THE ANALYSIS OF ENUMERATIVE SOURCE CODES AND THEIR USE IN BURROWS–WHEELER COMPRESSION ALGORITHMS

by

Andre Martin McDonald

Submitted in partial fulfilment of the requirements for the degree

Master of Engineering (Electronic Engineering)

in the

Department of Electrical, Electronic and Computer Engineering

Faculty of Engineering, Built Environment and Information Technology

UNIVERSITY OF PRETORIA

February 5, 2010

Study leader: Professor J. C. Olivier

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

THE ANALYSIS OF ENUMERATIVE SOURCE CODES AND THEIR USE IN BURROWS–WHEELER COMPRESSION ALGORITHMS

by

Andre Martin McDonald

Submitted in partial fulfilment of the requirements for the degree

Master of Engineering (Electronic Engineering)

in the

Department of Electrical, Electronic and Computer Engineering

Faculty of Engineering, Built Environment and Information Technology

UNIVERSITY OF PRETORIA

February 5, 2010

Study leader: Professor J. C. Olivier

Many people supported me while I worked on this project. I would like to extend my gratitude to:

Mom and Dad for helping me through all the tough times — you are an example to other parents, and I am fortunate to be your son;

Corne Olivier for providing me with opportunities to grow (both personally and professionally), and who remained steadfast in his support. You went above and beyond the call of duty as study leader;

Prof. Gideon Kuhn for the insightful discussions and for your valuable advice — you are an inspiration to the next generation of engineers;

Prof. Hendrik Ferreira from the University of Johannesburg, for your advice and the opportunities you have provided me;

Profs. Frans Willems and Tjalling Tjalkens from the Technical University of Eindhoven, for your hospitality, advice, and the technical discussions;

Prof. Han Vinck for your hospitality during my visit to Germany, and for providing me with the opportunity to learn from your group;

Prof. Jos Weber for your friendly advice;

Anel Bekker, for remaining patient while I worked on this project, and for your support. You are very special, and I am a richer person for knowing you;

Charl Bruwer, who remained a true friend in difficult times.

And last, but definitely not least, I would like to thank God for providing me with the ability to explore His wonderful creation, and for all the joy this has provided me over the years. I hope to fulfil my potential to better the lives of others.

SUMMARY

THE ANALYSIS OF ENUMERATIVE SOURCE CODES AND THEIR USE IN BURROWS–WHEELER COMPRESSION ALGORITHMS
by
Andre Martin McDonald
Study leader: Professor J. C. Olivier

Master of Engineering (Electronic Engineering)
Department of Electrical, Electronic and Computer Engineering
Faculty of Engineering, Built Environment and Information Technology
University of Pretoria

In the late 20th century the reliable and efficient transmission, reception and storage of information proved to be central to the most successful economies all over the world. The Internet, once a classified project accessible to a selected few, is now part of the everyday lives of a large part of the human population, and as such the efficient storage of information is an important part of the information economy. The improvement of the information storage density of optical and electronic media has been remarkable, but the elimination of redundancy in stored data and the reliable reconstruction of the original data is still a desired goal.

The field of source coding is concerned with the compression of redundant data and its reliable decompression. The arithmetic source code, which was independently proposed by J. J. Rissanen and R. Pasco in 1976, revolutionized the field of source coding. Compression algorithms that use an arithmetic code to encode redundant data are typically more effective and computationally more efficient than compression algorithms that use earlier source codes such as extended Huffman codes. The arithmetic source code is also more flexible than earlier source codes, and is frequently used in adaptive compression algorithms. The arithmetic code remains the source code of choice, despite having been introduced more than 30 years ago.

The problem of effectively encoding data from sources with known statistics (i.e. where the probability distribution of the source data is known) was solved with the introduction of the arithmetic code. The probability distribution of practical data is seldomly available to the source encoder, however. The source coding of data from sources with unknown statistics is a more challenging problem, and remains an active research topic.

Enumerative source codes were introduced by T. J. Lynch and L. D. Davisson in the 1960s. These lossless source codes have the remarkable property that they may be used to effectively encode source sequences from certain sources without requiring any prior knowledge of the source statistics. One drawback of these source codes is

the computationally complex nature of their implementations. Several years after the introduction of enumerative source codes, J. G. Cleary and I. H. Witten proved that approximate enumerative source codes may be realized by using an arithmetic code. Approximate enumerative source codes are significantly less complex than the original enumerative source codes, but are less effective than the original codes. Researchers have become more interested in arithmetic source codes than enumerative source codes since the publication of the work by Cleary and Witten.

This thesis concerns the original enumerative source codes and their use in Burrows–Wheeler compression algorithms. A novel implementation of the original enumerative source code is proposed. This implementation has a significantly lower computational complexity than the direct implementation of the original enumerative source code. Several novel enumerative source codes are introduced in this thesis. These codes include optimal fixed–to–fixed length source codes with manageable computational complexity.

A generalization of the original enumerative source code, which includes more complex data sources, is proposed in this thesis. The generalized source code uses the Burrows–Wheeler transform, which is a low–complexity algorithm for converting the redundancy of sequences from complex data sources to a more accessible form. The generalized source code effectively encodes the transformed sequences using the original enumerative source code. It is demonstrated and proved mathematically that this source code is universal (i.e. the code has an asymptotic normalized average redundancy of zero bits).

SAMEVATTING

DIE ANALISE VAN TEL–BRONKODES EN DIE GEBRUIK DAARVAN IN BURROWS–WHEELER KOMPRESSIEALGORITMES
deur
Andre Martin McDonald
Studieleier: Professor J. C. Olivier

Meester van Ingenieurswese (Elektroniese Ingenieurswese)
Departement van Elektriese, Elektroniese en Rekenaaringenieurswese
Fakulteit van Ingenieurswese, Bou–omgewing en Inligtingstegnologie
Universiteit van Pretoria

Die betroubare en doeltreffende versending, ontvangs en berging van inligting vorm teen die einde van die twintigste eeu die kern van die mees suksesvolle ekonomieë in die wêreld. Die Internet, eens op 'n tyd 'n geheime projek en toeganklik vir slegs 'n klein groep verbruikers, is vandag deel van die alledaagse lewe van 'n groot persentasie van die mensdom, en derhalwe is die doeltreffende berging van inligting 'n belangrike deel van die inligtingsekonomie. Die verbetering van die bergingsdigteid van optiese en elektroniese media is merkwaardig, maar die uitwissing van oortolligheid in gebergde data, asook die betroubare herwinning van oorspronklike data, bly 'n doel om na te streef.

Bronkodering is gemoeid met die kompressie van oortollige data, asook die betroubare dekompressie van die data. Die rekenkundige bronkode, wat onafhanklik voorgestel is deur J. J. Rissanen en R. Pasco in 1976, het 'n revolusie veroorsaak in die bronkoderingsveld. Kompressiealgoritmes wat rekenkundige bronkodes gebruik vir die kodering van oortollige data is tipies meer doeltreffend en rekenkundig meer effektief as kompressiealgoritmes wat vroeëre bronkodes, soos verlengde Huffman kodes, gebruik. Rekenkundige bronkodes, wat gereeld in aanpasbare kompressiealgoritmes gebruik word, is ook meer buigbaar as vroeëre bronkodes. Die rekenkundige bronkode bly na 30 jaar steeds die bronkode van eerste keuse.

Die probleem om data wat afkomstig is van bronne met bekende statistieke (d.w.s. waar die waarskynlikheidsverspreiding van die brondata bekend is) doeltreffend te enkodeer is opgelos deur die instelling van rekenkundige bronkodes. Die bronenkodeerder het egter selde toegang tot die waarskynlikheidsverspreiding van praktiese data. Die bronkodering van data wat afkomstig is van bronne met onbekende statistieke is 'n groter uitdaging, en bly steeds 'n aktiewe navorsingsveld.

T. J. Lynch and L. D. Davisson het tel–bronkodes in die 1960s voorgestel. Tel–bronkodes het die merkwaardige eienskap dat bronsekwensies van sekere bronne effektief met hierdie foutlose kodes geënkodeer kan word, sonder dat die bronenkodeerder

enige vooraf kennis omtrent die statistieke van die bron hoef te besit. Een nadeel van tel–bronkodes is die hoë rekenkompleksiteit van hul implementasies. J. G. Cleary en I. H. Witten het verskeie jare na die instelling van tel–bronkodes bewys dat benaderde tel–bronkodes gerealiseer kan word deur die gebruik van rekenkundige bronkodes. Benaderde tel–bronkodes het 'n laer rekenkompleksiteit as tel–bronkodes, maar benaderde tel–bronkodes is minder doeltreffend as die oorspronklike tel–bronkodes. Navorsers het sedert die werk van Cleary en Witten meer belangstelling getoon in rekenkundige bronkodes as tel–bronkodes.

Hierdie tesis is gemoeid met die oorspronklike tel–bronkodes en die gebruik daarvan in Burrows–Wheeler kompressiealgoritmes. 'n Nuwe implementasie van die oorspronklike tel–bronkode word voorgestel. Die voorgestelde implementasie het 'n beduidende laer rekenkompleksiteit as die direkte implementasie van die oorspronklike tel–bronkode. Verskeie nuwe tel–bronkodes, insluitende optimale vaste–tot–vaste lengte tel–bronkodes met beheerbare rekenkompleksiteit, word voorgestel.

