CSE 589
Applied Algorithms
Spring 1999

Arithmetic Coding
Dictionary Coding

Arithmetic Coding

- Huffman coding works well for larger alphabets and gets to within one bit of the entropy lower bound. Can we do better. Yes
- Basic idea in arithmetic coding:
  - represent each string $x$ of length $n$ by an interval $[l, r)$ in $[0, 1)$.
  - The width $r - l$ of the interval $[l, r)$ represents the probability of $x$ occurring.
  - The interval $[l, r)$ can itself be represented by any number, called a tag, within the half open interval.
  - The $k$ significant bits of the tag $t_1 t_2 \ldots t_k$ is the code of $x$. That is, $t_1 t_2 \ldots t_k 000\ldots$ is in the interval $[l, r)$.

Example of Arithmetic Coding (1)

- $P(a) = 1/3$, $P(b) = 2/3$.

<table>
<thead>
<tr>
<th>$a$</th>
<th>$0/27$</th>
<th>$0.00000000\ldots$</th>
<th>$0.0001001\ldots$</th>
<th>$0$</th>
<th>$\text{aaa}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ab$</td>
<td>$0.00011000\ldots$</td>
<td>$0.00001010\ldots$</td>
<td>$0.00001100\ldots$</td>
<td>$001$</td>
<td>$\text{abb}$</td>
</tr>
<tr>
<td>$abb$</td>
<td>$0.01001101\ldots$</td>
<td>$0.01001010\ldots$</td>
<td>$0.01001110\ldots$</td>
<td>$011$</td>
<td>$\text{bab}$</td>
</tr>
<tr>
<td>$bbb$</td>
<td>$0.10101000\ldots$</td>
<td>$0.10101100\ldots$</td>
<td>$0.10101110\ldots$</td>
<td>$101$</td>
<td>$\text{bba}$</td>
</tr>
<tr>
<td>$bba$</td>
<td>$0.11011000\ldots$</td>
<td>$0.11011110\ldots$</td>
<td>$0.11101011\ldots$</td>
<td>$111$</td>
<td>$\text{bbb}$</td>
</tr>
<tr>
<td>$bba$</td>
<td>$0.11011110\ldots$</td>
<td>$0.11011111\ldots$</td>
<td>$0.11101010\ldots$</td>
<td>$111$</td>
<td>$\text{bbb}$</td>
</tr>
</tbody>
</table>

SomeTags are Better than Others

- $P(a) = 1/3$, $P(b) = 2/3$.

<table>
<thead>
<tr>
<th>$a$</th>
<th>$0/27$</th>
<th>$0.00000000\ldots$</th>
<th>$0.0001001\ldots$</th>
<th>$0$</th>
<th>$\text{aaa}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ab$</td>
<td>$0.00011000\ldots$</td>
<td>$0.00001010\ldots$</td>
<td>$0.00001100\ldots$</td>
<td>$001$</td>
<td>$\text{abb}$</td>
</tr>
<tr>
<td>$abb$</td>
<td>$0.01001101\ldots$</td>
<td>$0.01001010\ldots$</td>
<td>$0.01001110\ldots$</td>
<td>$011$</td>
<td>$\text{bab}$</td>
</tr>
<tr>
<td>$bbb$</td>
<td>$0.10101000\ldots$</td>
<td>$0.10101100\ldots$</td>
<td>$0.10101110\ldots$</td>
<td>$101$</td>
<td>$\text{bba}$</td>
</tr>
<tr>
<td>$bba$</td>
<td>$0.11011000\ldots$</td>
<td>$0.11011110\ldots$</td>
<td>$0.11101011\ldots$</td>
<td>$111$</td>
<td>$\text{bbb}$</td>
</tr>
</tbody>
</table>

