Huffman Coding

National Chiao Tung University
Chun-Jen Tsai
10/2/2014

Huffman Codes

- Optimum prefix code developed by D. Huffman in a class assignment
- Construction of Huffman codes is based on two ideas:
  - In an optimum code, symbols with higher probability should have shorter codewords
  - In an optimum prefix code, the two symbols that occur least frequently will have the same length (otherwise, the truncation of the longer codeword to the same length still produce a decodable code)

Principle of Huffman Codes

- Starting with two least probable symbols γ and δ of an alphabet A, if the codeword for γ is $[m]0$, the codeword for δ would be $[m]1$, where $[m]$ is a string of 1's and 0's.
- Now, the two symbols can be combined into a group, which represents a new symbol ψ in the alphabet set. The symbol ψ has the probability $P(\gamma) + P(\delta)$.
- Recursively determine the bit pattern $[m]$ using the new alphabet set.

Example: Huffman Code

- Let $\mathcal{A} = \{a_1, \ldots, a_5\}$, $P(a_i) = \{0.2, 0.4, 0.2, 0.1, 0.1\}$.

Symbol	Step 1	Step 2	Step 3	Step 4	Codeword
a_2	0.4	0.4	0.4	0.6 0	1
a_1	0.2	0.2	0.4 0	0.4 1	01
a_3	0.2	0.2 0	0.2 1		000
a_4	0.1 0	0.2 1			0010
a_5	0.1 1				0011

Combine last two symbols with lowest probabilities, and
use one bit (last bit in codeword) to differentiate between them!

Efficiency of Huffman Codes

- Redundancy – the difference between the entropy and the average length of a code

Letter	Probability	Codeword
a_2	0.4	1
a_1	0.2	01
a_3	0.2	000
a_4	0.1	0010
a_5	0.1	0011

The average codeword length for this code is

$$l = 0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4 = 2.2 \text{ bits/symbol.}$$

The entropy is around 2.13. Thus, the redundancy is around 0.07 bits/symbol.

- For Huffman code, the redundancy is zero when the probabilities are negative powers of two.

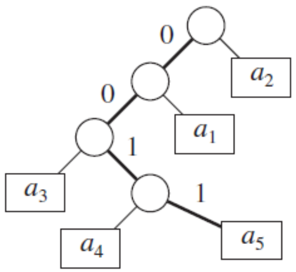
Minimum Variance Huffman Codes

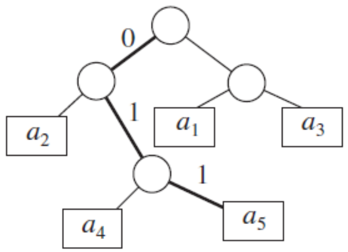
- When more than two “symbols” in a Huffman tree have the same probability, different merge orders produce different Huffman codes.

Symbol	Step 1	Step 2	Step 3	Step 4	Codeword
a_2	0.4	0.4	0.4	0.6 0	00
a_1	0.2	0.2	0.4 0	0.4 1	10
a_3	0.2	0.2 0	0.2 1		11
a_4	0.1 0	0.2 1			010
a_5	0.1 1				011

The average codeword length is still 2.2 bits/symbol.
But variances are different!

- Two code trees with same symbol probabilities:



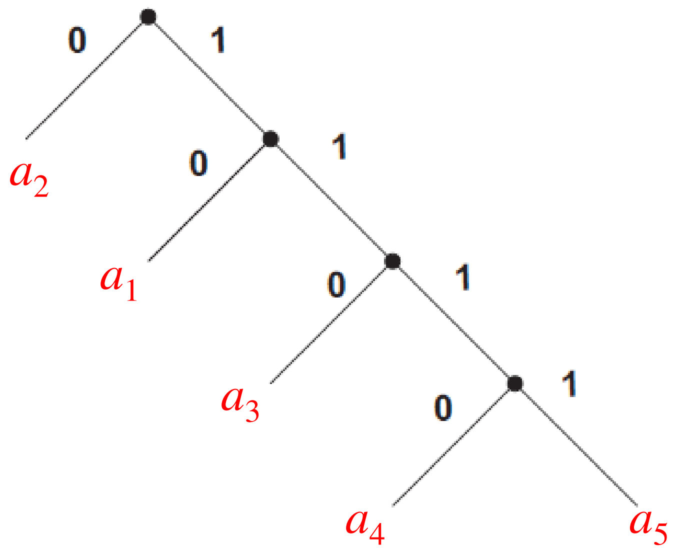


We prefer a code with smaller length-variance, Why?