'n Veralgemening van die oorspronklike tel–bronkode, wat meer komplekse databronne insluit as die oorspronklike tel–bronkode, word voorgestel in hierdie tesis. The veralgemeende tel–bronkode maak gebruik van die Burrows–Wheeler omskakeling. Die Burrows–Wheeler omskakeling is 'n lae–kompleksiteit algoritme wat die oortolligheid van bronsekwensies wat afkomstig is van komplekse databronne omskakel na 'n meer toeganklike vorm. Die veralgemeende bronkode enkodeer die omgeskakelde sekwensies effektief deur die oorspronklike tel–bronkode te gebruik. Die universele aard van hierdie bronkode word gedemonstreer en wiskundig bewys (d.w.s. dit word bewys dat die kode 'n asimptotiese genormaliseerde gemiddelde oortolligheid van nul bisse het).

TABLE OF CONTENTS

Mathematical notation

Several of the mathematical expressions presented in this thesis contain vectors, random variables, and matrices. All vectors are printed in bold — an n–element vector is expressed as

$$\begin{aligned} \mathbf{x}^n &= \{x_1, x_2, \ldots x_n\} \\ &= x_1 x_2 \ldots x_n \end{aligned} \tag{1}$$

where $x_1, x_2, \ldots x_n$ denotes the vector elements. All sequences are expressed using vector notation. In certain cases, the vector $\mathbf{x}^n$ is denoted by $\mathbf{x}$ (i.e. the length of the vector does not form part of its symbol).

Some of the expressions of this thesis involve one or more subsequences of an original sequence. A subsequence of the original sequence $\mathbf{x}$ is expressed as

$$\mathbf{x}_i^j = \{x_i, x_{i+1}, \ldots x_j\} \tag{2}$$

if $j \geq i$, and

$$\mathbf{x}_i^j = \{x_i, x_{i-1}, \ldots x_j\} \tag{3}$$

if $j < i$.

All random variables are expressed using capital letters. A vector of random variables is expressed as

$$\begin{aligned} \mathbf{X}^n &= \{X_1, X_2, \ldots X_n\} \\ &= X_1 X_2 \ldots X_n \end{aligned} \tag{4}$$

where $X_1, X_2, \ldots X_n$ denotes the random variables. Random sequences are expressed using vector notation, and subsequences of random sequences are expressed in the same manner as the subsequences of ordinary sequences. The vector $\mathbf{X}^n$ is also denoted as $\mathbf{X}$ in certain cases.

Matrices appear in a very small fraction of the expressions presented in this thesis, and are expressed using capital letters that are printed in bold. Matrices are clearly identified in the text in order to distinguish them from vectors of random variables.

Abbreviations and Acronyms

AEP	Asymptotic equipartition property
ASCII	American standard code for information interchange
avg.	Average
AWFC	Advanced weighted frequency count
bit	Binary digit
BWT	Burrows–Wheeler transform
CACM	Communications of the Association for Computing Machinery
CCIR	Consultative Committee on International Radio
CCITT	International Telegraph and Telephone Consultative Committee
CD	Compact disk
CDF	Cumulative distribution function
codec	Coder–decoder
CT	Context tree
DCC	Data Compression Conference
DCT	Distance coding transform
DNA	Deoxyribonucleic acid
DVD	Digital video disk
ECG	Electrocardiogram
EOF	End of file
EOL	End of line

EXP	Exponent
FSM	Finite state machine
FSMX	Finite–order FSM
GIF	Graphics interchange format
GNU	GNU's not UNIX
GSM	Global system for mobile communications
gzip	GNU zip
i.i.d.	Independent and identically distributed
IFT	Inversion frequencies transform
ITU	International Telecommunication Union
ITU–T	ITU telecommunication standardization sector
JPEG	Joint Photographic Experts Group
kB	Kilobyte
KT	Krichevsky–Trofimov
LBIV	Left–bigger inversion vector
LIPT	Length index preserving transform
LSB	Least significant bit
LSS	Least significant symbol
LUP	List update problem
LZ77	Lempel–Ziv 1977
LZ78	Lempel–Ziv 1978
LZW	Lempel–Ziv–Welch
MB	Megabyte
MSB	Most significant bit
MSS	Most significant symbol
p.i.i.d.	Piecewise independent and identically distributed
PPM	Prediction by partial match
PPM*	PPM with unbounded context length

prob.	Probability
RLE	Run–length encoder
RLE–0	Zero run length
RMB	RLE mantissa buffer
SBR	Sort–by–rank
seq.	Sequence
SOR	Start of run
src.	Source
theo.	Theory / Theoretical
TS(0)	Deterministic time–stamp algorithm
VLSI	Very large scale integration
w.r.t.	With respect to
WFC	Weighted frequency count

LIST OF SYMBOLS

$\beta_m(\cdot)$	Refer to equation 6.8 on page 142.
δ	An arbitrarily small, positive, real–valued constant (pages 5 and 161). A real–valued constant between 0 and 0.5 (page 115). A real–valued constant between 0 and 1 (pages 116 and 160).
$\delta(\cdot,\cdot)$	The Kronecker delta function.
\$	The end–of–file symbol.
ϵ	The escape symbol.
γ	A parameter of an algorithm for updating symbol frequency counts (page 87). A real–valued constant greater than zero (page 177).
Λ	The set of all possible values that the source parameter θ may assume (page 27). The concatenation function for two or more sequences (elsewhere).
λ	The parameter of a Poisson probability distribution.
λ_j, λ_n	Constants that are related to Stirling's approximation of the factorial of an integer.
μ	A real–valued constant greater than zero.
μ_c	The average time required by a practical implementation of a source encoder to encode a source sequence (seconds).
μ_d	The average time required by a practical implementation of a source decoder to decode a source sequence (seconds).
Ω	A random–valued parameter of an information source.
σ	A constant that is approximately equal to 0.08607.
σ_c	The standard deviation of the time required by a practical implementation of a source encoder to encode a source sequence (seconds).
σ_d	The standard deviation of the time required by a practical implementation of a source decoder to decode a source sequence (seconds).
Θ	A random–valued parameter of an information source.
θ	A deterministic parameter of an information source.
$\theta(j)$	The number of primes less than or equal to the integer j.
$\mathbf{0}^m$	The m–bit sequence that consists only of zero–valued bits.
$1(\cdot)$	An operator that is evaluated as unity if its operand is true, and zero if not.

$\mathcal{A}$	The symbol alphabet of an information source.
$\mathcal{A}^n$	The n–fold Cartesian product of the symbol alphabet $\mathcal{A}$.
$\mathcal{A}_\delta^n$	A set that consists of all n–bit sequences with a weight of w bits, where $w \in \{0, 1, \ldots \lfloor \delta n \rfloor\}$.
$\lvert\mathcal{A}\rvert$	The cardinality of the symbol alphabet $\mathcal{A}$ (symbols).
a_i	The short notation for $a_i(m, k)$ (page 137). The ith symbol of a symbol alphabet (elsewhere).
$a_i(m, k)$	The exponent of the prime number p_i in the prime–factor–based decomposition of the binomial coefficient $B_{m,k}$.
$\mathbb{B}$	The set $\{0, 1\}$.
$\mathbf{B}^{m'}$	The binary word that is assigned to the random q–ary source symbol X.
B_i	Bit i of the binary word that is assigned to the random q–ary source symbol X.
$B_{n,m}$	The binomial coefficient $\binom{n}{m}$.
$B_m(\cdot)$	Refer to section 6.1.1.3 on page 142.
$\mathrm{BWT}(\cdot)$	The forward Burrows–Wheeler transform function.
$\mathrm{BWT}^{-1}(\cdot, \cdot)$	The reverse Burrows–Wheeler transform function.
$\mathbf{b}^{m'}$	The binary word that is assigned to the deterministic q–ary source symbol x.
b_i	Bit i of the binary word that is assigned to the deterministic q–ary source symbol x.
$b_{j,k}$	Bit j of the binary word that is assigned to the q–ary symbol x_k of the deterministic source sequence $\mathbf{x}$.
$b_{n,k}^{(j)}$	The branch value between the integers $N_{n,k}$ and $N_{n+1,k+j}$ in the general version of Pascal's triangle.
$\mathscr{C}$	A source code.
C	The total number of i.i.d. symbol segments in a BWT output sequence.
$\mathcal{C}^n$	The set of all uniquely decodable fixed–to–variable length source codes for sequences of n source symbols.
$\mathbf{C}_\mathbf{x}$	The codeword that is assigned to the random source sequence $\mathbf{X}$.
$C(\mathbf{x}_1^j)$	The number of source sequences in the set S_b with a prefix equal to $\mathbf{x}_1^j$.
$C_\mathrm{D}(j)$	The total number of bit operations involved in the computation of the factorial $j!$
$\mathcal{C}_{n,m}(\cdot, \cdot)$	The concatenation function for an n–symbol sequence and an m–symbol sequence.
$C_\mathrm{LB}(n)$	The size of the binomial coefficient lookup table that is used to encode sequences of up to n symbols (bits).
$C_\mathrm{LF}(n)$	The size of the factorial lookup table that is used to encode sequences of up to n symbols (bits).

