Introduction to information theory and coding

Louis WEHENKEL

Set of slides No 5

State of the art in data compression

- Stochastic processes and models for information sources
- First Shannon theorem : data compression limit
- Overview of state of the art in data compression
- Relations between data mining and data compression

Optimal prefix codes - Huffman Algorithm

Note : illustrations in the binary case.

Let n be the length of the longest codeword (q-ary code).

Complete q-ary tree of depth n : acyclic graph built recursively starting at the root (cf figure).

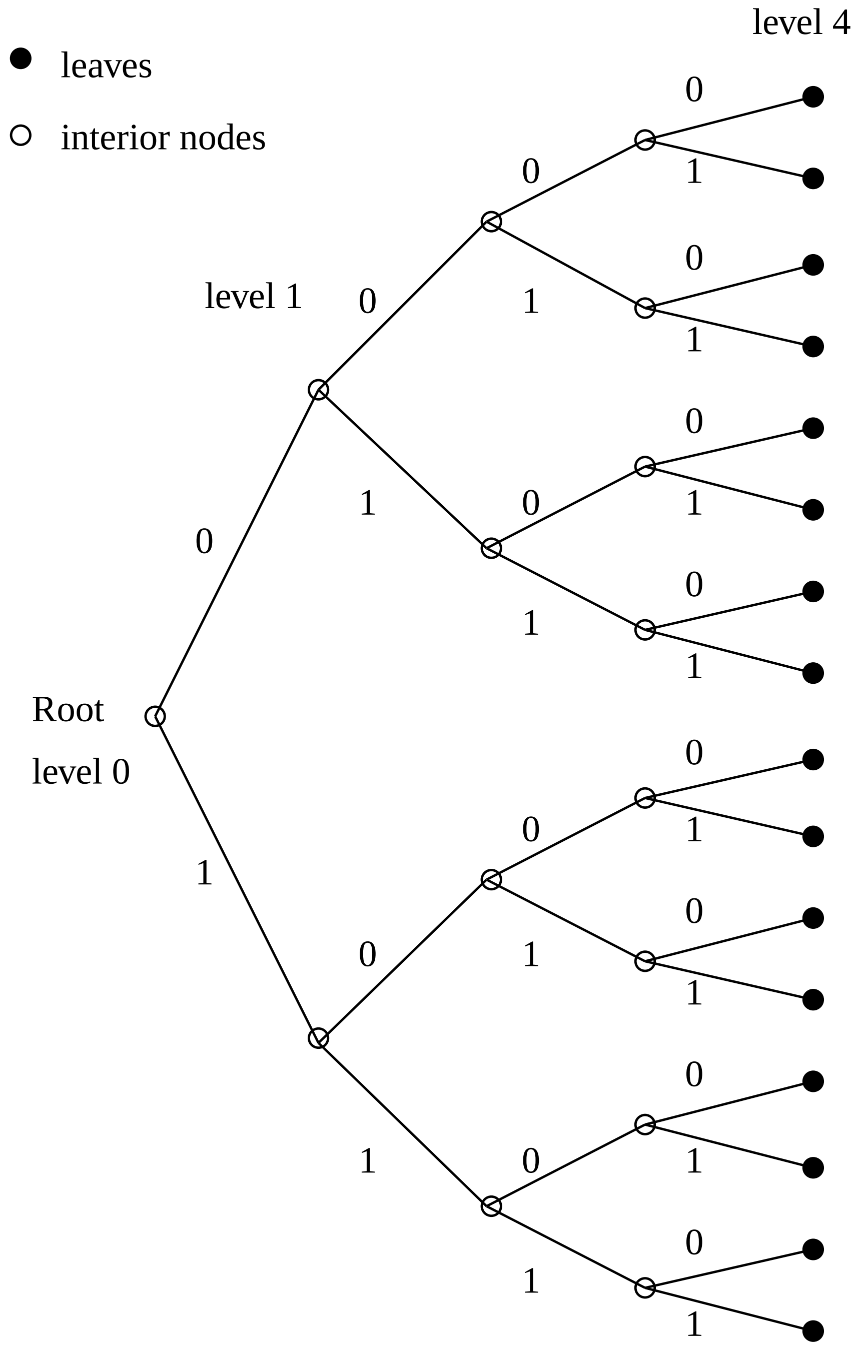
Oriented arcs labeled by the q code symbols.

Father, son, siblings (brothers), descendants, ascendants...

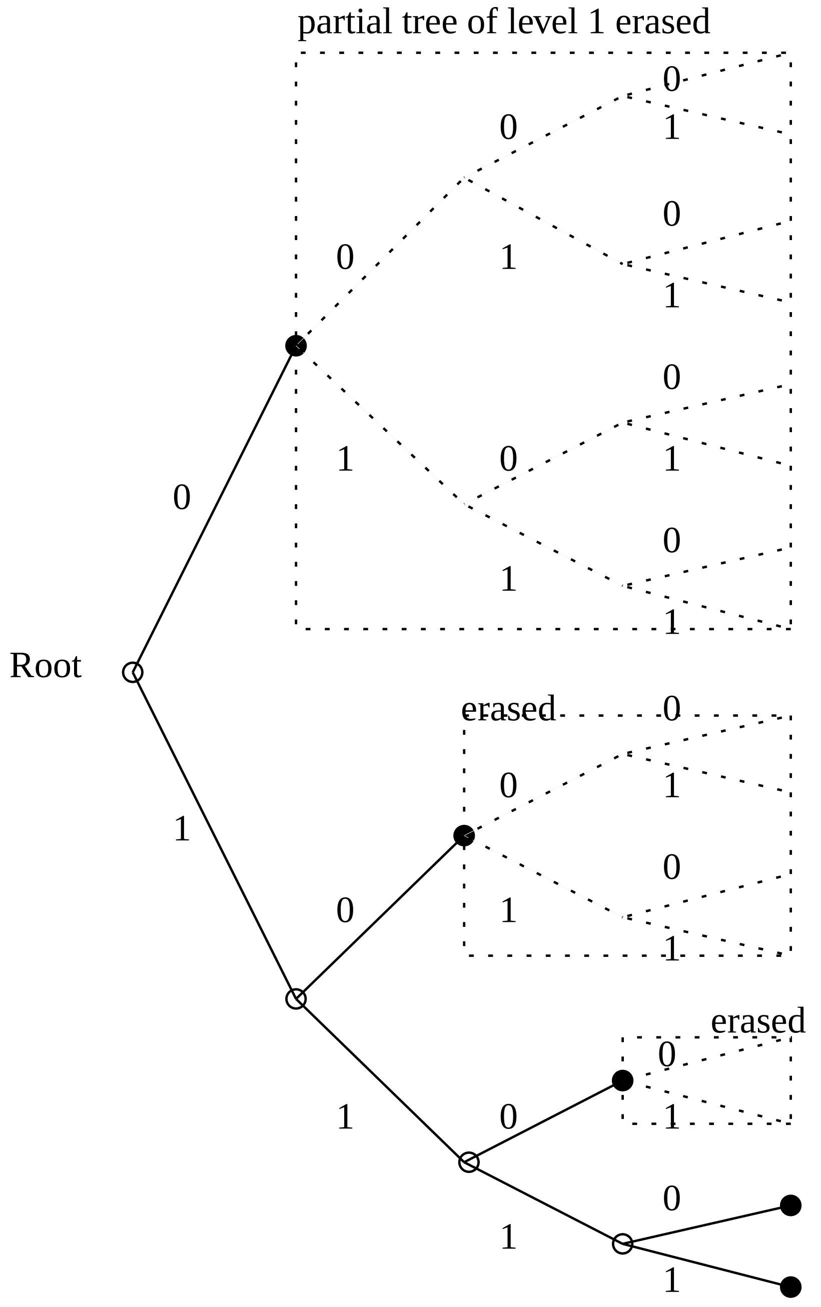
Interior vs terminal nodes (leaves)

Path : sequence of arcs (u_{i_1}, u_{i_2}) where $u_{i_1} = u_{(i-1)_2}$.