Canonical Huffman Codes

- Transmitting the code table to the receiver of the messages may be expansive.
- If a canonical Huffman tree is used, we can just send the code lengths of the symbols to the receiver.

- Example:
  If the code length of $\{ a_1, a_2, a_3, a_4, a_5 \}$ are $\{2, 1, 3, 4, 4\}$, what is the code table?



Length-Limited Huffman Codes

- Optimal code design only concerns about minimizing the average codeword length.
- Length-limited code design tries to minimize the maximal codeword length l_{max} as well. If m is the size of the alphabet, clearly we have $l_{max} \geq \lceil \log_2 m \rceil$.
- The package-merge algorithm by Larmore and Hirchberg (1990) can be used to design length-limited Huffman codes.

Example: Package-Merge Algorithm

Letter	Probability	Codeword
a_1	0.05	0100
a_2	0.1	0101
a_3	0.15	011
a_4	0.2	10
a_5	0.2	11
a_6	0.3	00

Average codeword length = 2.45

$$L_0 = [a_1(0.05), a_2(0.1), a_3(0.15), a_4(0.2), a_5(0.2), a_6(0.3)]$$

Length limit = 3

$$\text{Package}_1 : [a_{12}(.15), a_{34}(.35), a_{56}(.5)]$$

$$\text{Merge}_1 : [a_1(0.05), a_2(0.1), a_3(0.15), a_{12}(0.15), a_4(0.2), a_5(0.2), a_6(0.3), a_{34}(0.35), a_{56}(0.5)]$$

Odd number of items, discard the highest probability item!

$$\text{Package}_2 : [a_{12}(0.15), a_{312}(0.3), a_{45}(0.4), a_{634}(0.65)]$$

$$\text{Merge}_2 : [a_1(0.05), a_2(0.1), a_3(0.15), a_{12}(0.15), a_4(0.2), a_5(0.2), a_6(0.3), a_{312}(0.3), a_{45}(0.4), a_{634}(0.65)]$$

Count the number of occurrences of each symbol, the codeword lengths are: { 3, 3, 3, 3, 2, 2 }

Average codeword length = 2.5

Letter	Probability	Codeword
a_1	0.05	100
a_2	0.1	101
a_3	0.15	110
a_4	0.2	111
a_5	0.2	00
a_6	0.3	01

Conditions for Optimal VLC Codes

- Given any two letters, a_j and a_k, if $P[a_j] \geq P[a_k]$, then $l_j \leq l_k$, where l_j is the number of bits in the codeword for a_j.
- The two least probable letters have codewords with the same maximum length l_m.
- In the tree corresponding to the optimum code, there must be two branches stemming from each intermediate node.
- Suppose we change an intermediate node into a leaf node by combining all of the leaves descending from it into a composite word of a reduced alphabet. Then, if the original tree was optimal for the original alphabet, the reduced tree would be optimal for the reduced alphabet.

Length of Huffman Codes (1/2)

- Given a sequence of positive integers $\{l_1, l_2, \ldots, l_k\}$ satisfies

$$\sum_{i=1}^{k} 2^{-l_i} \leq 1,$$

there exists a uniquely decodable code whose codeword lengths are given by $\{l_1, l_2, \ldots, l_k\}$.

- The optimal code for a source $\mathcal{S}$ has an average code length l_{avg} with the following bounds:

$$H(\mathrm{S}) \leq l_{avg} < H(\mathrm{S}) + 1,$$

where $H(\mathcal{S})$ is the entropy of the source.

Length of Huffman Codes (2/2)

- The lower-bound can be obtained by showing that:

$$H(\mathcal{S}) - l_{avg} = -\sum_{i=1}^{k} P(a_i)\log_2 P(a_i) - \sum_{i=1}^{k} P(a_i) l_i$$

$$= \sum_{i=1}^{k} P(a_i)\log_2\left[\frac{2^{-l_i}}{P(a_i)}\right] \le \log_2\left[\sum_{i=1}^{k} 2^{-l_i}\right] \le 0.$$

Jensen's inequality

- For the upper-bound, notice that given an alphabet $\{a_1, a_2, \ldots, a_k\}$, and a set of codeword lengths

$$l_i = \lceil \log_2(1/P(a_i)) \rceil < \log_2(1/P(a_i)) + 1,$$

the code satisfies the Kraft-McMillan inequality and has $l_{\text{avg}} < H(\mathcal{S}) + 1$.