c	A real–valued constant (page 45).
	The integer codeword that is assigned to an index sequence (as used in the example on page 105).
$\hat{c}_i$	The running total prior to decoding the ith index of an index sequence.
$c_{\mathbf{x}}$	The integer codeword that is assigned to the deterministic source sequence $\mathbf{x}$.
$\mathbf{c}_{\mathbf{x}}$	The codeword that is assigned to the deterministic source sequence $\mathbf{x}$.
$\mathbf{c}_j$	Refer to lemma 6.2.4 on page 213.
$\mathbf{c}'_j$	Refer to equation 6.220 on page 214.
$\mathbf{c}_{j,k}$	Refer to equation 6.207 on page 212.
$\mathbf{c}'_{j,k}$	Refer to equation 6.213 on page 213.
$\mathbf{c}_{1,j,k}$	Refer to equation 6.208 on page 212.
$c_i(j)$	The exponent of the prime number p_i in the prime factor decomposition of the integer j.
D	The set $\{\mathbf{d} \in \mathbb{N}^w : 1 \leq d_i \leq n \ \wedge \ d_i > d_j \ \forall \ i > j\}$.
$\mathcal{D}_{n,m}(\cdot)$	The division function for an $(n+m)$–symbol sequence (the sequence is divided into an n–symbol and an m–symbol sequence).
$D(\delta\|\|p)$	The Kullback–Leibler divergence between the bit distributions $\{\delta, 1-\delta\}$ and $\{p, 1-p\}$, where $0 < \delta, p < 1$.
$D(p(\mathbf{x}^n)\|\|q(\mathbf{x}^n))$	The Kullback–Leibler divergence between the distributions $p(\mathbf{x}^n)$ and $q(\mathbf{x}^n)$.
d_n	Refer to equation 5.3 on page 130.
d_x	Refer to equation 5.2 on page 130.
$\mathbf{d}_{\mathbf{x}}$	The degree vector $\{d_{1,\mathbf{x}}, d_{2,\mathbf{x}} \ldots d_{w,\mathbf{x}}\}$ of the nonzero–valued bits of the bit sequence $\mathbf{x}$.
$d_{i,\mathbf{x}}$	The degree of the $(n-i+1)$th nonzero–valued bit of the sequence $\mathbf{x}$.
$\mathcal{E}$	A set that consists of all the suffixes of all the sequences that are associated with the leaf nodes of a tree.
$e_{i,k}$	The integer in row i and column k of the coding matrix (refer to equation 4.8 on page 107).
EOF	The end–of–file symbol.
EOL	The ASCII end–of–line symbol.
F	A discrete monotone function.
$f(\cdot)$	The next–state function that is associated with an FSM source (page 39).
	The function that appears in the definition of the Lee–weight of a source sequence (page 123).
	The function that is defined as $f \triangleq f_2 \circ f_1$ (refer to equation 6.6 on page 141).

$f_1(\cdot)$	Refer to section 6.1.1.3 on page 141.
$f_2(\cdot)$	Refer to equation 6.7 on page 142.
$f_3(\cdot)$	Refer to equation 6.48 on page 155.
$f_{rr}(x_i)$	The integer that is assigned to the source symbol x_i by the recency–rank encoder.
$f_{int}(x_i)$	The integer that is assigned to the source symbol x_i by the encoder of the interval code.
$f(y\|\mathbf{x}^k)$	A source model's frequency count of the symbol y, immediately prior to encoding symbol x_{k+1} of the sequence $\mathbf{x}$.
G	A discrete monotone function.
$g_1(\cdot)$	Refer to equation 6.9 on page 142.
$g_2(\cdot)$	Refer to equation 6.47 on page 155.
$g_3(\cdot)$	Refer to equation 6.106 on page 178.
$g_4(\cdot)$	Refer to equation 6.202 on page 211.
H	The average 'amount' of information that an information source produces (bits).
$\mathcal{H}_n(\cdot)$	A function that maps a sequence to its n–symbol prefix.
$H(X)$	The entropy of the source symbol X (bits).
$H(\mathcal{X})$	The entropy rate of an information source (bits per symbol).
$\hat{H}(i)$	Refer to equations 6.97 and 6.194 on pages 176 and 209.
$H(\mathbf{X}^n)$	The entropy of the source sequence $\mathbf{X}^n$ (bits).
$H_y(X)$	The entropy of a symbol that the FSM closure of a context–tree source produces in state $\mathbf{s}'_y$ (bits).
$\hat{H}_{\mathbf{x}}(\mathcal{X})$	The entropy rate associated with the empirical distribution of the source sequence $\mathbf{X}$ (bits per symbol).
$\hat{H}(i,j)$	Refer to equations 6.98 and 6.195 on pages 176 and 209.
$H(\mathbf{X}^n\|\Theta)$	The conditional entropy of the source sequence $\mathbf{X}^n$ (bits).
$h(\cdot)$	The binary entropy function.
$\mathcal{I}$	The BWT index that is associated with a random source sequence.
I	A suffix array (page 38). The BWT index that is associated with a deterministic source sequence (pages 169 and 204).
I_n	The set $\{j \in \mathbb{N}_0 : 0 \leq j \leq 2^n - 1\}$.
$I(k)$	Element k of the suffix array I.
$i_{S_b}(\mathbf{x}^n)$	The index of the sequence $\mathbf{x}^n$ in the ordered set S_b.
$I(\mathbf{X}^n;\Theta)$	The mutual information of the sequence $\mathbf{X}^n$ and the parameter Θ (bits).
K_n	A real–valued constant between 0 and 1, defined for each integer n greater than one.
k	The number of blocks in a block segment (page 119). The length of each codeword of a fixed–to–fixed length source code, measured in bits (elsewhere).

k_{ave}	The mean number of source bits that are represented by a codeword of a variable–to–fixed length source code (refer to equation 4.16 on page 111).
$k_1(n)$	The length of a level–one block of the segmentation algorithm, under the assumption that source sequences of n symbols are encoded (symbols).
$k_2(n)$	The length of a level–two block of the segmentation algorithm, under the assumption that source sequences of n symbols are encoded (symbols).
$\mathcal{L}_{\mathcal{T}}$	The set of leaf nodes of the tree that is defined by the set $\mathcal{T}$.
$L(X)$	The average length of codewords that are assigned to individual source symbols X (bits).
$L(\mathbf{X}^n)$	The average length of codewords that are assigned to source sequences $\mathbf{X}^n$ (bits).
$L_{\mathscr{C}}(\mathbf{X}^n\|\theta)$	The conditional average codeword length that is associated with the source code $\mathscr{C}$ (bits).
l_A	The length of a codeword that is produced by the encoder of an arithmetic code, measured in bits (refer to equation 6.44 on page 151).
l_k	A level of the full binary tree that is associated with an extended Huffman code.
$l(\mathbf{x})$	The length of the codeword that is assigned to the sequence $\mathbf{x}$ (bits).
$l(a_i)$	The length of the codeword that is assigned to the alphabet symbol a_i (bits).
$l_{\mathscr{C}}(\mathbf{x}^n)$	The length of the codeword that is assigned to the sequence $\mathbf{x}^n$ by the encoder of the source code $\mathscr{C}$ (bits).
$l_i(\mathbf{x})$	The length of the ith field of the codeword that is assigned to the sequence $\mathbf{x}$ (bits).
l_{ss}	The length of a subsequence (bits).
$l'_j(\mathbf{x})$	Refer to equation 6.219 on page 214.
$l^{(s)}_{1,i}(\mathbf{x})$	The length of the codeword that is assigned to the ith i.i.d. symbol segment of the BWT output sequence, where it is assumed that the sequence $\mathbf{x}$ is encoded (bits).
$l_{i,j}(\mathbf{x})$	The length of the ith field of the codeword that is assigned to the jth i.i.d. symbol segment of the BWT output sequence, where it is assumed that the sequence $\mathbf{x}$ is encoded (bits).
$l'_{j,k}(\mathbf{x})$	Refer to equation 6.212 on page 213.
$l_{i,j,k}(\mathbf{x})$	The length of the ith field of the codeword that is assigned to the subsequence $S_{j,k}(\mathbf{x})$ (bits).

$l'_{v,i,j,k}(\mathbf{x})$ The length of the vth field of the codeword that is assigned to the subsequence $S_{j,k}(\mathbf{y}'_i)$, where $\mathbf{y}'_i$ denotes the ith symbol segment of the BWT output sequence (which may include symbols from up to m additional segments), and where it is assumed that the sequence $\mathbf{x}$ is encoded (bits).

$l_{v,i,j,k}(\mathbf{x})$ The length of the vth field of the codeword that is assigned to the subsequence $S_{j,k}(\mathbf{y}_i)$, where $\mathbf{y}_i$ denotes the ith i.i.d. symbol segment of the BWT output sequence, and where it is assumed that the sequence $\mathbf{x}$ is encoded (bits).