Levels : root = level 0,



(a) complete tree



(b) incomplete tree

Incomplete trees

Complete tree of which some complete subtrees have been “erased”

$\exists$ trees which are neither complete nor incomplete

But “tree is complete or incomplete” $\Leftrightarrow$

$$A = (L-1)\frac{q}{q-1}.$$

Relations between trees and codes

The paths towards the leaves of complete q-ary tree of depth n are in one-to-one relation with sequences of length n built from a q-ary alphabet.

Each and every instantaneous (prefix-free) code may be represented by a complete or incomplete tree, and reciprocally, every such tree defines a prefix-free code.

(Nb. Some leaves may possibly not be labeled by a codeword)

Proof :

Let assume we have a prefix-free code of word-lengths n_i.

Let

$$n = \max_{i=1,\ldots,Q} \{n_i\}.$$

We start with a complete tree of depth n :

All codewords necessarily correspond to paths in the tree starting at the root and ending somewhere in the complete tree (at a leave or maybe at some interior node).

All these paths are different, since the code is necessarily regular.

The complete tree initially has a total of q^n leaves.

Let us construct the uncomplete tree corresponding to the code by pruning away some of its (complete) subtrees.

How ?

We merely insert all the codewords and mark the end-nodes of the corresponding paths.

Now, let m_i be one of these codewords : we erase all successor nodes of the last node in the corresponding path.

By doing this, we will not delete any of the marked nodes. Why ?

In addition, the number of leaves of the complete tree that we remove in this process is equal to q^{n-n_i}.

Thus, we remove like this a total of $\sum_{i=1}^{Q} q^{n-n_i}$ leaves of the original complete tree, by iterating the deletion process through all codewords, without removing any of the Q nodes originally marked as a codeword.

Consequence : Since the original tree has q^n leaves, and since (obviously) this process works, we deduce that

$$\sum_{i=1}^{Q} q^{n-n_i} \leq q^n$$

$\Rightarrow$ prefix-free code must satisfy the Kraft inequality.

Now let us show that if Kraft is true there exists a uniquely decodable code with the given word length :

If the n_i verify Kraft inequality, then $\exists$ an instantaneous code (hence uniquely decodable) based on these lengths.

Let us start with $\sum_{i=1}^{n} r_i q^{n-i} \leq q^n : \Rightarrow \forall p \leq n : \sum_{i=1}^{p} r_i q^{p-i} \leq q^p$.

Why is it true ? (r_i is the number of words of length i)

(Multiply Kraft by q^{p-n}, and truncate the sum after the p-th term.)

Now let us proove that we can build a prefix-free code with the original word lengths (r_i words of length i, $\forall i$).

1. r_1 words of length 1 : is it possible ?

2. r_2 words of length 2 : is it still possible ? We must have $(q - r_1)q \leq r_2$ (?)

3. r_p words of length p : suppose we have already chosen the $p-1$ first groups of words, and then verify that there is still enough room to insert the r_p words of length p.

Questions.

If I give you the word lengths of an instantaneous code, could you build any such a code compatible ?

Under which condition can we complete the code with more words ?

If we want to add words of minimal length :

How many words of length $\leq n$ can we certainly add ?

What is the meaning of the condition $\sum_{i=1}^{n} r_i q^{-i} = 1$?

And in terms of code tree ?

Given source symbol probabilities $P(S_i)$, how to build an optimal code ?

NB: optimal = average word length minimal.

Let us explore the problem in the particular case where $q = 2$ (binary code)

A first idea to construct Q prefix free codewords :

Start with a complete tree of depth $n = \lceil \log_2 Q \rceil$.

NB : if $Q = 2^n$ and $P(S_i)$ uniform, it is not necessary to work further (Why ?).

Otherwise : if $Q < 2^n$ we delete $2^n - Q$ subtrees of depth $n - 1$.

Is it possible ?

$\Rightarrow$ **We have an algorithm to build a first, not necessarily optimal, code-tree**

Idea : try to modify the tree, until it becomes an optimal tree.

Search operator : exchange subtrees in such a way that the average word length always decreases, and detect when optimality has been reached.

Optimality : is $\overline{n} = \sum_{i=1}^{Q} n_i P(S_i)$ minimal ?

How to recognize an optimal tree ?

How to improve the tree in order to reach optimality ?

Reasoning tool : node probabilities

We decorate the leaves of the tree with the source symbol probabilities.

We propagate this information upwards towards the root : a node receives the sum of the probabilities of its sons.

The recursive structure of the code and of the tree, implies that if a codetree is optimal, than all its subtrees are also optimal (with respect to sub-source alphabets).

In particular, the partial (pending) trees must be optimal for the subset of source symbols. Why ?

And also, the reduced trees that we would get by deleting some of the subtrees, must be optimal with respect to the probabilities attached at the corresponding nodes. Why ?

But there is more :

An optimal tree must also respect a *non-local* condition which implies pairs of partial trees of different levels :

If T_1 and T_2 are two partial trees of different levels (levels of their root) and of different probabilities, then the most probable of the two must be the least deep one.

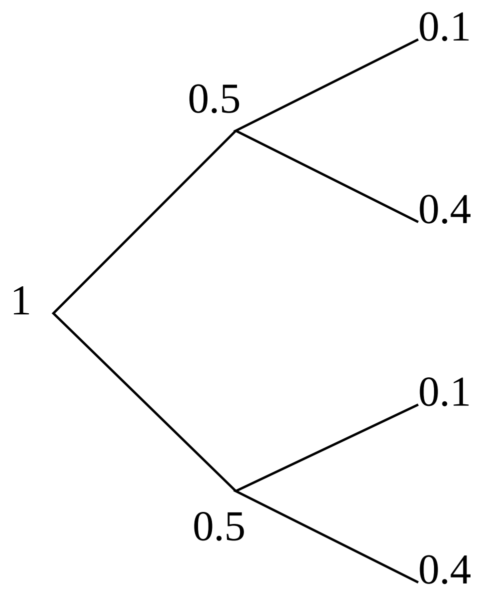
Indeed, otherwise we could swap the two subtrees and thereby improve (reduce) average wordlength. By how much ?

For example if $n_1 < n_2$ and $p_1 < p_2$: $\Delta\overline{n} = (n_2 - n_1)(p_2 - p_1)$.

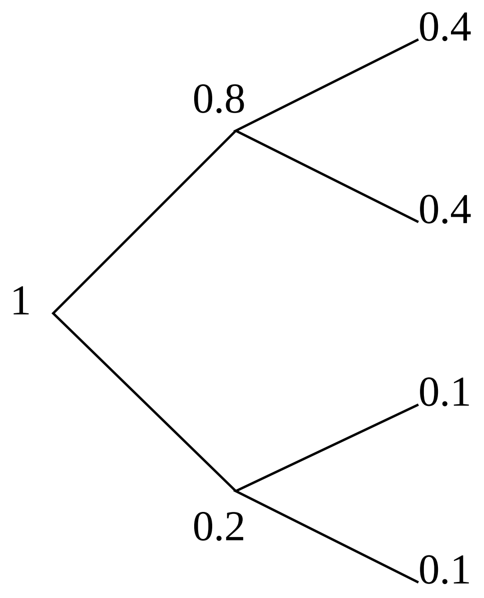
This criterion provides a nice criterion to improve our codetree!

