Lecture 4: Adaptive source coding algorithms

February 1, 2019
Outline

1. Motivation;
2. adaptive Huffman encoding;
3. Gallager and Knuth’s method;
1. Motivation

Huffman/arithmetic encoding needs two passes

1. first pass to compute the source statistics

2. second pass: Huffman/arithmetic encoding

Moreover additional information is needed for the decoder

- either the statistics are known;
- or the encoding table is known.
Universal source coding

Universal source coding: no assumption on the source.

Idea: Compute at each time a dynamic source model that could produce the observed text and use this model to compress the text which has been observed so far.

Illustration:

- adaptive Huffman algorithm (memoryless source model),
- adaptive arithmetic coding algorithm,
- Lempel-Ziv algorithm and its variants (using the dependencies between the symbols).
2. Adaptive Huffman – Outline

Assume that \( n \) letters have been read so far, and that they correspond to \( K \) distinct letters.

- Let \( X_n \) be the source over \( K + 1 \) symbols formed by the \( K \) letters observed so far with probability proportional to their number of occurrences in the text + void symbol whose probability 0.

- Compute the Huffman code tree \( T_n \) of this source,

- the \( n + 1 \)-th letter is read and encoded
  - with its codeword when it exists,
  - with the \( K + 1 \)-th codeword + ascii code of the letter otherwise.
Adaptive Huffman – Coding

The initial Huffman tree has a single leaf corresponding to the void symbol. Each time a new letter $x$ is read

- if already seen
  - print its codeword,
  - update the Huffman tree,

- else
  - print the codeword of the void symbol followed by an unencoded version of $x$ (ascii code for instance),
  - add a leaf to the Huffman tree,
  - update the Huffman tree.
Adaptive Huffman – Decoding

The initial tree is formed by a single leaf corresponding to the void symbol. Until all the encoded text is read, perform a walk in the tree by going down left when '0' is read and going down right when '1' is read until a leaf is reached.

• if the leaf is not the void symbol
  – print the letter,
  – update the tree,

• else
  – print the 8 next bits to write the ascii code of the letter
  – add a leaf to the tree,
  – update the tree.
Adaptive Huffman – preliminary definitions

Prefix code of a d.s. \( X \): binary tree with \(|X|\) leaves = letters of \( X \).

Definition Consider a prefix code.

- the leaf weight is the probability of the corresponding letter
- the node weight is the sum of the weights of its children.

Definition A Gallager order \( u_1, \ldots, u_{2K-1} \) on the nodes of an irreducible code (of a source of cardinality \( K \)) verifies

1. the weights of \( u_i \) are decreasing,

2. \( u_{2i} \) and \( u_{2i+1} \) are brothers for all \( i \) such that \( 1 \leq i < K \).
Example

```
3 5 5
2 1
3 2
a
b d
c e f
11
1
23
5 4
7 5 6 6
10
98
2
```

Information Theory
Adaptive Huffman – Properties

Theorem 1. [Gallager] Let $T$ be a binary tree corresponding to a prefix code of a source $X$. $T$ is a Huffman tree of $X$ iff there exists a Gallager order on the nodes of $T$. 
Proof

$T$ Huffman $\implies T$ admits a Gallager order.

The two codewords of minimum weight are brothers $\Rightarrow$ remove them and keep only their common parent.

The obtained tree is a Huffman tree which admits a Gallager order (induction) $u'_1 \geq \cdots \geq u'_{2K-3}$.

The parent appears somewhere in this sequence. Take its two children and put them at the end. This gives a Gallager order.

$$u'_1 \geq \cdots \geq u'_{2K-3} \geq u_{2K-2} \geq u_{2K-1}$$
Proof

$T$ admits a Gallager order $\implies T$ Huffman.

$T$ has a Gallager order $\implies$ nodes are ordered as

$$u_1 \geq \cdots \geq u_{2K-3} \geq u_{2K-2} \geq u_{2K-1}$$

where $u_{2K-2}$ and $u_{2K-1}$ are brothers, leaves and are of minimum weight.

Let $T'$ be the tree corresponding to $u_1, \ldots, u_{2K-3}$. It has the Gallager order $u_1 \geq \cdots \geq u_{2K-3}$. It is a Huffman tree (induction).