$\overline{l}_{v,i,j,k}(\mathbf{x})$ The length of vth field of the codeword that is assigned to the subsequence $S_{j,k}(\overline{\mathbf{y}}_i)$, where $\overline{\mathbf{y}}_i$ denotes the ith expected i.i.d. symbol segment of the BWT output sequence, and where it is assumed that the sequence $\mathbf{x}$ is encoded (bits).

$\hat{l}_{v,i,j,k}(\mathbf{x})$ The length of the vth field of the codeword that is assigned to the subsequence $S_{j,k}(\mathbf{v}_i)$, where $\mathbf{v}_i$ denotes the sequence of symbols that the context–tree source produces in its ith lexicographically–ranked state, and where it is assumed that the sequence $\mathbf{x}$ is encoded (bits).

$l(w,k)$ The number of bits that are required to encode the weights of the k blocks in a block segment, where w denotes the total weight of the blocks (refer to equation 4.33 on page 119).

$M(r)$ The metric of the rth level–one block of the segmentation algorithm.

m The length, in bits, of each short block of the combinatorial source code (refer to section 4.1.7 on page 121).
The length, in symbols, of the longest context of a finite–memory source or context–tree source (elsewhere).

m' Refer to equation 6.174 on page 200.

m_k The number of codewords of an extended Huffman code that occupy level l_k of a full binary tree.

m_{ab} The frequency count of the bit pair ab in a source sequence.

$m(i,k)$ Refer to equation 4.25 on page 113.

$\mathbb{N}$ The set of natural numbers, $\{1,2,3,\ldots\}$.

$\mathbb{N}_0$ The set of natural numbers, including zero, $\{0,1,2,\ldots\}$.

$\mathbf{N}$ Refer to equations 6.238 and 6.279 on pages 217 and 223.

$\mathbf{N}''$ Refer to equation 6.281 on page 223.

$N_{j,k}$ The total number of j–symbol q–ary sequences with a Lee–weight of k (page 124).
The length of the random subsequence $S_{j,k}(\mathbf{X})$, measured in bits (elsewhere).

$N_i(0)$ The number of zero–valued bits in the ith level–one block of the segmentation algorithm.

$N_i(1)$ The number of nonzero–valued bits in the ith level–one block of the segmentation algorithm.

$N(n,l,q)$	Refer to equation 4.43 on page 124.
n	The length of a source sequence (symbols).
$\mathbf{n}'$	Refer to equations 6.240 and 6.282 on pages 217 and 223.
$\mathbf{n}''$	Refer to equation 6.284 on page 223.
n_i	The length of the ith i.i.d. symbol segment of the BWT output sequence (symbols).
$\overline{n}_i$	The expected length of the ith i.i.d. symbol segment of the BWT output sequence (symbols).
n_i''	An integer element of the set $\{0,1,\ldots n\}$.
n_{yy}	The short notation for $n_{yy}(\mathbf{x}_i^n)$, which denotes the number of bit pairs yy that are present in the sequence $\mathbf{x}_i^n$ (page 128).
$n_{j,k}$	The length of the deterministic subsequence $S_{j,k}(\mathbf{x})$ (bits).
$n'_{j,k}$	An integer element of the set $\{0,1,\ldots n\}$.
$n_{i,j,k}$	The length of the deterministic subsequence $S_{j,k}(\mathbf{y}_i)$, where $\mathbf{y}_i$ denotes the ith i.i.d. symbol segment of the BWT output sequence (bits).
$n'_{i,j,k}$	An integer element of the set $\{0,1,\ldots n_i''\}$.
$\hat{n}_{i,j,k}$	The length of the deterministic subsequence $S_{j,k}(\mathbf{v}_i)$, where $\mathbf{v}_i$ denotes the sequence of symbols that the context–tree source produces in its ith lexicographically–ranked state (bits).
$n(\mathbf{s})$	The total number of symbols that occur within context $\mathbf{s}$ of a source sequence.
$n_d(\mathbf{s})$	The total number of distinct symbols that occur within context $\mathbf{s}$ of a source sequence.
$n_z(\mathbf{x}^n)$	The number of times that the symbol z (or conditioning class z) appears in the sequence $\mathbf{x}^n$.
$n_{zy}(\mathbf{x}^n)$	The number of times that the symbol y appears within conditioning class z of the sequence $\mathbf{x}^n$.
$n(x_i,\mathbf{s})$	The number of times that the symbol x_i appears within context $\mathbf{s}$ of the sequence $\mathbf{x}$.
$n_{S_b}(x_1,x_2,\ldots x_j)$	The number of sequences in the ordered set S_b with the prefix $\{x_1,x_2,\ldots x_j\}$.
$O(\cdot)$	Bachmann–Landau (big–O) notation.
$\mathbf{P}$	The state probability vector of the FSM closure of a context–tree source.
P	The pattern set that is associated with a source code (refer to section 4.1.9 on page 124).
P_b	The block–error probability of a source code (refer to equation 6.31 on page 147).

$\Pr(x)$	The probability of a random source symbol X being equal to x.
$\Pr(\mathbf{x}^n)$	The probability of a random source sequence $\mathbf{X}^n$ being equal to $\mathbf{x}^n$.
$P_c(\mathbf{x}^i)$	The Krichevsky–Trofimov estimate of the probability of occurrence of the sequence $\mathbf{x}^i$.
$\Pr(X = y)$	The probability of the random source symbol X being equal to y.
p	The probability that a random source bit from a stationary binary memoryless source has a nonzero value.
$\hat{p}$	Refer to equation 6.73 on page 160.
p_i	The running product after the ith step of the computation of a factorial (refer to equation 5.5 on page 131). The ith prime number (page 137). The probability that the FSM closure of a context–tree source is in state $\mathbf{s}'_i$ (elsewhere).
$p_{j,k}$	The probability of a state transition to state $\mathbf{s}'_k$ of the FSM closure of a context–tree source, conditioned on the FSM closure being in state $\mathbf{s}'_j$ (refer to equations 6.90 and 6.167 on pages 173 and 195). Refer to equation 6.244 on page 218.
$p''_{j,k}$	Refer to equation 6.247 on page 219.
$p_{i,j,k}$	Refer to equation 6.287 on page 224.
$p''_{i,j,k}$	Refer to equation 6.290 on page 225.
$p_{\max}$	The probability of occurrence of the most likely symbol in a symbol alphabet.
$\text{pre}(\mathbf{x}^n)$	A prefix of the sequence $\mathbf{x}^n$.
$\mathcal{Q}$	The set that consists of the reversed states of a finite–memory source.
$\mathcal{Q}^c$	The set of states that correspond to novel symbol contexts. The novel symbol contexts are a result of applying the BWT to sequences from a context–tree source.
$\mathcal{Q}'$	The union of the sets $\mathcal{Q}$ and $\mathcal{Q}^c$.
$\mathcal{Q}(\mathbf{x}^n, d_{w,x})$	The number of n–bit source sequences with a weight of w that are found to be numerically smaller than the sequence $\mathbf{x}^n$ (only those source sequence bits with a degree less than or equal to $d_{w,\mathbf{x}}$ are considered during the comparison).
q	The probability that a random source bit from a stationary binary memoryless source has a value of zero (page 109). The number of symbols in the symbol alphabet of a q–ary source (elsewhere).
q_i	The probability that a context–tree source is in its ith lexicographically–ranked state.
$\mathbf{q}$	The set $\{0, 1, \ldots n\}^{2^{m'}-1}$.
$\mathbf{q}_1$	The set $\{0, 1, \ldots n\}^{\lvert\mathcal{S}\rvert}$.
$\mathbf{q}_2$	The set $\{0, 1, \ldots n\}^{\lvert\mathcal{S}\rvert(2^{m'}-1)}$.

R	A p.i.i.d. symbol probability distribution.
R_a	The code rate of a fixed–to–fixed length source code, measured in bits per symbol (refer to equation 6.77 on page 161).
R_n	The entropy–normalized redundancy of a source code (refer to equation 6.78 on page 163).
R'_s	The normalized strong minimax redundancy of a source code (bits per symbol).
R'_w	The normalized weak minimax redundancy of a source code (bits per symbol).
$\mathcal{R}(\cdot)$	The sequence reversal function.
$R(\mathbf{X}^n)$	The average per–codeword redundancy of a source code, assuming that n–symbol sequences $\mathbf{X}^n$ are encoded (bits).
$R'(\mathbf{X}^n)$	The normalized average per–codeword redundancy of a source code, assuming that n–symbol sequences $\mathbf{X}^n$ are encoded (bits per symbol).
$R'_{min,ave}$	The normalized minimum average redundancy of a source code (bits per symbol).
$R'_{\mathscr{C}}(\mathbf{X}^n, \theta)$	The normalized redundancy of the source code $\mathscr{C}$, conditioned on the source parameter θ (bits per symbol).
r_a	Refer to equation 4.15 on page 110.
r_i	Refer to equation 4.29 on page 114.
$r_i(x)$	The number of times that the symbol x appears in the first i symbols of a BWT output sequence.
$\mathcal{S}$	A finite set of states that is associated with a context–tree source.
$\mathbf{S}$	The state–transition probability matrix of the FSM closure of a context–tree source.
$\mathcal{S}'$	A finite set of states that is associated with the FSM closure of a context–tree source.
$\lvert\mathcal{S}\rvert$	The number of states that are associated with a context–tree source.
$\lvert\mathcal{S}'\rvert$	The number of states that are associated with the FSM closure of a context–tree source.
$\mathbf{S}_{\text{tr,c}}$	The compact transition matrix of a context–tree source's FSM closure.
S_b	An ordered set of sequences.
S_i	Refer to equation 6.49 on page 156.
$S_{j,k}(\cdot)$	Refer to equation 6.178 on page 202.
$\mathbf{s}_i$	The ith state of the state set that is associated with a context–tree source.
$\mathbf{s}'_i$	The ith state of the state set that is associated with the FSM closure of a context–tree source.
$\mathbf{s}_i^{(t)}$	The state in which a source resides at time instance i.
SOR	The start–of–run symbol.