We merely need to localise subtrees of different levels and different probabilities which violate the condition, and swap them to improve our code.

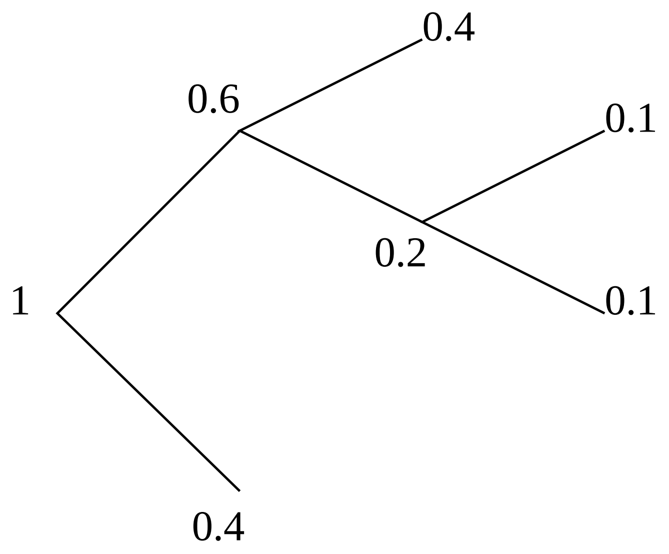
Illustration(s)



(a) Average length : 2



(b) Average length : 2



(c) Average length : 1.8

Can we exchange the partial trees which should be exchanged while improving (strictly) the average codelength ?

Do we have an algorithm ?

Yes, but it doesn't work... (in all cases)

Conclusion :

We need to impose one more (at least) constraint on our codetree, in order to be optimal.

Actually, we can (easily) prove the following :

For any source probability distribution, there exists an optimal prefix-free code that satisfies the following properties :

1. If $p_j > p_i$, then $n_j \leq n_i$.

2. The two longest codewords have the same length

3. The two longest codewords differ only in the last bit and correspond to the least likely symbols.

Summary : if $p_1 \geq p_2 \geq \ldots \geq p_Q$, then there exists an optimal (binary code) with length satisfying $n_1 \leq n_2 \leq \ldots \leq n_{Q-1} = n_Q$, and codewords m_{Q-1} and m_Q differing only in the last bit.

$\Rightarrow$ **We can restrict our search in the class of codes which satisfy these properties.**

Huffman Algorithm

Who has guessed ?

Who does remember ?

$$\begin{tikzcd}
S_1 & 0.1 \arrow[r] & 0.2 \arrow[r] & 0.6 \arrow[r] & 1 \\
S_2 & 0.1 \arrow[ur] & 0.4 \arrow[ur] & 0.4 \arrow[ur] & \\
S_3 & 0.4 \arrow[ur] & 0.4 \arrow[ur] & & \\
S_4 & 0.4 \arrow[ur] & & &
\end{tikzcd}$$

(a) Code construction

$$\begin{tikzcd}
S_1 & 0.1 \arrow[r, dash] & 0.2 \arrow[r, dash] & 0.6 \arrow[r, dash] & 1 \\
S_2 & 0.1 \arrow[ur, dash] & 0.4 \arrow[ur, dash] & 0.4 \arrow[ur, dash] & \\
S_3 & 0.4 \arrow[ur, dotted] & 0.4 \arrow[ur, dotted] & & \\
S_4 & 0.4 \arrow[ur, dotted] & & &
\end{tikzcd}$$

(b) Building of code-tree

This produces an optimal prefix-less code (not unique in general).

Synthesis.

Huffman produces an optimal code tree and prefix-free code, such that

$$\frac{H(S)}{\log q} \leq \overline{n} < \frac{H(S)}{\log q} + 1$$

Absolutely optimal code : If $\overline{n} = \frac{H(S)}{\log q}$

Iff $n_i = -\log_q P(S_i), \forall i \Leftrightarrow P(S_i) = q^{-n_i}, \forall i.$

Reciprocally :

For every set of word lengths n_i which respect the Kraft equality, there exists a source probability distribution q_i such that the optimal code has these word lengths and is absolutely optimal.

What if $p_i \neq q_i$? One can show that in this case (binary code)

$$\overline{n} = H_Q(p_1, \ldots, p_Q) + D((p_1, \ldots, p_Q) \| (q_1, \ldots, q_Q)).$$

What if q-ary code ?

Data compression algorithms

1. Reversible text compression
    - Zero order methods $\rightarrow$ arithmetic coding
    - Higher order methods
    - Adaptative methods
    - Dictionary methods
2. Image compression
    - Multi-dimensional information structures
    - Sources of redundancy
    - Image transform based methods

1. Reversible text compression

Let us assume binary input and output alphabets.

T : input text (sequence of bits)

$U = C(T)$: coded text (sequence of bits)

$\ell(\cdot)$: length

Compression rate of C on text T : $\frac{\ell(T)}{\ell(C(T))}$

Realized rate : average of texts T.

Preliminary stage :

Choice of a source alphabet $\rightarrow$ segmentation of text into a sequence of words

$S = \{s_1, \ldots, s_m\}$

Parsing of T : $T \approx s_{i1} \cdots s_{i_t}$

Zero order methods

Intuitively : we take into account only the frequencies of the s_i.

(Higher order : we take also into account correlations among successive symbols)

Non-adaptative : code independent of the position in the text.

(Adaptative : code evolves as the text is screened)

NB. Text screened by increasing order of indexes

$\Rightarrow$ One-dimensional (oriented) structure (time : from left to right)

1. Replacement schemes

Idea : replace each $s_i \rightarrow w_i$ (Shannon, Fano, Huffman)

2. Arithmetic codes

Remplace the whole text : $T \rightarrow r \in [0, 1[$.

Example : $T = 111110111111011101111101110110$

Let's suppose that the s_i are chosen as follows

$$\begin{aligned} s_1 &= 0 \\ s_2 &= 10 \\ s_3 &= 110 \\ s_4 &= 1110 \\ s_5 &= 1111, \end{aligned} \tag{1}$$

$\Rightarrow$ parsing of T gives : $T = s_5 s_2 s_5 s_4 s_4 s_5 s_1 s_4 s_3$.

Let the code be $C\ : s_i \rightarrow w_i$

$$\begin{aligned} s_1 &\rightarrow 1111 \\ s_2 &\rightarrow 1110 \\ s_3 &\rightarrow 110 \\ s_4 &\rightarrow 10 \\ s_5 &\rightarrow 0. \end{aligned} \tag{2}$$

$\Rightarrow C(T) = 01110010100111110110.$

Thus $\ell(C(T)) = 20$ (et $\ell(T) = 30$) $\Rightarrow$ compression rate of $3/2$.

Questions

How to chose the s_i ? (source alphabet)

How to chose the w_i ? (code)

NB: both influence the compression rate.

E.g. : if $S = \{0, 1\}$, replacement scheme always gives a compression rate ≤ 1.

1. Choice of S

Which types of S make sense ?

We restrict our choice to the sets which are sufficiently rich to parse any sequence of bits, and at the same time sufficiently small to do the parsing in a single way (no ambiguity).

This will mean that for such an S, every binary text may be parsed in a single way into a sequence of symbols in S.

Definition : SPP.

$S = \{s_1, s_2, \ldots, s_m\}$ (binary words) verify the *strong parsing property* SPP)
$\Leftrightarrow$ every binary text is representable in the form of a *unique* concatenation,

$$T = s_{i_1} \cdots s_{i_t} \nu, \tag{3}$$

of some of the s_i and a suffix ν (possibly empty), such that none of the s_i is a prefix of ν, and $\ell(\nu) < \max_{1 \le i \le m} \ell(s_i)$.