Extended Huffman Code (1/2)

- If a symbol a has probability 0.9, ideally, it's codeword length should be 0.152 bits $\rightarrow$ not possible with Huffman code (since minimal codeword length is 1)!
- To fix this problem, we can group several symbols together to form longer code blocks. Let $A = \{a_1, a_2, \ldots, a_m\}$ be the alphabet of an i.i.d. source $\mathcal{S}$, thus

$$H(\mathcal{S}) = -\sum_{i=1}^{m} P(a_i)\log_2 P(a_i)$$

  We know that we can generate a Huffman code for this source with rate R (bits per symbol) such that

$$H(\mathcal{S}) \leq R < H(\mathcal{S}) + 1.$$

Extended Huffman Code (2/2)

- If we group n symbols into a new "extended" symbol, the extended alphabet becomes:

$$A^{(n)} = \{\overbrace{a_1 a_1 \ldots a_1}^{n \text{ times}}, a_1 a_1 \ldots a_2, \ldots, a_m a_m \ldots a_m\}.$$

There are m^n symbols in $A^{(n)}$. For such source $S^{(n)}$, the rate $R^{(n)}$ satisfies:

$$H(S^{(n)}) \leq R^{(n)} < H(S^{(n)}) + 1.$$

Note that $R = R^{(n)} / n$ and $H(S^{(n)}) = nH(S)$.

Therefore, by grouping symbols, we can achieve

$$H(S) \leq R < H(S) + \frac{1}{n}.$$

Example: Extended Huffman Code

- Consider an i.i.d. source with alphabet $A = \{a_1, a_2, a_3\}$ and model $P(a_1) = 0.8$, $P(a_2) = 0.02$, and $P(a_3) = 0.18$. The entropy for this source is 0.816 bits/symbol.

Huffman code

Letter	Codeword
a_1	0
a_2	11
a_3	10

Average code length = 1.2 bits/symbol

Extended Huffman code

Letter	Probability	Code
a_1a_1	0.64	0
a_1a_2	0.016	10101
a_1a_3	0.144	11
a_2a_1	0.016	101000
a_2a_2	0.0004	10100101
a_2a_3	0.0036	1010011
a_3a_1	0.1440	100
a_3a_2	0.0036	10100100
a_3a_3	0.0324	1011

Average code length = 0.8614 bits/symbol

Huffman Code Decoding

- Decoding of Huffman code can be expensive:
  - If a large sparse code table is used, memory is wasted
  - If a code tree is used, too many if-then-else's are required
- In practice, we employ a code tree where small tables are used to represents sub-trees

Letter	Code
A	0
B	10101
C	11
D	101000
E	10100101
F	1010011
G	100
H	10100100
I	1011

Letter	Code
0000	A, 1
0001	A, 1
0010	A, 1
0011	A, 1
0100	A, 1
0101	A, 1
0110	A, 1
0111	A, 1
1000	G, 3
1001	G, 3
1010	Table II
1011	I, 4
1100	C, 2
1101	C, 2
1110	C, 2
1111	C, 2

Letter	Code
00	D, 2
01	Table III
10	B, 1
11	B, 1

Letter	Code
00	H, 2
01	E, 2
10	F, 1
11	F, 1

Non-binary Huffman Codes

- Huffman codes can be applied to n-ary code space. For example, codewords composed of $\{0, 1, 2\}$, we have ternary Huffman code
- Let $A = \{a_1, \ldots, a_5\}$, $P(a_i) = \{0.25, 0.25, 0.2, 0.15, 0.15\}$.