$\text{suf}(\mathbf{x}^n)$	A suffix of the sequence $\mathbf{x}^n$.
$\text{suf}_c(l)$	A set that consists of all the suffixes of the sequence that is associated with leaf node l of a tree.
$\mathcal{T}$	A set that consists of tree nodes.
$\mathcal{T}'$	A set of nodes that constitute a refined tree.
$\mathcal{T}_2$	The set of source sequences that are correctly encoded and decoded by the variable weight, fixed–to–fixed length source code.
$\mathcal{T}_2'$	The set of source sequences that are correctly encoded and decoded by the alternative to the variable weight, fixed–to–fixed length source code.
T_j	The index of the first symbol in the BWT output sequence that occurred within the jth lexicographically–ranked source context.
$\overline{T}_j$	The expected index of the first symbol in the BWT output sequence that occurred within the jth lexicographically–ranked source context.
$t_{x,y}$	The destination state of the transition that occurs when a source produces the symbol x in state $\mathbf{s}'_y$.
t_h	The run–length threshold of a run–length code.
t_{max}	The maximum number of source symbols that are potentially assigned nonzero–valued weights by the WFC algorithm as it updates the symbol ranks during a single iteration.
$\mathbf{V}_i(\mathbf{X})$	The sequence of symbols, from the random sequence $\mathbf{X}$, that was produced by a context–tree source in its ith lexicographically–ranked state.
$\mathbf{V}_i$	The short notation for $\mathbf{V}_i(\mathbf{X})$.
$\mathbf{v}_i(\mathbf{x})$	The sequence of symbols, from the deterministic sequence $\mathbf{x}$, that was produced by a context–tree source in its ith lexicographically–ranked state.
$\mathbf{v}_i$	The short notation for $\mathbf{v}_i(\mathbf{x})$.
v'_i	The length of segment i of the BWT output sequence (which may include bits from up to m additional segments).
v''_i	The weight of segment i of the BWT output sequence (which may include bits from up to m additional segments).
$v_{m,i}$	The exponent of the prime number p_i in the prime–factor–based decomposition of the factorial $m!$ (refer to equation 5.18 on page 138).
$\mathbf{W}$	Refer to equations 6.239 and 6.280 on pages 217 and 223.
W	The weight of a random source sequence (bits).
$\mathbf{W}^n$	The shortened BWT output sequence, assuming that the BWT was applied to a random source sequence that was reversed and appended with the end–of–file symbol.

$W(\mathbf{x})$	The weight of the deterministic bit sequence $\mathbf{x}$ (bits).
$W_{j,k}$	The weight of the random subsequence $S_{j,k}(\mathbf{X})$ (bits).
$W_i(a_j)$	The weight that the WFC algorithm assigns to the alphabet symbol a_j immediately prior to transforming the ith BWT output sequence symbol.
w	The weight of a deterministic source sequence (bits).
w'	Refer to equation 6.67 on page 159.
$\mathbf{w}'$	Refer to equations 6.241 and 6.283 on pages 217 and 223.
$\mathbf{w}^n$	The shortened BWT output sequence, assuming that the BWT was applied to a deterministic source sequence that was reversed and appended with the end–of–file symbol.
$w(n)$	The length of the sequence intervals that are encoded independently from one another, measured in symbols (refer to section 3.3.2.4 on page 83).
$w_{j,k}$	The short notation for $w_{j,k}(\mathbf{x})$.
$w'_{j,k}$	An integer element of the set $\{0, 1, \ldots n_{j,k}\}$.
w_k^d	The number of q–ary symbols that are equal to d in the sequence $\mathbf{x}_k^n$.
$w_f(t)$	The weight function of the WFC algorithm.
$w_{i,j,k}$	The short notation for $w_{j,k}(\mathbf{y}_i)$, where $\mathbf{y}_i$ denotes the ith i.i.d. symbol segment of the BWT output sequence.
$w'_{i,j,k}$	An integer element of the set $\{0, 1, \ldots n'_{i,j,k}\}$.
$\hat{w}_{i,j,k}$	The short notation for $w_{j,k}(\mathbf{v}_i)$, where $\mathbf{v}_i$ denotes the sequence of symbols that a context–tree source produces in its ith lexicographically–ranked state.
$w_L(\mathbf{x}^n)$	The Lee–weight of the source sequence $\mathbf{x}^n$.
$w_{j,k}(\mathbf{x})$	The weight of the deterministic subsequence $S_{j,k}(\mathbf{x})$ (bits).
X	A random source symbol.
$\mathbf{X}^n$	The random source sequence $\{X_1, X_2, \ldots X_n\}$.
$\hat{\mathbf{X}}^n$	The output sequence of a source decoder, when used to decode the codeword $\mathbf{C}_\mathbf{x}$ of the random sequence $\mathbf{X}^n$.
$\mathbf{X}_i^j$	The random sequence $\{X_i, X_{i+1}, \ldots X_j\}$.
X_i	The ith symbol of a random source sequence $\mathbf{X}$.
x	A deterministic source symbol.
$\mathbf{x}^n$	The deterministic source sequence $\{x_1, x_2, \ldots x_n\}$.
x_i	The ith symbol of the deterministic source sequence $\mathbf{x}$.
$\mathbf{x}_i^j$	The deterministic sequence $\{x_i, x_{i+1}, \ldots x_j\}$.
$\mathbf{Y}^n$	The BWT output sequence, assuming that the BWT was applied to a random n–symbol source sequence that was reversed.
y'	The complement of the bit y.
$\mathbf{y}'_i$	The ith symbol segment of the BWT output sequence (which may include symbols from up to m additional segments).

$\mathbf{y}_i$ The ith i.i.d. symbol segment of the BWT output sequence.

$\overline{\mathbf{y}_i}$ The ith expected i.i.d. symbol segment of the BWT output sequence.

$y_0(i)$ The running total of the zero–valued bits in a source sequence, immediately prior to processing its ith bit (refer to section 4.1.3 on page 108).

$y_1(i)$ The running total of the nonzero–valued bits in a source sequence, immediately prior to processing its ith bit (refer to section 4.1.3 on page 108).

$\mathbf{Z}^{n+1}$ The BWT output sequence, assuming that the BWT was applied to a random source sequence that was reversed and appended with the end–of–file symbol.

Z The set $\{z \in \mathbb{N}_0 : 0 \leq z \leq \binom{n}{w} - 1\}$.

$\mathbf{z}^{n+1}$ The BWT output sequence, assuming that the BWT was applied to a deterministic source sequence that was reversed and appended with the end–of–file symbol.

z_w Refer to equation 6.3 on page 140.

CHAPTER 1

Introduction

In the late 20th century the reliable and efficient transmission, reception and storage of information proved to be central to the most successful economies of the world. The Internet, once a classified project accessible to a select few, is now part of the everyday lives of a large percentage of the human population, and as such the efficient storage of information is an important part of the information economy.

The monetary values associated with information and information technology are staggering — in 2007, for example, a total of 29 922 million US dollars worth of recorded music was sold worldwide [1]. Cisco systems, which develops and manufactures routers and switches for the transmission and reception of information, sold 39 500 million US dollars worth of equipment in 2008 alone [2]. It is estimated that there were 3 399 million GSM cellular telephone subscriptions worldwide in October 2008 [3]. If each subscription represented a natural person, this total would represent more than half of the planet's population. Given these enormous numbers, it is clearly worthwhile to improve our capacity to store and transmit information.

It appears that the demand for information increases more rapidly than the rate at which cutting–edge technology can supply information. Researchers are starting to encounter fundamental physical limits in the design of information technology. Smaller and denser microprocessors, for example, operate at higher temperatures, and require special means of cooling. Physical limits are a constraint in the design of optical and electronic storage media. The improvement of the information storage density of these storage media has been impressive, but engineers are struggling to maintain their earlier rate of improvement.

Researchers are currently examining alternative means of storing and transmitting information in order to satisfy our increasing demand for it. These technologies are typically very expensive. A different approach to increasing the capacity of information technology to store and transmit information centers on the information itself. Digital electronics operate using two voltage levels, and therefore store and manipulate data in a binary format. The storage and transmission capacity of a digital device depends on the manner in which it represents information as a sequence of zero–valued bits and nonzero–valued bits. The art of source coding, or data compression, is concerned with the effective representation of information, in a manner that requires less capacity to store, transmit or receive.