ν = *leaf* of the parsing of T by the s_i.

Uniqueness : If

$$T = s_{i_1} \cdots s_{i_t} \nu = s_{j_1} \cdots s_{j_r} \mu$$

with ν and μ having none of the s_i as prefix and $\ell(\nu), \ell(\mu) < \max_{1 \le i \le m} \ell(s_i)$, then $t = r$, $i_1 = j_1, \ldots, i_t = i_r$, and $\nu = \mu$.

Necessary and sufficient condition for SPP : no prefix + Kraft <u>equality</u>

$$\sum_{s_i \in S} 2^{-\ell(s_i)} = 1. \tag{4}$$

Avantage prefix-less : efficient and on-line parsing

Avantage completeness : works for any text

Examples :

1. $S = \{0,1\}^L$: fixed block lengths

E.g. $L = 8$ computer files (cf bytes).

Interest : natural redundancy (8 bits for 60 ASCII characters)

$\Rightarrow$ free compression : $m = 60$.

2. Complete prefix-free codes (cf. codetrees).

NB: we are going to neglect the parsing leaf in what follows...

How to choose the s_i ? Standard solution : $S = \{0,1\}^8$, but...

2. Choice of the (data compression) code

T given in binary, then parsed into Z using a source alphabet S (SPP) :

$$Z = s_{i_1} \cdots s_{i_n} \qquad \text{(+ possibly } \nu\text{)}$$

Choice of a w_i for each s_i, suchthat $\ell(U) = \ell(w_{i_1} \cdots w_{i_n})$ is minimal

$$\ell(U) = \sum_{j=1}^{n} \ell(w_{i_j}) = n \sum_{i=1}^{m} f_i \ell(w_i). \tag{5}$$

NB: mathematically same problem than source coding ($p_i \rightarrow f_i$).

Conclusion : same solutions applicable, and same limitations (Shannon).

Optimal solution : Huffman using the f_i.

NB:
If f_i change from text to text
$\rightarrow$ code changes $\rightarrow$ must transmit the f_i or the code $\rightarrow$ overhead.
Or we take a fixed source model.

Binary expansion of a number $r \in [0,1[$

$$r = \lim_{n\to\infty} \sum_{j=1}^{n} a_j 2^{-j}, \qquad a_j \in \{0,1\} \tag{6}$$

The n first bits : $\rightarrow$ a word $a_1a_2\cdots a_n$.

Notation : $0.a_1a_2\cdots a_n = \sum_{j=1}^{n} a_j 2^{-j}$.

Dyadic fraction, if $\exists$ 'exact' finite expansion.

Dyadic fraction $\Rightarrow$ two binary representations :

$$0.a_1a_2\cdots a_{n-1}1 = 0.a_1a_2\cdots a_{n-1}011111\ldots$$

Convention

If dyadic : we use finite expansion

Otherwise : $\exists$ 1 single expansion (infinite).

Shannon code

We have already seen the word lengths (proof of first Shannon theorem), but not the method invented by Shannon to build the prefix-free code using these word-lengths.

Let the $s_1, \ldots, s_m$ be sorted suchthat $f_1 \geq f_2 \geq \cdots \geq f_m > 0$

(we can remove those s_j which do not appear at all after parsing text T.)

Let $F_1 = 0$ and $F_k = \sum_{i=1}^{k-1} f_i, 2 \leq k \leq m$ and denote by $\ell_k = \lceil \log_2 f_k^{-1} \rceil$.

The Shannon code $s_i \rightarrow w_i$ consists in using for w_i the ℓ_i first bits of the binary expansion of F_i.

Question : is this code prefix-free ?

Convince yourself...

Average length:

$$\overline{\ell}_{\text{Shannon}} \leq H_m(f_1, \ldots, f_m) + 1. \tag{7}$$

Example : $s_5 s_2 s_5 s_4 s_4 s_5 s_1 s_4 s_3 \rightarrow f_1 = f_2 = f_3 = 1/9, f_4 = f_5 = 3/9$

Sort : $s'_i = s_{5-i+1} \rightarrow \ell'_1 = \ell'_2 = 2$ and $\ell'_3 = \ell'_4 = \ell'_5 = 4$

$$\begin{array}{rcl} F'_1 = & 0 & = (.00\ldots) \\ F'_2 = & 3/9 & = (.01\ldots) \\ F'_3 = & 6/9 & = (.1010\ldots) \\ F'_4 = & 7/9 & = (.1100\ldots) \\ F'_5 = & 8/9 & = (.1110\ldots) \end{array} \tag{8}$$

Shannon code

$$\begin{array}{l} s_5 = s'_1 \rightarrow 00 \\ s_4 = s'_2 \rightarrow 01 \\ s_3 = s'_3 \rightarrow 1010 \\ s_2 = s'_4 \rightarrow 1100 \\ s_1 = s'_5 \rightarrow 1110 \end{array} \tag{9}$$

Average length is $8/3 = 2.666$, to compare with $H = 2.113$.

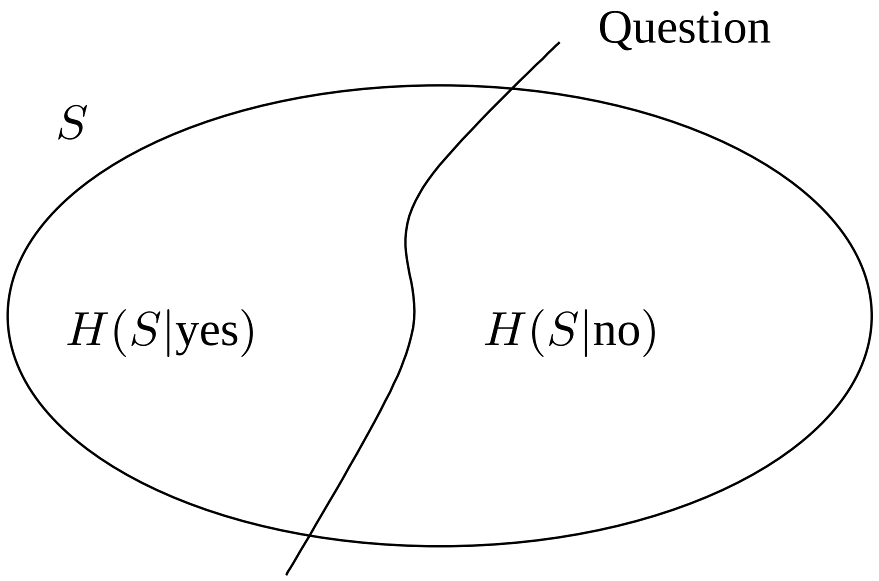
Compression rate $5/4$ (24 bits to represent the text U).

Fano code (philosophy)

Construction of a codetree in a “top-down” fashion.

Strategy is similar to decision tree building techniques.

We start with source entropy $H(S)$ (given) : how to divide the set of symbols so as to minimize average conditional entropy ?



$$H(S) = H(S,Q) = H(S|Q) + H(Q)$$

$\Rightarrow$ maximize $H(Q)$
$\Rightarrow$ equilibrate probabilities

Code : decision tree

Fano code (algorithm)

- f_i and s_i sorted by decreasing order of the f_i.
- we split according to this order, so as to maximize $H(Q)$:

$$\Rightarrow \sum_{i=1}^{k} f_i \text{ and } \sum_{i=k+1}^{m} f_i \text{ as close as possible}$$

- one proceeds recursively with each subset $\rightarrow$ singletons

Example :

$$\begin{array}{rcccccl}
s_1 & = & 3/9 & 0 & 0 & & \rightarrow 00 \\ \cline{5-7}
s_2 & = & 3/9 & 0 & 1 & & \rightarrow 01 \\ \cline{4-7}
s_3 & = & 1/9 & 1 & 0 & & \rightarrow 10 \\ \cline{5-7}
s_4 & = & 1/9 & 1 & 1 & 0 & \rightarrow 110 \\ \cline{6-7}
s_5 & = & 1/9 & 1 & 1 & 1 & \rightarrow 111.
\end{array} \tag{10}$$

Average length $20/9 = 2.222$.