By using Huffman’s algorithm, we know that the binary tree corresponding to $u_1, \ldots, u_{2K-1}$ is a Huffman tree, since once of its nodes is the merge of $u_{2K-2}$ and of $u_{2K-1}$. 
Proposition 1. Let $X_n$ be the source corresponding to the $n$-th step and let $T_n$ be the corresponding Huffman tree.

Let $x$ be the $n + 1$-th letter and let $u_1, \ldots, u_{2^K-1}$ be the Gallager order on the nodes of $T_n$.

If $x \in X_n$ and if all the nodes $u_{i_1}, u_{i_2}, \ldots, u_{i_\ell}$ that are on the path between the root and $x$ are the first ones in the Gallager order with this weight, then $T_n$ is a Huffman tree for $X_{n+1}$.

Proof Take the same Gallager order.
Adaptive Huffman – Updating the tree

Let $T_n$ be the Huffman tree at Step $n$ and let $u_1, \ldots, u_{2K-1}$ be its corresponding Gallager order.

Assumption: $x \in T_n$ (at node $u$).

repeat until $u$ is not the root

– let $\tilde{u}$ be the first node in the Gallager order of the same weight as $u$,
– exchange $u$ and $\tilde{u}$,
– exchange $u$ and $\tilde{u}$ in the Gallager order,
– Increment the weight of $u$ \textit{(weight = nb occurrences)}
– $u \leftarrow$ parent of $u$

This algorithm is due to D. Knuth.
Example

```
3
5 5
1110
2111
32
a
b d
c e f
11
1
23
5 4
7 5 6 6
10
98
3
```
Example
Example
Adaptive Huffman – Adding a Leaf

When the current letter \( x \) does not belong to the tree, the update uses the void symbol. It is replaced by a tree with two leaves, one for the void symbol and one for \( x \).

The two new nodes are added at the end of the sequence \((u_i)\).
Methods based on dictionaries

Idea: maintaining a dictionary (key,string).

The keys are written on the output, rather than the string.
The hope is that the keys are shorter than the strings.
Lempel-Ziv 78 – Outline

The Lempel-Ziv algorithm (1978) reads a text composed of symbols from an alphabet $\mathcal{A}$. Assume that $N$ symbols have been read and that a dictionary of the words which have been seen has been constructed.

- Read the text by starting from the $(N + 1)$-th symbol until a word of length $n$ which is not in the dictionary is found, print the index of the last seen word (it is of length $n - 1$) together with the last symbol.

- Add the new word (of length $n$) to the dictionary and start again at the $(N + n + 1)$-th symbol.
We need an efficient way of representing the dictionary. Useful property: when a word is in the dictionary all its prefixes are also in it.

⇒ the dictionaries that we want to represent are $|\mathcal{A}|$-ary trees. Such a representation gives a simple and efficient implementation for the functions which are needed, namely

- check if a word is in the tree,
- add a new word
Lempel-Ziv 78 – Coding

The dictionary is empty initially. Its size is $K = 1$ (empty word). Repeat the following by starting at the root until this is not possible anymore

- walk on the tree by reading the text letters until this is not possible anymore

  Let $b_1, \ldots, b_n, b_{n+1}$ be the read symbols and let $i$, $0 \leq i < K$ ($K$ being the size of the dictionary), be the index of the word $(b_1, \ldots, b_n)$ in the dictionary,

- $(b_1, \ldots, b_n, b_{n+1})$ is added to the dictionary with index $K$,

- print the binary representation of $i$ with $\lceil \log_2 K \rceil$ bits followed by the symbol $b_{n+1}$. 
### Lempel-Ziv 78 – Example

<table>
<thead>
<tr>
<th>Dictionary</th>
<th>pair (index,symbol)</th>
<th>codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\varepsilon$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(0,1)</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>(0,0)</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>(2,1)</td>
</tr>
<tr>
<td>4</td>
<td>011</td>
<td>(3,1)</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>(1,0)</td>
</tr>
<tr>
<td>6</td>
<td>00</td>
<td>(2,0)</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>(1,1)</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>(5,0)</td>
</tr>
</tbody>
</table>

**Information Theory**
Lempel-Ziv 78 – Decoding

The dictionary is now a table which contains only the empty word $M_0 = \emptyset$ and $K = 1$. Repeat until the whole encoded text is read

- read the $\lceil \log_2 K \rceil$ first bits of the encoded text to obtain index $i$. Let $M_i$ be the word of index $i$ in the dictionary

- read the next symbol $b$,

- add a $K$-th entry to the table $M_K \leftarrow M_i \parallel b$,

- print $M_K$. 