Many of the technologies we use on a daily basis would not be possible were it not for source coding — examples include audio CDs, movie DVDs, cellular telephones, and even fax machines. To illustrate the impact of source coding, consider a typical 100–minute movie stored on a DVD. To store this movie in an uncompressed digital format, one can use the CCIR 601 standard [4]. Approximately 20 megabytes of memory

is required to store one second of the movie in this format. A typical single–layer DVD would be able to store approximately 4 minutes of the movie if it was uncompressed. A total of 25 DVDs would be required to store the entire 100–minute movie. The uncompressed representation of the movie is not only inefficient, but impractical as well. The actual source code that is used to represent a movie on a DVD reduces the size of the data by a factor of at least 25, which clearly illustrates the effectiveness (and necessity) of source coding.

The above example illustrates that data compression is an important enabling technology, and that research in this field is a worthwhile pursuit. Data compression was once considered the domain of a relatively small group of scientists and engineers, but has evolved into a multi–million dollar a year industry.

The arithmetic source code, which was independently proposed by J. J. Rissanen and R. Pasco in 1976, revolutionized the field of source coding. Compression algorithms that use an arithmetic code to encode redundant data are typically more effective and computationally more efficient than compression algorithms that use earlier source codes such as extended Huffman codes. The arithmetic source code is also more flexible than earlier source codes, and is frequently used in adaptive compression algorithms. The arithmetic code remains the source code of choice, despite having been introduced more than 30 years ago.

The problem of effectively encoding data from sources with known statistics (i.e. where the probability distribution of the source data is known) was solved with the introduction of the arithmetic code. The probability distribution of practical data is seldomly available to the source encoder, however. The source coding of data from sources with unknown statistics is a more challenging problem, and remains an active research topic.

Enumerative source codes were introduced by T. J. Lynch and L. D. Davisson in the 1960s [5,6]. These lossless source codes have the remarkable property that they may be used to effectively encode source sequences from certain sources without requiring any prior knowledge of the source statistics. The initial enumerative source codes were only applicable to sources without memory, as well as to first–order Markov sources [7]. One drawback of these source codes is the computationally complex nature of their implementations.

Several years after the introduction of enumerative source codes, J. G. Cleary and I. H. Witten [8] proved that approximate enumerative source codes may be realized by using an arithmetic code. Approximate enumerative source codes are significantly less complex than the original enumerative source codes, but are less effective than the original codes. Researchers have become more interested in arithmetic source codes than enumerative source codes since the publication of the work by Cleary and Witten.

1.1 The topic of this thesis

This thesis reinvestigates the original, exact enumerative source codes, and considers their use in Burrows–Wheeler compression algorithms. The exact enumerative codes for memoryless sources are generalized to sources with memory by using the Burrows–Wheeler transform [9]. This reversible transform typically changes the output of a

stationary ergodic context–tree source into a data sequence with a biased first–order distribution [10]. This data sequence may be source coded in a universal fashion using the exact enumerative source codes for memoryless sources.

Another topic that is addressed by this thesis concerns the development of efficient enumerative source code implementations. Both the computational complexity and the memory requirements of an enumerative source code implementation are of importance. A novel implementation of the original enumerative source code is proposed. This implementation has a significantly lower computational complexity than the direct implementation of the original enumerative source code.

A number of novel enumerative source codes are proposed in this thesis. These codes include optimal fixed–to–fixed length source codes with manageable computational complexity. Proofs regarding the unique decodability of these source codes are provided, and the performance of these codes are investigated both mathematically and empirically.

The motivation behind the use of enumerative source codes is twofold. Firstly, to the best of the author's knowledge, the generalization of enumerative source codes to sources with memory by using the Burrows–Wheeler transform has not been investigated in the literature. The effectiveness of these codes is of theoretical interest, as the original enumerative codes are universal source codes. Secondly, by developing more efficient implementations of exact enumerative source codes, these source codes may become practical.

This thesis is set out as follows. Chapter 2 follows the introduction and provides a brief history of source coding, and also introduces concepts which are referred to and used in the later chapters. Chapter 3 contains a summary of the literature that concerns the Burrows–Wheeler transform, as well as a summary of source codes that use this transform. Chapter 4 contains a summary of existing enumerative source codes. Chapter 5 covers the efficient computation of large binomial coefficients, which is a computationally intensive routine performed by enumerative source code implementations. Chapter 6 contains the mathematical analysis and the proofs regarding the unique decodability of the existing and novel enumerative source codes. Empirical results regarding the effectiveness of the proposed source codes are also presented in this chapter. Chapter 7 presents several conclusions regarding enumerative source codes and their use in Burrows–Wheeler–based compression algorithms.

CHAPTER 2

The history of source coding

This survey of source coding starts with the birth of information theory, and continues with more specific advances in source coding. Much of the information presented in this chapter was obtained from reference [11].

2.1 Shannon and the birth of information theory

Source coding has its roots in the landmark paper of Claude Shannon [12], which was published in 1948. Shannon established some of the fundamental laws of source coding and data transmission in his paper. By modeling information sources as stochastic processes, Shannon derived a measure of the average amount of information that a source produces. This measure, which is referred to as the entropy H of an information source, may be interpreted as the minimum number of bits that may be used, on average, to uniquely represent each source symbol. If fewer bits are used on average to represent each source symbol, some source symbols can not be assigned codewords that are uniquely decodable. Any excess bits that are used to represent the source symbols (i.e. in excess of the source entropy) are considered redundant.

Shannon [12] derived an expression for the entropy of a memoryless information source (i.e. a source that produces independent symbols) by posing three axioms that an information measure should satisfy. This approach to deriving an expression for the entropy of an information source is known as the axiomatic approach. It was proved that the expression for the entropy of an information source, as derived by Shannon, is the only expression that satisfies the three axioms.

A more pragmatic approach to deriving an expression for the information content of a source involves the derivation of certain source statistics in which the expression for source entropy appears, and interpreting this expression as an information measure. The pragmatic approach does not rely on the definition of any axioms, and produces the same expression for source entropy as the axiomatic approach.

Shannon [12] proved that the asymptotic equipartition property (AEP)[1] applies to memoryless information sources. The AEP of memoryless sources is equivalent to the weak law of large numbers for independent and identically distributed (i.i.d.) random variables defined over a finite number of positive values. The property states that each sequence of n symbols from a memoryless source, where n is sufficiently large, may be placed within one of two sets, namely

[1]Experts in the field of statistical mechanics are familiar with the AEP. McMillan [13] was the first to use the term AEP in the context of information sources.

1. an atypical set, with elements having a total probability of occurrence that is negligible, or

2. a typical set, with approximately 2^{nH} distinct elements that are equiprobable, each with a probability of occurrence approximately equal to 2^{-nH}.

Shannon [12] proved that the AEP holds for memoryless sources, and noticed that it applies to stationary Markov sources as well. Khinchin [14] proved that the AEP applies to Markov sources in 1953. McMillan [13] subsequently proved that the AEP applies to any stationary ergodic source with finite alphabet — this theorem became known as the Shannon–McMillan theorem.

A fixed–to–fixed length source code for memoryless sources may be derived using the AEP as follows. Let each n–symbol sequence that belongs to the typical set be assigned a distinct codeword of $nH + \delta$ bits (where δ is an arbitrarily small positive constant), and let all sequences belonging to the atypical set be disregarded. This source code may be used to correctly encode and decode sequences from a memoryless source with a probability that approaches unity as the length of the sequences is increased.