In general :

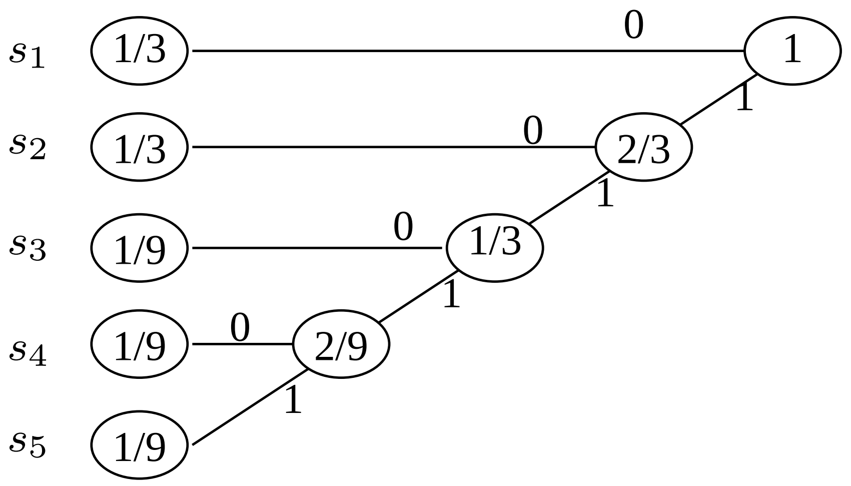
One can show that : $\overline{\ell}_{\text{Fano}} \leq H_m(f_1, \ldots, f_m) + 2$.

NB: if for each question $H(Q) = 1$, then $\overline{\ell}_{\text{Fano}} = H_m(f_1, \ldots, f_m)$

Huffman code

Bottom-up construction of the codetree $\Rightarrow$ optimal.

Example



$s_1 \rightarrow 0$

$s_1 \rightarrow 10$

$s_1 \rightarrow 110$

$s_1 \rightarrow 1110$

$s_1 \rightarrow 1111$

Average length $20/9 = 2.222$.

We have : $\overline{\ell}_{\text{Huffman}} \leq H_m(f_1, \ldots, f_m) + 1$.

Summary (symbol codes) :

What about the choice of the source alphabet s_i ?

What about the f_i ?

Given by a source model

Or estimated from each given text :

- necessity to transmit code (what is the overhead ?)
- necessity to adopt some conventions in code construction algorithms
- is not an on-line method.

Optimality :

$$\boxed{\bar{\ell}_{\text{Huffman}} \leq \min\{\bar{\ell}_{\text{Shannon}}, \bar{\ell}_{\text{Fano}}\}} \quad \text{but} \quad \boxed{\bar{\ell}_{\text{Shannon}} ? \gtrless ? \bar{\ell}_{\text{Fano}}}$$

What if the $p_i \neq f_i$?

What if we consider the extended source ?

Arithmetic coding (the Rolls)

Idea (stream code)

For a given source text length N.

We associate to each possible text a sub-interval of $[0,1[$ (they don't overlap)

Sub-interval defined by a probabilistic model of the source.

$C(T) = r \in$ sub-interval, represented in binary with just enough bits to avoid confusion among different numbers corresponding to different texts.

Small sub-intervals = unlikely texts :

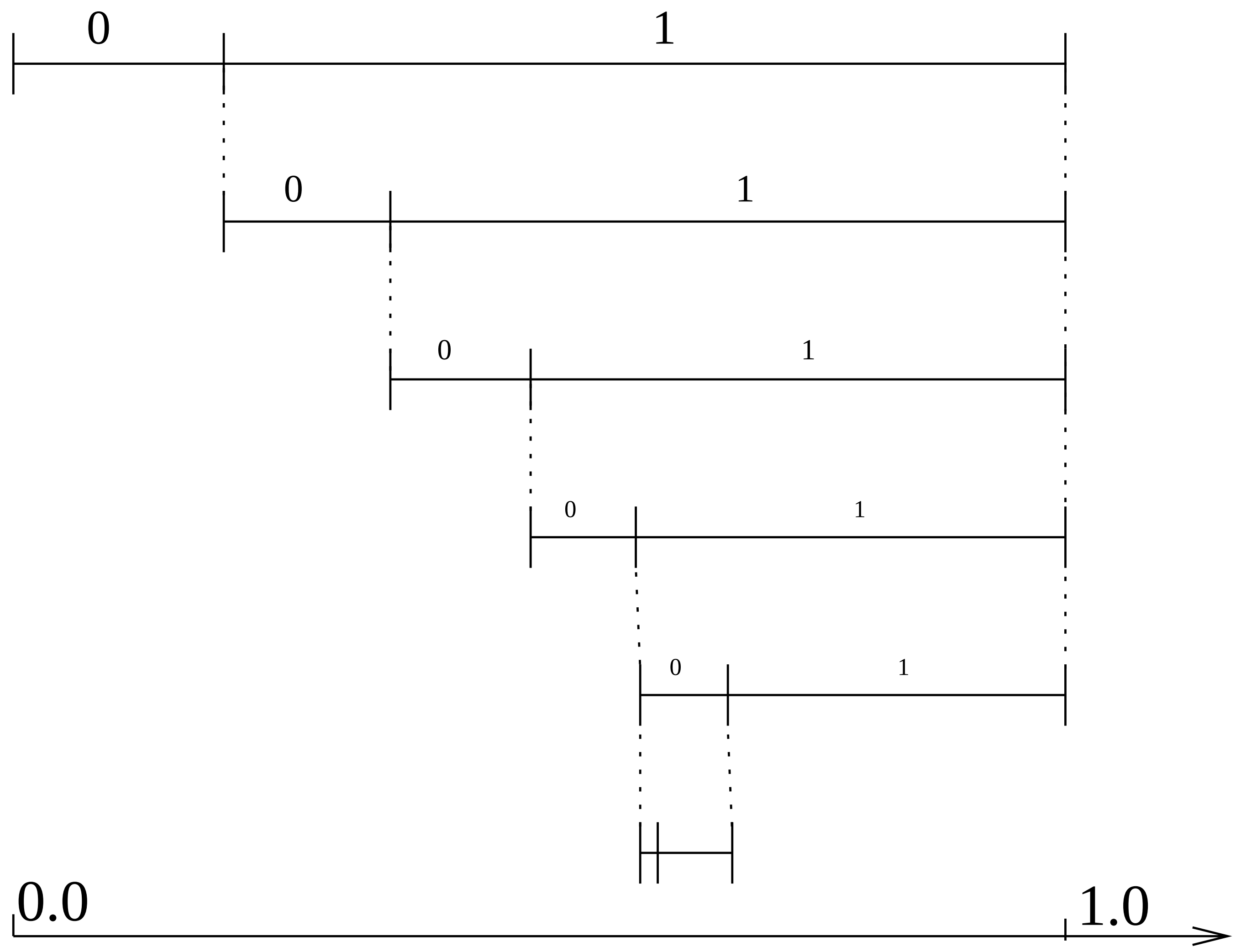
$\Rightarrow$ unlikely texts : need many bits to specify r.