Code Generation from Tag

- If binary tag is $t_1 t_2 \ldots t_k = (l-r)/2$ in $[l, r)$ then we want to choose $k$ to form the code $t_1 t_2 \ldots t_k$.
- Short code:
  - choose $k$ to be as small as possible so that $l < t_1 t_2 \ldots t_k 000\ldots < r$.
- Guaranteed code:
  - choose $k = \lceil \log_2 \left( \frac{r-l}{2} \right) \rceil + 1$
  - $l < t_1 t_2 \ldots t_k b_1 b_2 b_3 \ldots < r$ for any bits $b_1 b_2 b_3 \ldots$
  - for fixed length strings provides a good prefix code.
- Example: $0.00000000\ldots 0.00001001\ldots$, tag $= 0.00001001\ldots$
  - Short code: 0
  - Guaranteed code: 000001
Arithmetic Coding Algorithm

- \( P(a_1), P(a_2), \ldots, P(a_m) \)
- \( C(a_i) = P(a_1) + P(a_2) + \ldots + P(a_{i-1}) \)
- Encode \( x_1x_2\ldots x_n \)

\[
\text{Initialize } l := 0 \text{ and } r := 1; \\
\text{for } i = 1 \text{ to } n \text{ do } \\
w := r - l; \\
l := l + wC(x_i); \\
r := l + wP(x_i); \\
t := (l+r)/2; \\
\text{choose code for the tag}
\]

Arithmetic Coding Example

- \( P(a) = 1/4, P(b) = 1/2, P(c) = 1/4 \)
- \( C(a) = 0, C(b) = 1/4, C(c) = 3/4 \)
- \( abca \)

<table>
<thead>
<tr>
<th>symbol</th>
<th>( w )</th>
<th>( l )</th>
<th>( r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1/8</td>
<td>5/32</td>
<td>6/32</td>
</tr>
<tr>
<td>b</td>
<td>1/4</td>
<td>1/16</td>
<td>3/16</td>
</tr>
<tr>
<td>c</td>
<td>1/8</td>
<td>5/32</td>
<td>6/32</td>
</tr>
</tbody>
</table>

\[ t = \frac{(l+r)/2}{5/32 + 21/128} = 41/256 = .001010010... \]
\[ l = .001010000... \]
\[ r = .001010100... \]
\[ \text{code} = 00101 \]
\[ \text{prefix code} = 00101001 \]

Decoding (1)

- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...

\[
\text{output a}
\]

Decoding (2)

- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...

\[
\text{output a}
\]

Decoding (3)

- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...

\[
\text{output a}
\]

Arithmetic Decoding Algorithm

- \( P(a_1), P(a_2), \ldots, P(a_m) \)
- \( C(a_i) = P(a_1) + P(a_2) + \ldots + P(a_{i-1}) \)
- Decode \( b_1b_2\ldots b_m \), number of symbols is \( n \)

\[
\text{Initialize } l := 0 \text{ and } r := 1; \\
t := b_1b_2\ldots b_m000... \\
\text{for } i = 1 \text{ to } n \text{ do } \\
w := r - l; \\
\text{find } j \text{ such that } l + wC(a_j) \leq t < l + w(C(a_j)+P(a_j)) \\
\text{output } a_j; \\
l := l + wC(a_j); \\
r := l + wP(a_j); \\
\]

CSE 589 - Lecture 12 - Spring 1999
Decoding Example

- P(a) = 1/4, P(b) = 1/2, P(c) = 1/4
- C(a) = 0, C(b) = 1/4, C(c) = 3/4
- 00101

```
tag = .00101000... = 5/32
w   l    r    output
  0   1
 1/4 1/16 3/16 b
 1/8 5/32 6/32 c
1/32 5/32 21/128 a
```

Practical Arithmetic Coding

- Scaling:
  - By scaling we can keep l and r in a reasonable range of values so that w = r - l does not underflow.
  - The code can be produced progressively, not at the end.
  - Complicates decoding some.
- Integer arithmetic coding avoids floating point altogether.