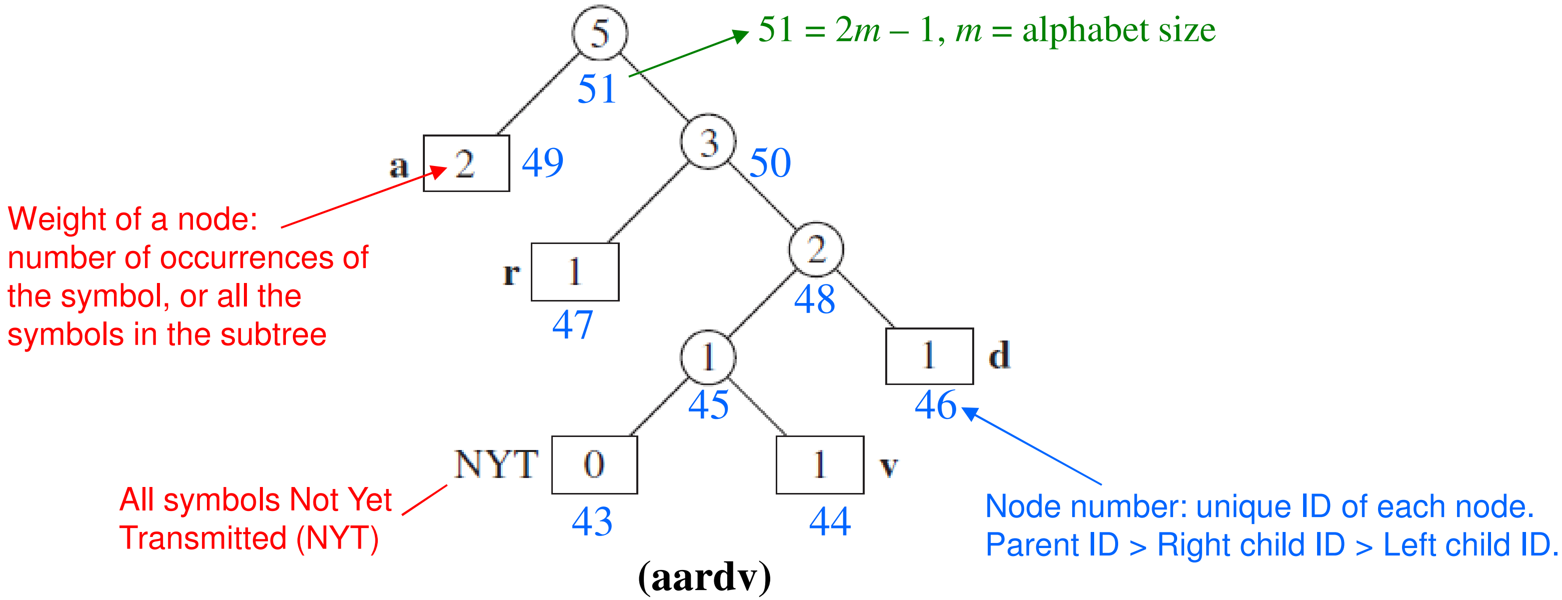
Symbol	Step 1	Step 2	Codeword
a_1	0.25	0.5 0	1
a_2	0.25	0.25 1	2
a_3	0.20 0	0.25 2	00
a_4	0.15 1		01
a_5	0.15 2		02

Adaptive Huffman Coding

- Huffman codes require exact probability model of the source to compute optimal codewords. For messages with unknown duration, this is not possible.
- One can try to re-compute the probability model for every received symbol, and re-generate a new set of codewords based on the new model for the next symbol from scratch $\rightarrow$ too expensive!
- Adaptive Huffman coding tries to achieve this goal at lower cost.