Sub-interval of a text is included in the sub-interval of any prefix of this text :

$\Rightarrow$ recursive (and on-line) construction

Illustration

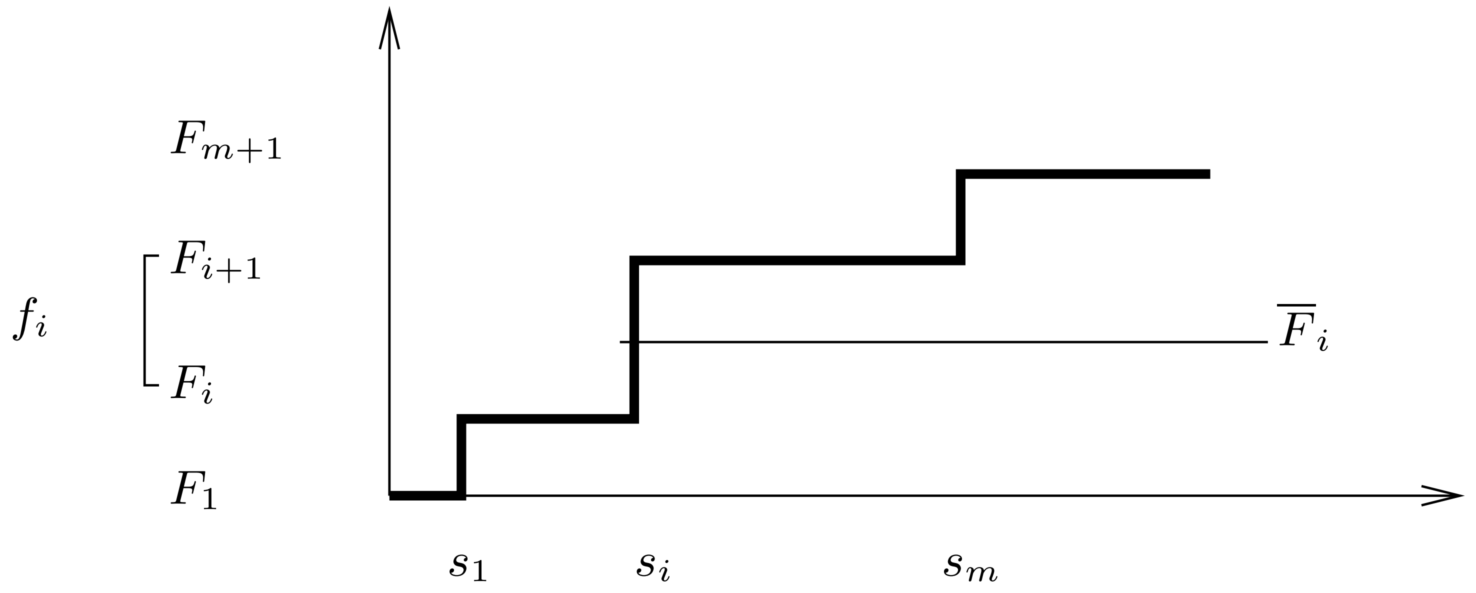
$T = 111110111111011101111101110110$ thus $f(0) = 6/30 = 0.2$, $f(1) = 0.8$



How does it work ?

Shannon-Fano-Elias $\rightarrow$ starting point

Symbol code : cumulative symbol frequency diagram (NB: symbols are in arbitrary order.)



$$\overline{F}_i = F_i + \frac{1}{2} f_i.$$

$\lfloor r \rfloor_\ell$: keep the ℓ first bits of the binary expansion of r.

Let us take for $r = \overline{F}_i$ and $\ell_i = \lceil -\log f_i \rceil + 1$.

One can check that : $0.w_i = \lfloor \overline{F}_i \rfloor_{\ell_i}$ in $]F_i, F_{i+1}[$.

One can also check that code $s_i \rightarrow w_i$ is prefix-free.

Average length : $\overline{\ell} < H_m(f_1, \ldots, f_m) + 2$.

NB: we pay for 1 bit because of the prefix condition, which is imposed by the nature of a symbol code (can be dropped for a stream code).

By itself not very efficient, but the idea is at the basis of arithmetic coding.

If, instead of coding source symbols we code blocs of source symbols : same idea still works but the overhead of the rounding and prefix bits become less dramatic.

If we code the whole text (Mega-Block) : no need for the prefix condition : we can assume $\ell(T) = \lceil -\log f(T) \rceil$.

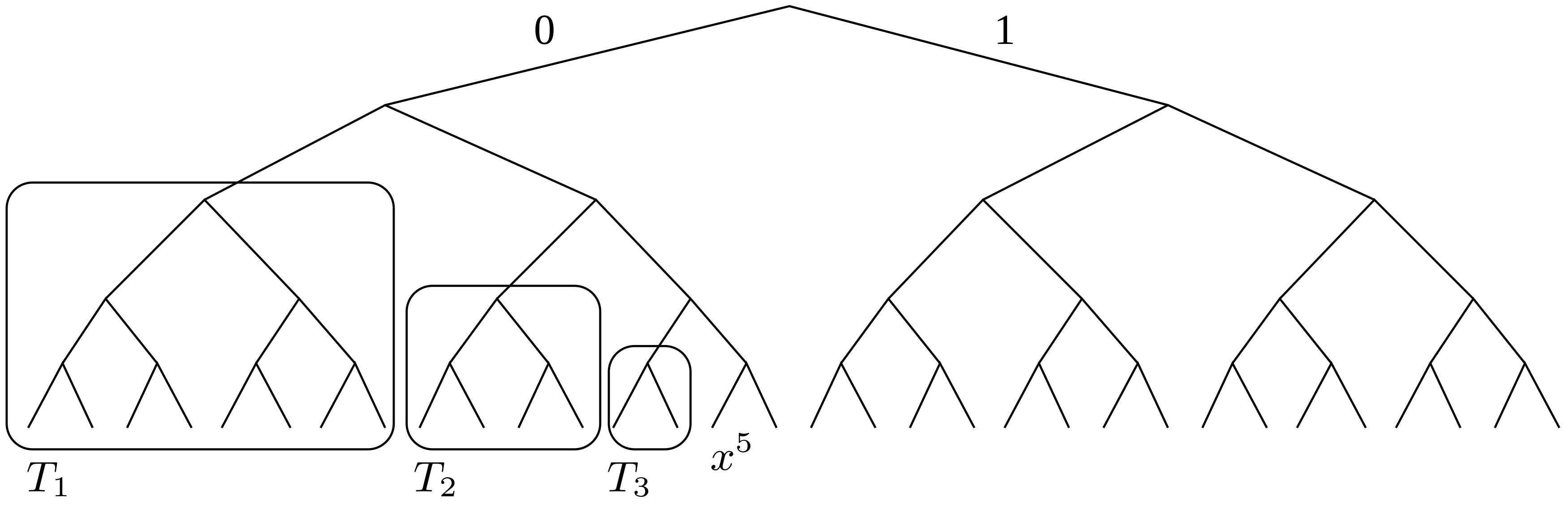
$\Rightarrow$ **Arithmetic code**

How to encode and decode efficiently

NB. For a long text, explicit method doesn't work.

Let n be the length of the text T.

We use a tree of depth n to represent (implicitly) all possible texts of length n.



The leaves sorted from left to right correspond to the lexicographic order of all possible texts.

Let $n = 5$ and let's suppose that x^5 represents our text.

We must determine $F(x^5)$ and $f(x^5)$ to encode the text.

$f(x^n)$: given by the source model (see also subsequent discussion).

$F(x^n)$: is in principle laborious, since it is defined as a sum of $f(\cdot)$ over all texts which are on the left of x^n : about 2^{n-1} terms (in the average).

But, this sum can be decomposed in a different way : sum of the probs of the subtrees which are on the left of x^n : only about $\frac{n}{2}$ terms.