Coding with Scaling (1)

- Assume the length is known to be 3.
- bba

```
l = 1/9  r = 22/27
l = 2/9  r = 22/27
```

Coding with Scaling (2)

- Assume the length is known to be 3.
- bba 1

```
l = 5/9  r = 1
l = 1/9  r = 1
```

Coding with Scaling (3)

- Assume the length is known to be 3.
- bba 10

```
l = 1/9  r = 11/27
l = 2/9  r = 22/27
```

Coding with Scaling (4)

- Assume the length is known to be 3.
- bba 101

```
l = 2/9 = .000100101...
l = 2/9 = .000100001
(l+r)/2 = 14/27 = .100001001...
```
Notes on Arithmetic Coding

• Arithmetic codes come close to the entropy lower bound.
• Grouping symbols is effective for arithmetic coding.
• Arithmetic codes can be used effectively on small symbol sets. Advantage over Huffman.
• Context can be added so that more than one probability distribution can be used.
  – The best coders in the world use this method.
• There are very effective adaptive arithmetic coding methods.

Dictionary Coding

• Most popular methods are based on Ziv and Lempel's seminal work in 1977 and 1978.
• Basic idea: Maintain a dictionary of commonly used strings. Each commonly used string has an index.
  – Static dictionary, fixed and does not change.
  – Dynamic dictionary, adapts to the changing string.

Static Dictionary

<table>
<thead>
<tr>
<th>Number</th>
<th>Symbol</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>aa</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>ab</td>
<td>11</td>
</tr>
</tbody>
</table>

Encoding: from the current position find the longest string in source string that matches a string in the dictionary. Output its index.
Decoding: for each index output the corresponding string in the dictionary.

Static Dictionary Example

```
a a b c c a d b a a a d a     30 bits with 2 bits/symbol
a a
b c c
a
b
a a a a
b d a
```

4 7 8 9 10 11 11 11

28 bits at 4 bits/symbol

Dynamic Dictionary

• For a static dictionary both the encoder and decoder have to have the dictionary.
• Dynamic dictionary
  – The encoder builds the dictionary as it scans the input.
  – The decoder emulates the encoder, building the same dictionary as it decodes the string.

LZW Compression

• Invented by Ziv and Lempel in 1978 and improved upon by Welch in 1984.
• Unix compress and GIF are based on LZW
• In LZW both encoder and decoder share the same indexes of the symbol alphabet ahead of time.
  – For standard symbols sets like ASCII this is no problem.
LZW Encoding Algorithm

Repeat
find the longest match \( w \) in the dictionary
output the index of \( w \)
put \( wa \) in the dictionary where \( a \) was the
unmatched symbol

LZW Encoding Example (1)
Dictionary
\[
\begin{align*}
0 & : a \\
1 & : b \\
2 & : c \\
3 & : d \\
4 & : r \\
\end{align*}
\]
\text{abracadabra}

LZW Encoding Example (2)
Dictionary
\[
\begin{align*}
0 & : a \\
1 & : b \\
2 & : c \\
3 & : d \\
4 & : r \\
5 & : ab \\
\end{align*}
\]
\text{abracadabra}

LZW Encoding Example (3)
Dictionary
\[
\begin{align*}
0 & : a \\
1 & : b \\
2 & : c \\
3 & : d \\
4 & : r \\
5 & : ab \\
6 & : br \\
\end{align*}
\]
\text{abracadabra}

LZW Encoding Example (4)
Dictionary
\[
\begin{align*}
0 & : a \\
1 & : b \\
2 & : c \\
3 & : d \\
4 & : r \\
5 & : ab \\
6 & : br \\
7 & : ra \\
\end{align*}
\]
\text{abracadabra}

LZW Encoding Example (5)
Dictionary
\[
\begin{align*}
0 & : a \\
1 & : b \\
2 & : c \\
3 & : d \\
4 & : r \\
5 & : ab \\
6 & : br \\
7 & : ra \\
8 & : ac \\
\end{align*}
\]
\text{abracadabra}
LZW Encoding Example (6)

Dictionary
0   a 9   ca
1   b
2   c
3   d
4   r
5   ab
6   br
7   ra
8   ac

LZW Encoding Example (7)