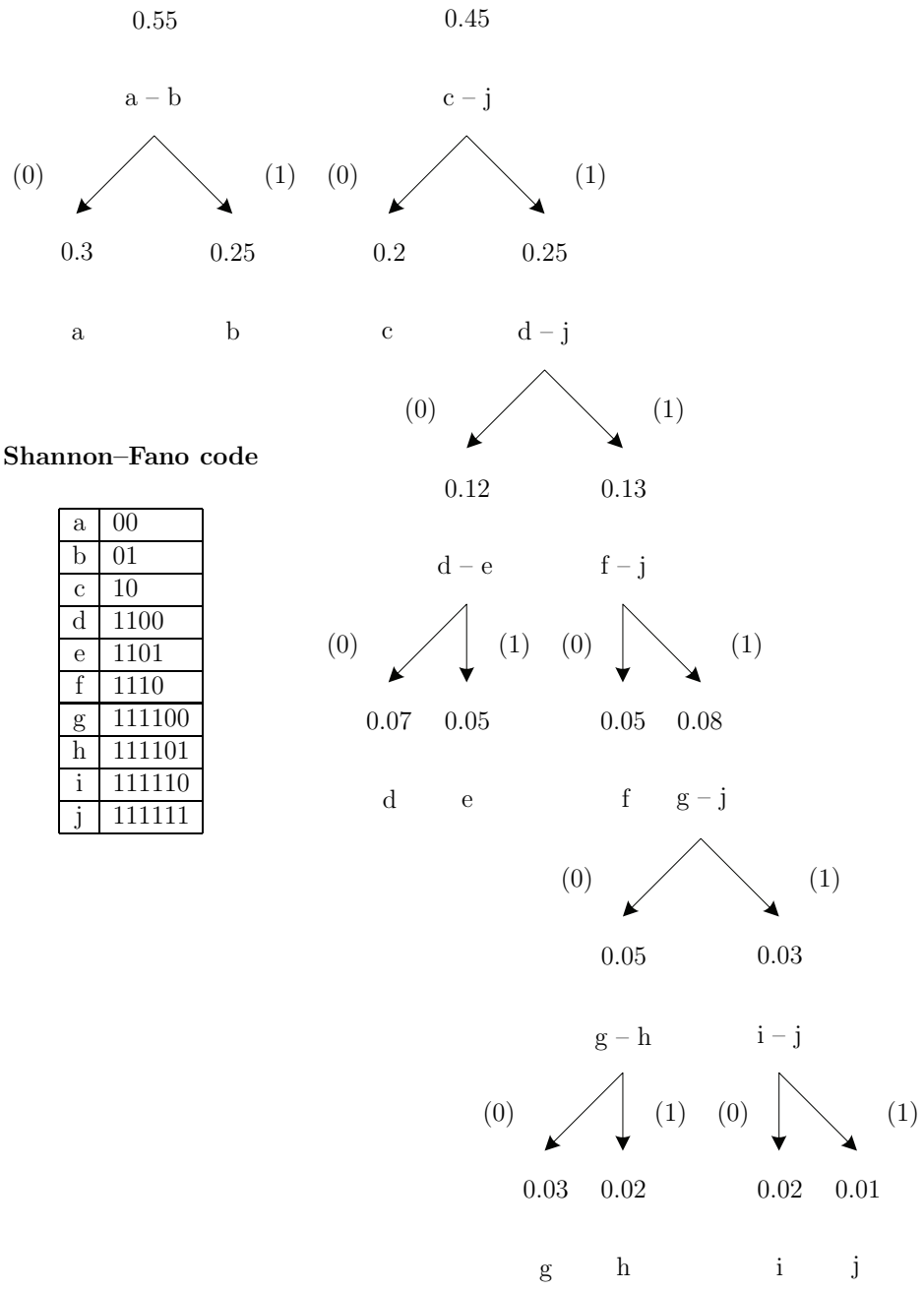
The fixed–to–fixed length source code that was derived using the AEP is not optimal in terms of its probability of correctly encoding and decoding finite–length source sequences. An optimal fixed–to–fixed length source code assigns each of the most probable source sequences a unique codeword, and disregards the remaining sequences. Shannon [12] proved that the probability of correctly encoding and decoding sequences from a source using any fixed–to–fixed length source code with a code rate less than the entropy of the source tends to zero asymptotically. This theorem is known as the strong converse source coding theorem.

It is impossible to successfully encode and decode each finite–length sequence from a source using any nontrivial fixed–to–fixed length code, as each source sequence cannot be assigned a unique codeword. An alternative source code assigns variable–length codewords to fixed–length source sequences — this source code is referred to as a fixed–to–variable length source code. Source sequences may be successfully encoded and decoded with a probability of unity using a proper fixed–to–variable length source code. The average codeword length (or equivalently, the average code rate) of a fixed–to–variable length source code is of interest.

Shannon [12] proved the existence of fixed–to–variable length source codes having codewords with average lengths that exceed the entropy of an information source by no more than one bit. The minimum average codeword length of noiseless (i.e. without error) source codes was not established in Shannon's paper of 1948 [12], however. The construction of an optimal fixed–to–variable length source code (i.e. a code with a minimum average codeword length) for an arbitrary symbol probability distribution also remained an unsolved problem.

Shannon and Fano independently developed the same fixed–to–variable length source code for symbols with arbitrary probability distributions [15, 16]. This source code became known as the Shannon–Fano code. The construction of a Shannon–Fano code for symbols with an arbitrary distribution proceeds as follows (refer to figure 2.1 on page 6). The symbols are initially sorted in a nonincreasing fashion according to their probabilities of occurrence. The ordered list of symbols is split in such a manner that

a	b	c	d	e	f	g	h	i	j
0.3	0.25	0.2	0.07	0.05	0.05	0.03	0.02	0.02	0.01



Shannon–Fano code

a	00
b	01
c	10
d	1100
e	1101
f	1110
g	111100
h	111101
i	111110
j	111111

Figure 2.1: The construction of a Shannon–Fano code for the symbol alphabet that appears at the top of the figure. Unbracketed numbers represent probabilities of occurrence, and bracketed numbers represent the bits assigned to symbol codewords. The Shannon–Fano code appears at the left–hand side of the figure.

the sums of the symbol probabilities of the two sublists are as equal as possible. The codewords of all symbols in the first sublist are appended with a zero–valued bit, and the codewords of all symbols in the second sublist are appended with a nonzero–valued bit. The two sublists are recursively sorted and divided in an identical, independent fashion. The recursive sorting and division of the sublists terminate upon obtaining sublists that each contain only a single symbol. The Shannon–Fano code that is obtained in this manner is a prefix code (i.e. a code with no codeword that is a prefix of any other codeword), and is uniquely decodable.

The Shannon–Fano code is not guaranteed to be optimal in terms of having the shortest average codeword length for an arbitrary symbol probability distribution [16]. Its suboptimality is a result of the manner in which the list (and each sublist) is split. The Shannon–Fano code of the symbol alphabet of figure 2.1 has an average codeword length of 2.66 bits, compared to the source entropy of 2.63 bits. The optimal unextended code for this source has codewords with an average length of 2.66 bits, which proves that the Shannon–Fano code is in some cases optimal.

2.2 Huffman source codes

Huffman [17] introduced a systematic technique for the construction of lossless source codes for information sources with arbitrary symbol distributions. The average codeword lengths of these prefix codes are minimal, but only under the assumption that their codewords must consist of an integer number of bits.

2.2.1 Construction

Huffman [17] derived an algorithm for constructing an optimal fixed–to–variable length source code by observing that a codebook (i.e. a set of codewords) has to satisfy certain requirements in order for it to be considered optimal. The algorithm constructs an optimal codebook in an iterative fashion, and each iterative codebook fulfils the requirements for optimality. This approach guarantees that the final codebook is optimal.

The construction of a Huffman code for the symbol alphabet of figure 2.1 is illustrated in figure 2.2. The algorithm for constructing a binary Huffman code first sorts the symbols in a nonincreasing fashion according to their probabilities of occurrence [17]. The algorithm subsequently (and repeatedly) merges the two least probable symbols into a single composite symbol, and assigns a zero–bit label to one of the merged symbols, and a nonzero–bit label to the other. The composite symbol has a probability of occurrence equal to the sum of the probabilities of the merged symbols. The algorithm repeats its initial steps by sorting the new set of symbols, and merging the two least probable symbols. These steps are repeated until the set contains only a single composite symbol with a probability of occurrence equal to unity.

The merging of the alphabet symbols may be illustrated using a diagram that resembles a tree (refer to figure 2.2). The root of the tree corresponds to the final composite symbol, while the leaves correspond to the original alphabet symbols. This tree diagram may be used to assign codewords to the original symbols. A codeword is