Let $T_{x_1 x_2 \cdots x_{k-1} 0}$ denote the sub-tree pending below the prefix $x_1 x_2 \cdots x_{k-1} 0$. The frequency of this sub-tree is

$$\begin{aligned} f(T_{x_1 x_2 \cdots x_{k-1} 0}) &= \sum_{y_{k+1} \cdots y_n} f(x_1 x_2 \cdots x_{k-1} 0 y_{k+1} \cdots y_n) && (11) \\ &= f(x_1 x_2 \cdots x_{k-1} 0), && (12) \end{aligned}$$

$\Rightarrow$ coding reduces to the computation of order n values of $f(\cdot)$.

For example, if we use a zero order model, we compute $f(x^n) = \prod_{i=1}^{n} f(x_i)$.

Thus, this improved version will require order n^2 operations.

Example

Binary text of the figure, with $f(1) = \theta$ et $f(0) = 1 - \theta$.

Order zero hypothesis (successive symbols independent) : $f(s_1, \ldots, s_n) = f(s_1) \cdots f(s_n$

Let us compute the value of $F(01110)$ (x^5 on the figure).

We find that

$$\begin{aligned} F(01110) &= f(T_1) + f(T_2) + f(T_3) \\ &= f(00) + f(010) + f(0110) \\ &= f(0)f(0) + f(0)f(1)f(0) + f(0)f(1)f(1)f(0) \\ &= f(0)(1 + f(1)(1 + f(1)))f(0) \\ &= (1-\theta)(1+\theta(1+\theta))(1-\theta), \end{aligned}$$

Observation : many identical terms in the $f(\cdot)$ which have to be recomputed.

$\Rightarrow$ Recursive computation of the $f(\cdot)$: linear time complexity

$\Rightarrow$ reduction of the number multiplications/additions by avoiding to recompute common factors of the f_i.

Encoding algorithm

The source symbols are treated sequentially :

1. Let x^k be the prefix already treated at stage k, $f(x^k)$ the corresponding relative frequency, $F(x^k)$ the cumulative frequency (left trees), and u_k the current node.
2. **Initialization :** $k = 0$; x^0 empty string; $f(x^0) = 1$; $F(x^0) = 0$
3. **Updating :** Let b denote $(k+1)$-the bit read of the source text.
   - if $b = 1$, $F(x^{k+1}) = F(x^k) + f(x^k 0)$.
   - if $b = 0$, $F(x^{k+1}) = F(x^k)$.
   - $x^{k+1} = x^k b$; $f(x^{k+1})$ see comments below; current node u_{k+1} is implicitly updated (following branch b) from node u_k.
4. **Iteration :** if $k < n$, we iterate, otherwise the values $F(x^n)$ and $f(x^n)$ are returned.
5. **Termination :** the codeword $\lfloor F(x^n) + f(x^n) \rfloor_{\lceil \log f(x^n) \rceil}$ is constructed.

Computing the $f(x^k)$ recursively

Independent symbols : $f(x^{k+1}) = f(x^k)f(x_{k+1})$

In general : $f(x^{k+1}) = f(x_{k+1}|x^k)f(x^k)$

Markov : $f(x^{k+1}) = f(x_{k+1}|x_k)f(x^k)$

m**-ary source alphabet :** cumulate frequencies of all left subtrees at each stage.

Decoding : works symetrically.

The decoder uses the binary expansion of the number $r = 0.w_i$ in order to select branches in the tree.

Same computations, leaving on the left all subtrees such that $F(x^k) < 0.w_i$.

At each transition the encoder will produce one source symbol.

The decoding process stops after n symbols

$\Rightarrow$ the decoder needs to be informed of the source message length.

On-line Algorithm

Stopping criterion is problematic.

Transmission of length n vs special end of text symbol "."

Why on-line ?

Average length

Zero order mode : $\frac{1}{n}$ bits more than the zero-order entropy limit $H_m(f_1, \ldots, f_m)$.

Remarkably flexible

Can easily adapt to any left-to-right oriented probabilistic source model.

Technicalities (...)

Mainly : computing with the very-long dyadic fractions (high-precision real-number computations)

E.g: text of 1MB $\rightarrow$ a real number with about 10^6 bits precision.

Data compression with higher order models

Instead of using the model of “monogram” $f(s_i)$, one uses the frequencies of multi-grams $f(s^{k+1})$ (for an order k model).

Models : either provided a priori or determined from the given text, or from a sample of representative texts.

If model depends on encoded text : $\Rightarrow$ overhead (transmit multigram frequencies).

Higher order Huffman encoding

How would you do ? In practice : two possible solutions

1. Code blocs of length $k+1 \Rightarrow$ big Huffman tree.

2. Construct m^k small Huffman trees for the conditional distributions $f(s_{k+1}|s^k)$, and take into account the previous k symbols to encode/decode $\Rightarrow$ border effects (initialization)

Which one is better : no general rule.

Adaptative data compression

These techniques allow us to treat two problems :

1. Non stationary sources : a single code is not good for the whole text.

2. Don’t need to transmit probabilistic model to decoder (on-line...)

Very simple generic idea :

Let $T = s^N$ be the text to endode/decode.

When coding (and hence also when decoding) the k-the symbol we use a probabilistic model determined from the already seen symbole (prefix s^{k-1}).

Model initialized e.g. with a unform distribution, and then updated sequentially after each source symbol.

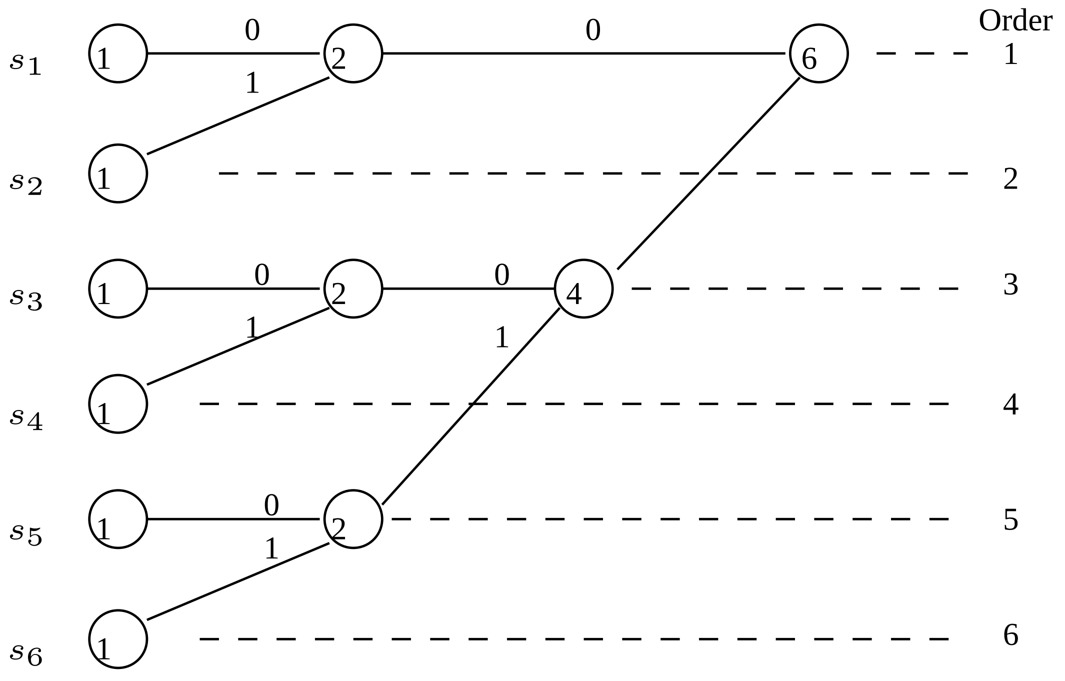
NB: idea is also compatible with higher order source models.