Dictionary
0   a 9   ca
1   b 10  ad
2   c
3   d
4   r
5   ab
6   br
7   ra
8   ac

LZW Encoding Example (8)

Dictionary
0   a 9   ca
1   b 10  ad
2   c 11  da
3   d
4   r
5   ab
6   br
7   ra
8   ac

LZW Encoding Example (9)

Dictionary
0   a 9   ca
1   b 10  ad
2   c 11  da
3   d 12  aba
4   r
5   ab
6   br
7   ra
8   ac

LZW Encoding Example (10)

Dictionary
0   a 9   ca
1   b 10  ad
2   c 11  da
3   d 12  aba
4   r 13  ar
5   ab
6   br
7   ra
8   ac

LZW Encoding Example (11)

Dictionary
0   a 9   ca
1   b 10  ad
2   c 11  da
3   d 12  aba
4   r 13  ar
5   ab 14  rab
6   br
7   ra
8   ac
LZW Encoding Example (12)

Dictionary
0  a  9  ca  01402035076
1  b  10  ad
2  c  11  da
3  d  12  aba
4  r  13  ar
5  ab  14  rab
6  br  15  bra
7  ra
8  ac

LZW Encoding Example (13)

Dictionary
0  a  9  ca  014020350760
1  b  10  ad
2  c  11  da
3  d  12  aba
4  r  13  ar
5  ab  14  rab
6  br  15  bra
7  ra
8  ac

LZW Decoding Algorithm

- Emulate the Encoder in building the dictionary.
- Decode each index according to its index.
- Problem: the current index have an incomplete entry because it is currently being added to the dictionary.
  - The problem is solved because there is enough information in the incomplete entry to continue decoding.

LZW Decoding Example (1)

Dictionary
0  a  012436
1  b

LZW Decoding Example (2)

Dictionary
0  a  012436
1  b  a
2  a...

LZW Decoding Example (3)

Dictionary
0  a  012436
1  b  a b
2  ab
3  b...
LZW Decoding Example (4)

Dictionary
0  a
1  b
2  ab
3  ba
4  ab...

The next index is 4, but it is incomplete!

---

LZW Decoding Example (5)

Dictionary
0  a
1  b
2  ab
3  ba
4  aba

The entry has a first symbol which is all we need to complete it.

---

LZW Decoding Example (6)

Dictionary
0  a
1  b
2  ab
3  ba
4  aba
5  abab

---

LZW Decoding Example (7)

Dictionary
0  a
1  b
2  ab
3  ba
4  aba
5  abab
6  ba...

---

LZW Decoding Example (8)

Dictionary
0  a
1  b
2  ab
3  ba
4  aba
5  abab
6  bab
7  bab...

complete 6

---

LZW Decoding Example (9)

Dictionary
0  a
1  b
2  ab
3  ba
4  aba
5  abab
6  bab
7  bab...

0 1 2 3 6
a b ab aba

0 1 2 4 3 6
a b ab aba ba

0 1 2 4 3 6
a b ab aba ba bab
Trie Data Structure for Dictionary

- Fredkin (1960)

<table>
<thead>
<tr>
<th>0</th>
<th>a</th>
<th>9</th>
<th>ad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td>10</td>
<td>da</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>11</td>
<td>aba</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>12</td>
<td>ar</td>
</tr>
<tr>
<td>4</td>
<td>r</td>
<td>13</td>
<td>ra</td>
</tr>
<tr>
<td>5</td>
<td>ab</td>
<td>14</td>
<td>abr</td>
</tr>
<tr>
<td>6</td>
<td>br</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ac</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ca</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Depending on the size of the dictionary it might be wise to have two array levels to minimize searching.

Notes on Dictionary Coding

- Extremely effective when there are repeated patterns in the data that are widely spread. Where local context is not as significant.
  - text
  - some graphics
  - program sources or binaries
- Variants of LZW are pervasive.