Adaptive Huffman Coding Tree

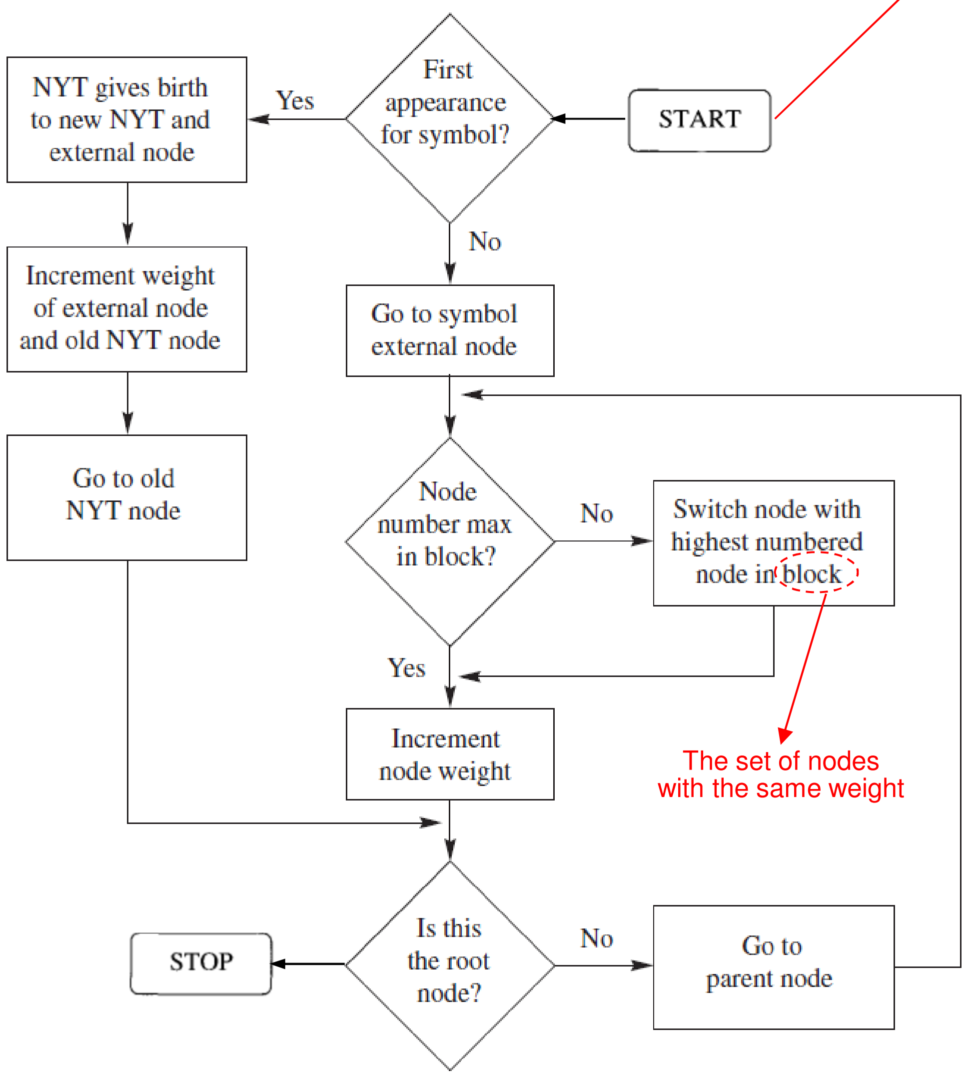
- Adaptive Huffman coding maintains a dynamic code tree. The tree will be updated synchronously on both transmitter-side and receiver-side. If the alphabet size is m, the total number of nodes $\leq 2m - 1$.