The Welsh variant

• Initially, all words of length 1 are in the dictionary.

• Instead of printing the pair \((i, b)\) print only \(i\).

• Add \((i, b)\) to the dictionary.

• Start reading again from symbol \(b\).

⇒ slightly more efficient.

used in the unix compress command, or for GIF87.

In practice, English text is compressed by a factor of 2.
### Lempel-Ziv-Welsh – Example

```
1 0 0 1 0 1 1 1 0 0 0 1 1 1 0 0 1 0 1 . . .
```

<table>
<thead>
<tr>
<th>indices</th>
<th>words</th>
<th>word</th>
<th>index</th>
<th>codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>10</td>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>7</td>
<td>110</td>
<td>11</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>8</td>
<td>000</td>
<td>00</td>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>01</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>10</td>
<td>1100</td>
<td>110</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>01</td>
<td>5</td>
<td>0101</td>
</tr>
</tbody>
</table>

Information Theory
Lempel-Ziv-Welsh, encoding

1. Read the text until finding $m$ followed by a letter $a$ such that $m$ is in the dictionary, but not $m||a$,

2. print the index of $m$ in the dictionary,

3. add $m||a$ to the dictionary,

4. Continue by starting from letter $a$. 
Lempel-Ziv-Welsh, decoding

1. read index $i$,

2. print the word $m$ of index $i$,

3. add at the end of the dictionary $m||a$ where $a$ is the first letter of the following word ($a$ will be known the next time a word is formed).
This variant appeared before the previous one. More difficult to implement.

Assume that $N$ bits have been read.

Read by starting from the $N + 1$-th bit the longest word (of length $n$) which is in the previously read text (=dictionary) and print $(i, n, b)$, where $b$ is the next bit.

In practice, implemented by a sliding window of fixed size, the dictionary is the set of words in this sliding window.

Used in gzip, zip.
Example

\[ (0,0,0) \]

0 1 0 1 1 1 0 0 0 1 1 1 0 0 1
Example

\[
\begin{array}{c}
0 \\
(0,0,0)
\end{array}
\quad
\begin{array}{c}
0 1 \\
(1,1,1)
\end{array}
\quad
0 1 1 1 0 0 0 1 1 1 0 0 1
\]
### Example

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(0,0,0)</td>
<td>(1,1,1)</td>
<td>(2,2,1)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c c c c c c c c}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
\end{array}
\]
Example

\[
\begin{array}{cccc}
0 & 01 & 011 & 100 \\
(0,0,0) & (1,1,1) & (2,2,1) & (3,2,0) \\
0 & 1 & 1 & 1 & 0 & 0 & 1
\end{array}
\]
Example

\[
\begin{array}{cc|cc|cc|cc}
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
(0,0,0) & (1,1,1) & (2,2,1) & (3,2,0) & (4,6,1)
\end{array}
\]
**Stationary Source**

**Definition** A source is *stationary* if its behavior does not change with a time shift. For all nonnegative integers $n$ and $j$, and for all $(x_1, \ldots, x_n) \in \mathcal{X}^n$

$$p_{X_1\ldots X_n}(x_1, \ldots, x_n) = p_{X_1+j\ldots X_n+j}(x_1, \ldots, x_n)$$

**Theorem 2.** For all stationary sources the following limits exist and are equal

$$H(\mathcal{X}) = \lim_{n \to \infty} \frac{1}{n} H(X_1, \ldots, X_n) = \lim_{n \to \infty} H(X_n \mid X_1, \ldots, X_{n-1}).$$

*the quantity $H(\mathcal{X})$ is called the entropy per symbol.*
The fundamental property of Lempel-Ziv

**Theorem 3.** For all stationary and ergodic sources $\mathcal{X}$ the compression rate goes to $H(\mathcal{X})$ with prob. 1 when the text size goes to $\infty$. 