloor - 1.
  \]
Encoder (Integer Implementation)

Initialize \( l \) and \( u \).
Get symbol.

\[
\begin{align*}
  l &\leftarrow l + \left\lfloor \frac{(u-l+1) \times \text{Cum\_Count}(x-1)}{\text{Total\_Count}} \right\rfloor \\
u &\leftarrow l + \left\lfloor \frac{(u-l+1) \times \text{Cum\_Count}(x)}{\text{Total\_Count}} \right\rfloor - 1
\end{align*}
\]

while (MSB of \( u \) and \( l \) are both equal to \( b \) or \( E_3 \) condition holds)
if (MSB of \( u \) and \( l \) are both equal to \( b \))
{  
  send \( b \)
  shift \( l \) to the left by 1 bit and shift 0 into LSB
  shift \( u \) to the left by 1 bit and shift 1 into LSB
  while(\( \text{Scale3} > 0 \))
  {
    send complement of \( b \)
    decrement \( \text{Scale3} \)
  }
}

if \( (E_3 \) condition holds)
{  
  shift \( l \) to the left by 1 bit and shift 0 into LSB
  shift \( u \) to the left by 1 bit and shift 1 into LSB
  complement (new) MSB of \( l \) and \( u \)
  increment \( \text{Scale3} \)
}

number of digits for \( E_3 \) scaling operations
Decoder (Integer Implementation)

Initialize $l$ and $u$.
Read the first $m$ bits of the received bitstream into tag $t$.

$k = 0$

while \[
\left\lceil \frac{(t - l + 1) \times \text{Total\_Count} - 1}{u - l + 1} \right\rceil \geq \text{Cum\_Count}(k) \]

$k \leftarrow k + 1$

Decode symbol $x$.

$l \leftarrow l + \left\lceil \frac{(u - l + 1) \times \text{Cum\_Count}(x - 1)}{\text{Total\_Count}} \right\rceil$

$u \leftarrow l + \left\lceil \frac{(u - l + 1) \times \text{Cum\_Count}(x)}{\text{Total\_Count}} \right\rceil - 1$

while (MSB of $u$ and $l$ are both equal to $b$ or $E_3$ condition holds)

if (MSB of $u$ and $l$ are both equal to $b$)

  \{
  \begin{align*}
  &\text{shift} \ l \text{ to the left by 1 bit and shift 0 into LSB} \\
  &\text{shift} \ u \text{ to the left by 1 bit and shift 1 into LSB} \\
  &\text{shift} \ t \text{ to the left by 1 bit and read next bit from received bitstream into LSB}
  \end{align*}
  \}

if ($E_3$ condition holds)

  \{
  \begin{align*}
  &\text{shift} \ l \text{ to the left by 1 bit and shift 0 into LSB} \\
  &\text{shift} \ u \text{ to the left by 1 bit and shift 1 into LSB} \\
  &\text{shift} \ t \text{ to the left by 1 bit and read next bit from received bitstream into LSB} \\
  &\text{complement (new) MSB of} \ l, \ u, \ \text{and} \ t
  \end{align*}
  \}
Binary Arithmetic Coders

- Most arithmetic coders used today are binary coders, i.e., the alphabet = \{0, 1\}
- For non-binary data sources, you must apply a “binarization” process to turn the messages into binary messages before coding
- Because there are only two letters in the alphabet, the probability model consists of a single number.
  - Easier to adopt context-sensitive probability models
  - Easier to adopt “quantized” probabilities for simplification of calculations
Arithmetic vs. Huffman Coding

- Average code length of \( m \) symbol sequence:
  - Arithmetic code: \( H(X) \leq l_A < H(X) + 2/m \)
  - Extended Huffman code: \( H(X) \leq l_H < H(X) + 1/m \)
- Both codes have same asymptotic behavior
- Extended Huffman coding requires large codebook for \( m^n \) extended symbols while AC does not
- In general,
  - Small alphabet sets favor Huffman coding
  - Skewed distributions favor arithmetic coding
- Arithmetic coding can adapt to input statistics easily
Adaptive Arithmetic Coding

- In arithmetic coding, since coding of each new incoming symbol is based on a probability table, we can update the table easily as long as the transmitter and receiver stays in sync.
- Adaptive arithmetic coding:
  - Initially, all symbols are assigned a fixed initial probability (e.g. occurrence count is set to 1).
  - After a symbol is encoded, update symbol probability (i.e. occurrence count) in both transmitter and receiver.
  - Note that the occurrence count may overflow, we have to rescale the count before this happens. For example:

\[ c = \left\lfloor c / 2 \right\rfloor. \]
Applications: Image Compression

- Compression of pixel values directly

<table>
<thead>
<tr>
<th>Image Name</th>
<th>Bits/Pixel</th>
<th>Total Size (bytes)</th>
<th>Compression Ratio (arithmetic)</th>
<th>Compression Ratio (Huffman)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sena</td>
<td>6.52</td>
<td>53,431</td>
<td>1.23</td>
<td>1.16</td>
</tr>
<tr>
<td>Sensin</td>
<td>7.12</td>
<td>58,306</td>
<td>1.12</td>
<td>1.27</td>
</tr>
<tr>
<td>Earth</td>
<td>4.67</td>
<td>38,248</td>
<td>1.71</td>
<td>1.67</td>
</tr>
<tr>
<td>Omaha</td>
<td>6.84</td>
<td>56,061</td>
<td>1.17</td>
<td>1.14</td>
</tr>
</tbody>
</table>

- Compression of pixel differences

<table>
<thead>
<tr>
<th>Image Name</th>
<th>Bits/Pixel</th>
<th>Total Size (bytes)</th>
<th>Compression Ratio (arithmetic)</th>
<th>Compression Ratio (Huffman)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sena</td>
<td>3.89</td>
<td>31,847</td>
<td>2.06</td>
<td>2.08</td>
</tr>
<tr>
<td>Sensin</td>
<td>4.56</td>
<td>37,387</td>
<td>1.75</td>
<td>1.73</td>
</tr>
<tr>
<td>Earth</td>
<td>3.92</td>
<td>32,137</td>
<td>2.04</td>
<td>2.04</td>
</tr>
<tr>
<td>Omaha</td>
<td>6.27</td>
<td>51,393</td>
<td>1.28</td>
<td>1.26</td>
</tr>
</tbody>
</table>