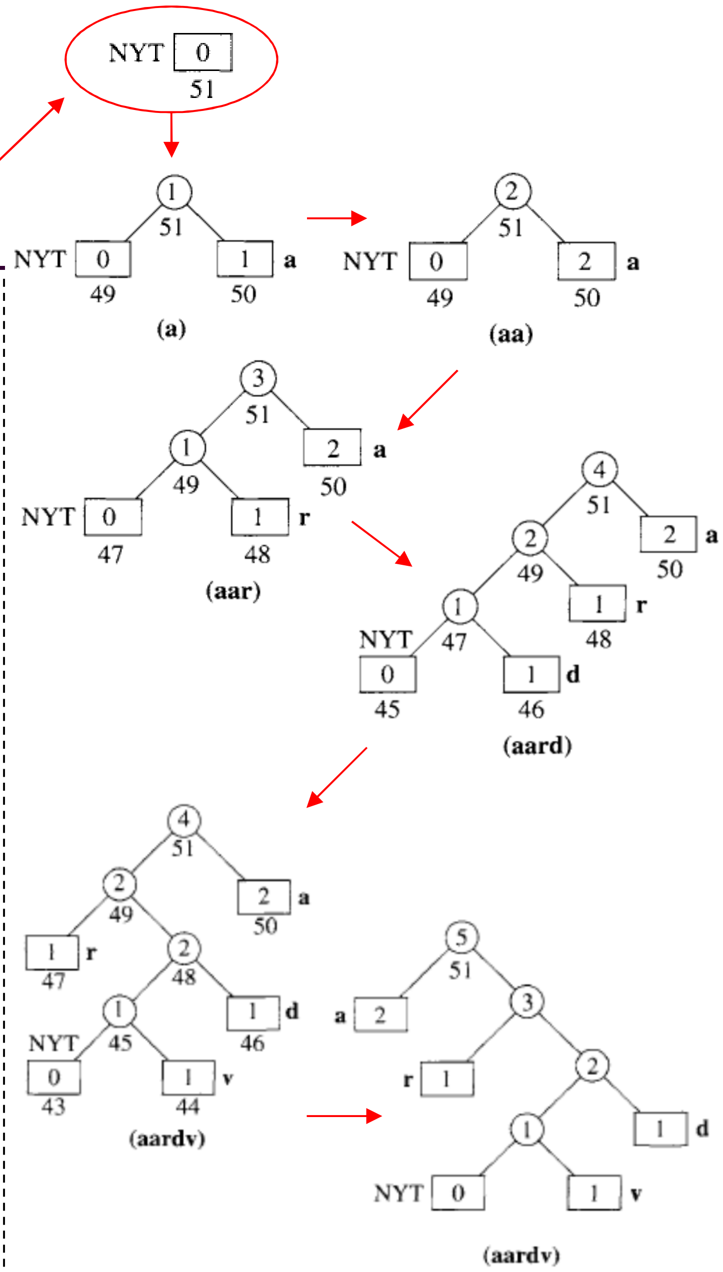


Initial Codewords

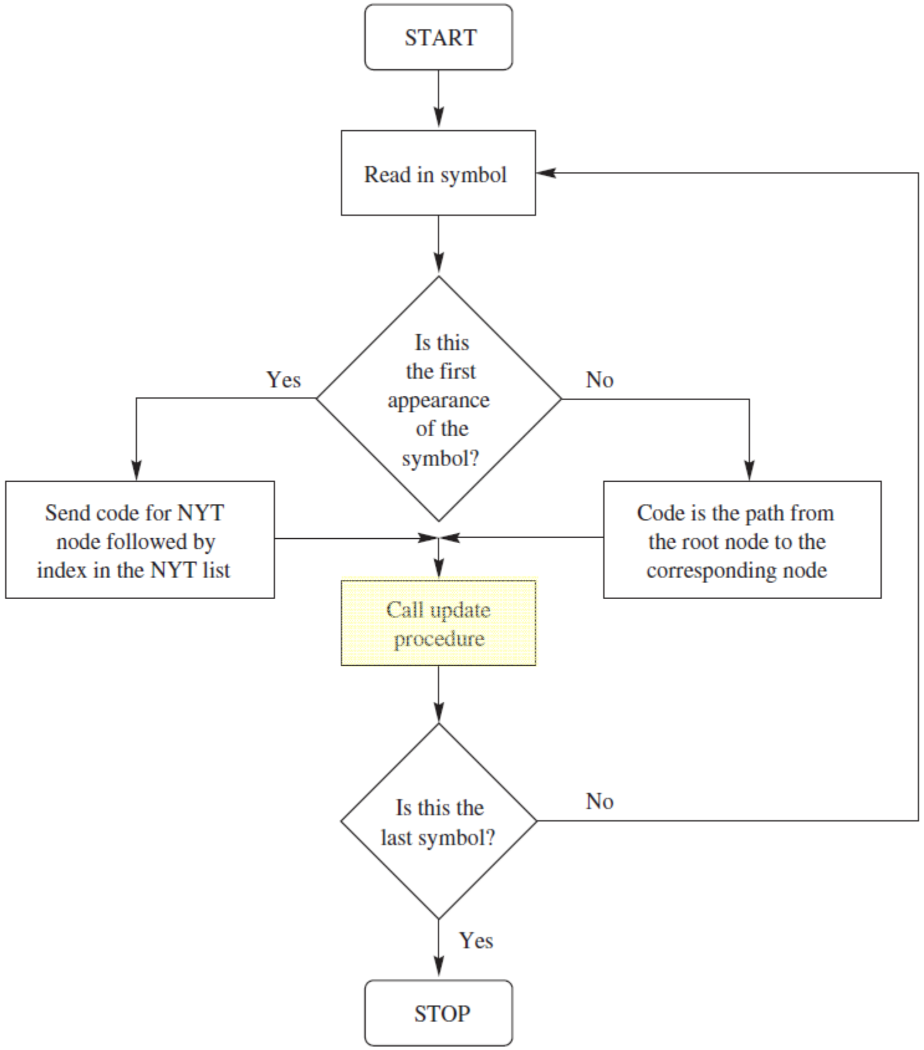
- Before transmission of any symbols, all symbols in the source alphabet $\{a_1, a_2, \ldots, a_m\}$ belongs to the NYT list.
  - Each symbol in the alphabet has an initial codeword using either $\lfloor \log_2 m \rfloor$ or $\lfloor \log_2 m \rfloor + 1$ bits fixed-length binary code.
- When a symbol a_i is transmitted for the first time, the code for NYT is transmitted, followed by the fixed code for a_i. A new node is created for a_i and a_i is taken out of the NYT list.
- From this point on, we follow the update procedure to maintain the Huffman code tree.