Implementation details (...)

Huffman

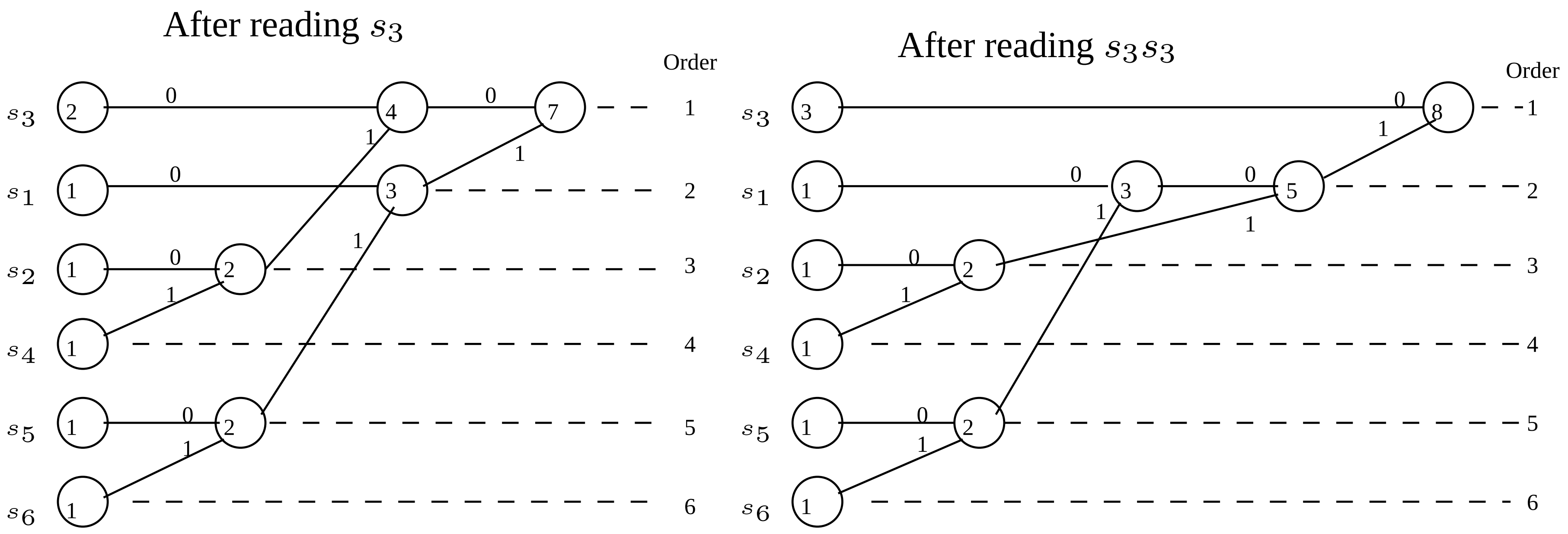
Since the source model changes after each symbol, the codetree must be recomputed $\Rightarrow$ not very practical.

Example : a source with six symbols : intitialization of the tree : used to code the first symbol.



NB: we need some conventions to treat multiple possibilities.

After the next symbols have been read :



What you should remember :

∃ an efficient algorithm to update Huffman trees incrementally (Knuth-Gallager)

Adaptive arithmetic coding (the adaptive Rolls)

Think about it yourself...

Dictionary methods for data compression

Basic idea :

Use a dictionary (set of frequently used words)

Parse text using the dictionary :

$\rightarrow$ encode text as a sequence of addresses in the dictionary

NB: similar (but not identical) to source alphabet parsing idea.

NB: but no (e.g. SPP) hypothesis on the contents of the dictionary.

Solutions: use a “library” of specialized dictionaries

E.g. : one dictionary for English texts, one for LaTeX source code…

Problem: maintenance of dictionaries; does not work for a “random” text

NB: dictionary methods $\rightarrow$ a generic approach in AI…

“Universal” dictionary methods

Rebuild the dictionary on the fly for each text, incrementally as the text is read.

$\rightarrow$ “universal” adaptive methods

$\Rightarrow$ algorithms invented by Lempel and Ziv (1977-78)

Tow basic methods : LZ77 et LZ78

$\Rightarrow$ numerous implementations (e.g. GNUzip, PKZIP, COMPRESS, GIF…)

Basic principle and a few discussions follow.

Basic Lempel-Ziv Algorithm

- one starts with an empty dictionary;
- then, at each step one reads symbols as long as current prefix belongs to the dictionary;
- the prefix together with the next source symbol form a word which is not yet in the dictionary $\Rightarrow$ this new word is inserted in the dictionary

  E.g. if $T = 1011010100010\ldots$, this yields $1, 0, 11, 01, 010, 00, 10, \ldots$.
- The present word is encoded : address of prefix in the dictionary + last bit

Let $c(n)$ denote the address (integer) in the dictionary. We have the following for our example text :

source words	λ	1	0	11	01	010	00	10
$c(n)$	0	1	2	3	4	5	6	7
$c(n)_{\text{binary address}}$	000	001	010	011	100	101	110	111
(address, bit)	–	(000,1)	(000,0)	(001,1)	(010,1)	(100,0)	(010,0)	(001,0)

$\Rightarrow$ encoded text : $U = 0001, 0000, 0011, 0101, 1000, 0100, 0010.$

Why does this idea allow to compress ?

Because the size of the dictionary grows “slowly” with the size of the source text.

Let $c(N)$ be the number of encoded entries for a text of length N.

$\Rightarrow \lceil \log c(N) - 1 \rceil + 1$ bits for every word

$\Rightarrow$ in average : $\frac{c(N)(\log c(N)+1)}{N}$ bits/symbol

One can show that : $\Rightarrow$ asympotically $\lim_{n\to\infty} \frac{c(n)(\log c(n)+1)}{n} = H(\mathcal{S})$

almost surely for messages of any stationary ergodic source

$\Rightarrow$ “universale” algorithm

On-line character : problem = address coding

Solution ⇒ use current dictionary size to determine number of bits.

source words	λ	1	0	11	01	010	00	10
$c(n)$	0	1	2	3	4	5	6	7
$c(n)_{\text{binray address}}$	000	001	010	011	100	101	110	111
$\lceil \log_2 c(n) \rceil$	-	0	1	2	2	3	3	3
(address, bit)	–	(,1)	(0,0)	(01,1)	(10,1)	(100,0)	(010,0)	(001,0)

$\Rightarrow U = 1, 00, 011, 101, 1000, 0100, 0010$

Adaptativity : ⇒ local dictionary

Variants : dictionary management, address coding (e.g. Huffman)

Relative optimality : ⇒ not very competitive in general but very robust (no assumption about source behavior).

The asympotic performances are reached only when the dictionary starts to become representative : contains a significant fraction of sufficiently long typical messages.
⇒ for very long texts

Summary of text compression : we have seen the state of the art.

Complementarity of good source models and good coding algorithms $\Rightarrow$ need both

Codes of fixed (given) word lengths : $\Rightarrow$ conceptual tool for AEP

Symbol codes : Huffman

Stream codes :

1. Are not constrained to use at least one bit per source symbol
$\Rightarrow$ work also for a binary source alphabet

2. Arithmetic coding : a nice probabilistic approach (source modeling)
$\Rightarrow$ allow one to exploit a priori knowledge about the real world.

3. Lempel-Ziv : universal method, able to learn "everything" about a stationary ergodic source, at the expense of more data (longer messages).

$$\boxed{\textbf{Data compression} \simeq \textbf{Automatic learning}}$$