Update Procedure

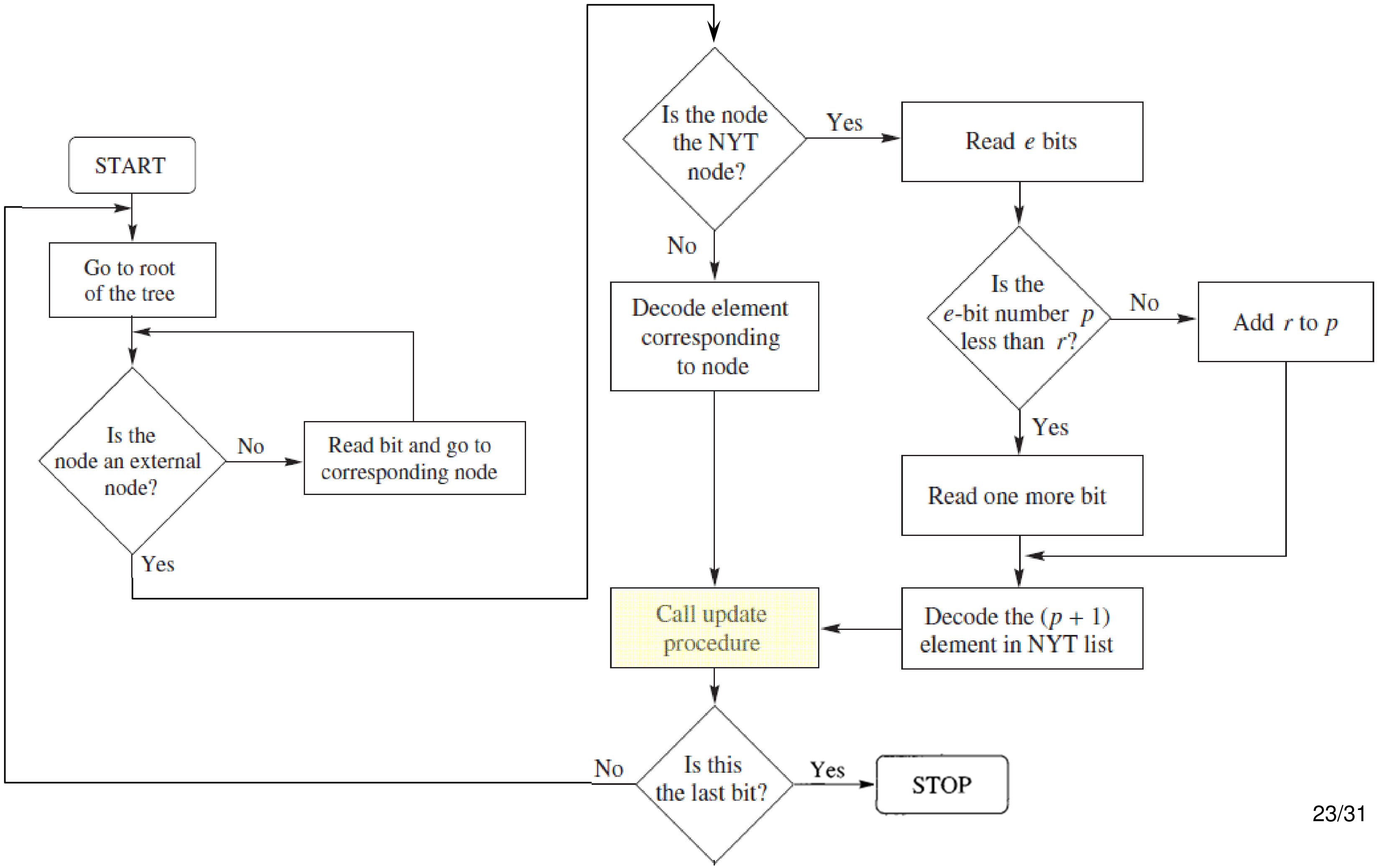




Encoding Procedure



Decoding Procedure



Unary Code

- Golomb-Rice codes are a family of codes that designed to encode integers where the larger the number, the smaller the probability
- Unary code:
  The codeword of n is n 1's followed by a 0.
  For example:

$$4 \rightarrow 11110,\ 7 \rightarrow 11111110, \text{ etc.}$$

  Unary code is optimal when $A = \{1, 2, 3, \ldots\}$ and

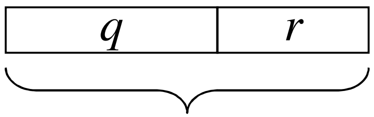
$$P(k) = \frac{1}{2^k}.$$

Golomb Codes

- For Golomb code with parameter m, the codeword of n is represented by two numbers q and r,

$$q = \left\lfloor \frac{n}{m} \right\rfloor, \quad r = n - qm,$$

  where q is coded by unary code, and r is coded by fixed-length binary code (takes $\lfloor \log_2 m \rfloor \sim \lceil \log_2 m \rceil$ bits).
- Example, $m = 5$, r needs $2 \sim 3$ bits to encode:



a Golomb codeword

n	q	r	Codeword	n	q	r	Codeword
0	0	0	000	8	1	3	10110
1	0	1	001	9	1	4	10111
2	0	2	010	10	2	0	11000
3	0	3	0110	11	2	1	11001
4	0	4	0111	12	2	2	11010
5	1	0	1000	13	2	3	110110
6	1	1	1001	14	2	4	110111
7	1	2	1010	15	3	0	111000

Optimality of Golomb Code

- It can be shown that the Golomb code is optimal for the probability model

$$P(n) = p^{n-1}q, \quad q = 1 - p,$$

when

$$m = \left\lceil -\frac{1}{\log_2 p} \right\rceil.$$

Rice Codes

- A pre-processed sequence of non-negative integers is divided into blocks of J integers.
  - The pre-process involves differential coding and remapping
- Each block coded using one of several options, e.g., the CCSDS options (with $J = 16$):
  - *Fundamental sequence option*: use unary code
  - *Split sample option*: an n-bit number is split into least significant m bits (FLC-coded) and most significant $(n - m)$ bits (unary-coded).
  - *Second extension option*: encode low entropy block, where two consecutive values are inputs to a hash function. The function value is coded using unary code.
  - *Zero block option*: encode the number of consecutive zero blocks using unary code

Tunstall Codes

- Tunstall code uses fixed-length codeword to represent different number of symbols from the source $\rightarrow$ errors do not propagates like variable-length codes (VLC).

- Example: The alphabet is $\{A, B\}$, to encode the sequence AAABAABAABAABAAA:

2-bit Tunstall code, OK

Sequence	Codeword
AAA	00
AAB	01
AB	10
B	11

Non-Tunstall code, Bad!

Sequence	Codeword
AAA	00
ABA	01
AB	10
B	11

Tunstall Code Algorithm

- Two design goals of Tunstall code
  - Can encode/decode any source sequences
  - Maximize source symbols per each codeword
- To design an n-bit Tunstall code (2^n codewords) for an i.i.d. source with alphabet size N:
  1. Start with N symbols of the source alphabet
  2. Remove the most probable symbol, add N new entries to the codebook by concatenate the rest of symbols with the most probable one
  3. Repeat the process in step 2 for K time, where

$$N + K(N - 1) \le 2^n.$$

Example: Tunstall Codes

- Design a 3-bit Tunstall code for alphabet $\{A, B, C\}$ where $P(A) = 0.6$, $P(B) = 0.3$, $P(C) = 0.1$.

Initial list

Letter	Probability
A	0.60
B	0.30
C	0.10

First iteration

Sequence	Probability
B	0.30
C	0.10
AA	0.36
AB	0.18
AC	0.06

Second iteration

Sequence	code
B	000
C	001
AB	010
AC	011
AAA	100
AAB	101
AAC	110

Applications: Image Compression

- Direct application of Huffman coding on image data has limited compression ratio

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	7.01	57,504	1.14
Sensin	7.49	61,430	1.07
Earth	4.94	40,534	1.62
Omaha	7.12	58,374	1.12

→ no model prediction

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	4.02	32,968	1.99
Sensin	4.70	38,541	1.70
Earth	4.13	33,880	1.93
Omaha	6.42	52,643	1.24

→ with model prediction
$(x'_n = x_{n-1})$