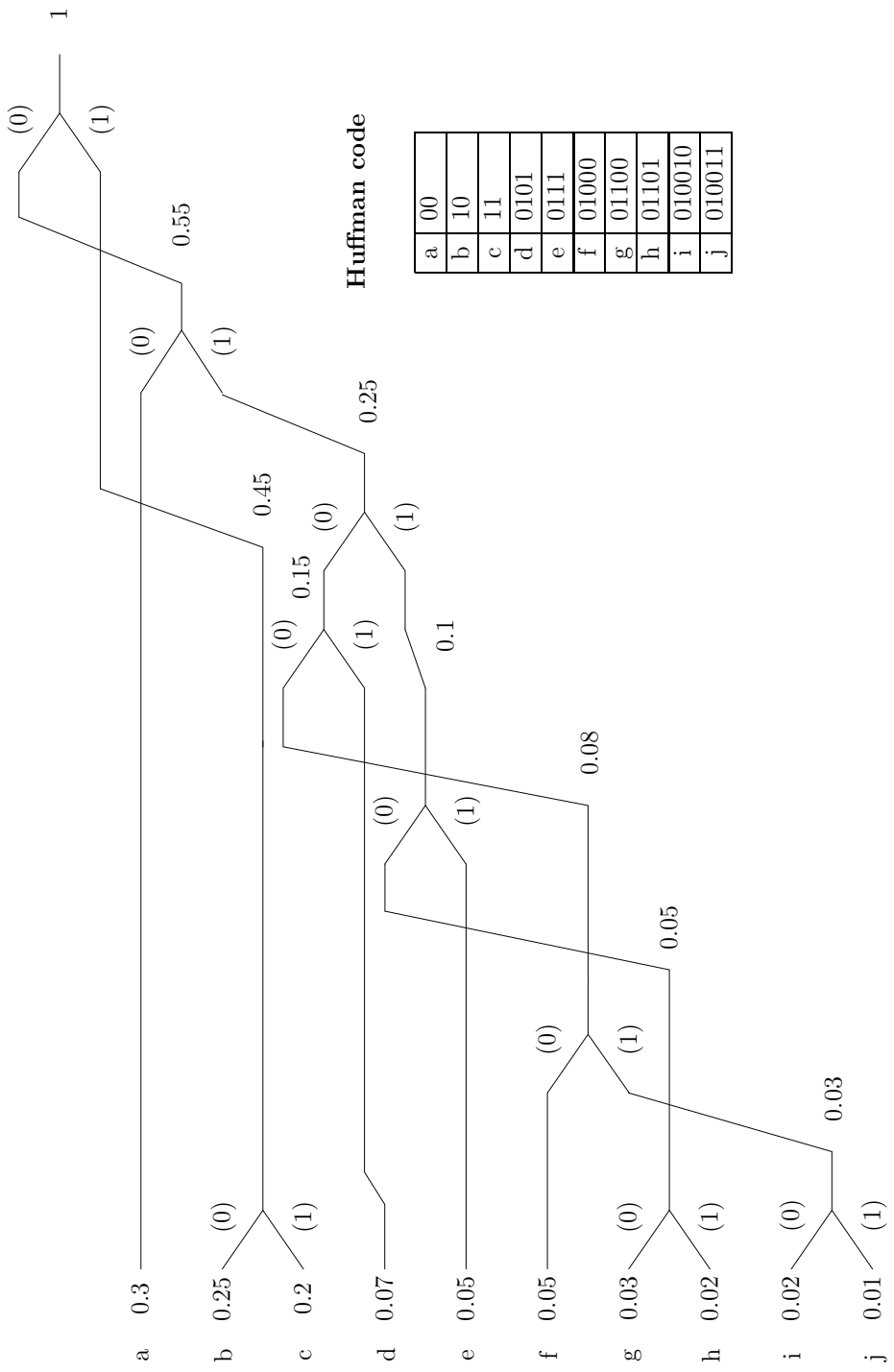


Figure 2.2: The construction of a Huffman code for the symbol alphabet that appears at the bottom of the figure. Unbracketed numbers denote probabilities of occurrence, and bracketed numbers denote codeword bits. The Huffman code appears at the top of the figure.

assigned to an alphabet symbol by traversing the tree from the corresponding leaf to the root, and attaching the bit labels that are encountered on the path as prefixes to the initially empty codeword.

The final codebook has a minimum average length when used to encode symbols from the distribution for which it was designed [17]. The Huffman code for the symbol distribution of figure 2.2 has an average codeword length of 2.66 bits, which exceeds the entropy of the source by only 0.03 bits.

2.2.2 The source encoder and decoder

The source encoder of a Huffman code replaces each source symbol with its codeword from the codebook [17]. The source encoder does not need to delimit each codeword, as Huffman codes are prefix codes (and therefore uniquely decodable). The source encoder does require a complete Huffman codebook in order to encode an arbitrary source symbol, however.

The source decoder processes the sequence of codewords in a bit–by–bit fashion, starting at the first bit of the first codeword. It traverses the tree that is associated with the code, starting at its root and moving towards its leaves. The decoder moves from one composite symbol to the next as each bit is processed. It selects the correct path by matching the bit labels of the composite symbols on the path to the bits of the codeword. Upon reaching a leaf node of the tree, the source decoder produces the source symbol associated with that leaf as output. The source decoder proceeds by restarting at the root of the tree, as the coded bits that follow those already processed correspond to a new codeword.

2.2.3 Performance

It was proved that the average codeword length of a Huffman code is bounded as [16]

$$H(X) \leq L(X) < H(X) + 1, \tag{2.1}$$

where $L(X)$ is the average codeword length of the Huffman code, X is a random source symbol with the distribution for which the code was constructed, and $H(X)$ is the source entropy. Gallager [18] refined the upper bound on the average codeword length as

$$L(X) < H(X) + p_{\max} + \sigma, \tag{2.2}$$

where $p_{\max}$ is the largest probability of occurrence associated with any of the source symbols, and σ is a constant that is approximately equal to 0.08607.

2.2.4 Applications and variations

Huffman codes are used in modern standards such as the CCITT[2] T.x fax recommendations [19] and the JPEG standard [20], despite the fact that these codes were introduced over 50 years ago. Several variations of the original Huffman codes have also been proposed since their introduction in 1952. Some of these variations are summarized in what follows.

[2]The CCITT changed its name to the ITU–T in 1993 [19].

2.2.4.1 Q–ary Huffman codes

The codewords of a Q–ary Huffman code consist of symbols from an alphabet of Q distinct symbols. The average codeword lengths of Q–ary Huffman codes are minimal. These codes are suitable for implementation on devices that use more than two distinct voltage levels on their input and output interfaces. The construction of these codes is summarized in references [4, 16].

2.2.4.2 Huffman–prefixed codes

Huffman–prefixed codes are appropriate codes for sources with very large symbol alphabets [16]. These source codes are constructed by first partitioning the source alphabet into equivalence classes, so that symbols with roughly the same probability of occurrence belong to the same class. Each class has a probability that is equal to the sum of the probabilities of occurrence of its symbols.

An ordinary Huffman code is subsequently constructed for the equivalence classes [16]. The codeword for a source symbol is obtained by concatenating a Huffman codeword (the prefix) and an index codeword (the suffix). The Huffman codeword represents the class to which the source symbols belongs, and the index codeword represents the index of the symbol in the class.

2.2.4.3 Length–constrained Huffman codes

A length–constrained Huffman code is an ordinary Huffman code, but with codewords that are limited in length to a certain number of symbols [21–24]. Length–constrained Huffman codes are appropriate in cases where one or more alphabet symbols have very small probabilities of occurrence. These symbols are assigned very long codewords during the construction of an ordinary Huffman code. Excessively long codewords are impractical, as they are typically only decoded after a significant delay (long delays are unacceptable in time–constrained telecommunication and multimedia applications).

2.2.4.4 Extended Huffman codes

Extended Huffman codes are effective when used to encode symbols with a severely biased distribution, symbols from small alphabets, or symbols from sources with memory [16]. Ordinary Huffman codes are typically ineffective when used to encode symbols with these properties. To illustrate this point, one may consider a source that produces binary symbols. The entropy of this source is less than or equal to one bit per symbol. An ordinary Huffman code for this source has an average codeword length of one bit, regardless of the symbol distribution[3]. It follows that the redundancy of the Huffman codewords equals the redundancy of the source.

Extended Huffman codes overcome the limitations of ordinary Huffman codes by assigning codewords to groups of n source symbols, instead of assigning codewords to individual symbols [16]. An extended code associates a unique codeword with each

[3]This statement is true under the condition that none of the alphabet symbols has a zero probability of occurrence.

distinct combination of n source symbols. An extended code has a normalized average codeword length that is bounded as

$$\frac{H(\mathbf{X}^n)}{n} \leq \frac{L(\mathbf{X}^n)}{n} < \frac{H(\mathbf{X}^n)}{n} + \frac{1}{n}, \tag{2.3}$$

where $\mathbf{X}^n$ is a sequence of n random source symbols with the distribution that the code was constructed for. This bound reveals that extended Huffman codes are able to exploit source memory. The upper bound on the normalized average codeword length of an extended Huffman code approaches the entropy rate of the source asymptotically.

It is impractical to derive extended Huffman codes for long sequences of source symbols, as the number of possible source sequences grows exponentially w.r.t. the sequence length [16]. Many of the codewords that are assigned during the construction of an extended Huffman code for longer source sequences are unlikely ever to be used.

2.2.4.5 Adaptive Huffman codes

Adaptive Huffman codes may be used if the source encoder and decoder have no apriori knowledge regarding the distribution of the source symbols [16]. These codes are appropriate for encoding symbols from nonstationary sources (i.e. the symbol distribution changes over time). The source encoder does not require access to the entire source sequence in order to encode its initial symbols using an adaptive Huffman code.

Both the source encoder and decoder of an adaptive Huffman code estimate the symbol distribution according to the source symbols that were previously encoded or decoded [16]. The source encoder and decoder of some adaptive Huffman code implementations maintain identical codebooks that are updated periodically, or when significant changes are observed in the estimate of the symbol distribution.

Rudimentary implementations of adaptive Huffman codes reconstruct the entire Huffman codebook as it is updated [16]. Many of these implementations have an excessively high computational complexity. Practical implementations that modify only certain codewords in the Huffman codebook during each update were proposed in the literature [18,25]. The codebooks of these implementations typically contain only those symbols that were encountered previously in the source sequence. These implementations use an escape symbol to add novel symbols to the codebook — the source encoder encodes the escape symbol if it encounters a novel symbol in the source sequence.

Many papers regarding adaptive Huffman codes were published in the literature, with the more important papers summarized in reference [16]. A number of distinct adaptive Huffman code implementations, as well as several modifications to existing implementations, were proposed. McIntyre et. al. [26] suggested, however, that adaptive Huffman codes only be used to source code long symbol sequences (when compared to the size of the alphabet). This is due to the fact that adaptive source codes only become effective as their empirically–derived estimates of the symbol distribution become accurate.

2.2.4.6 Efficient implementations

Both memory–efficient codebook construction algorithms [23,27,28], as well as decoding algorithms with low computational complexity [29–32], were proposed in the literature.

2.3 The Kraft inequality

Kraft [33] derived an inequality regarding the codewords of any variable–length prefix code. Consider any binary prefix code with m codewords of lengths $l(a_i)$, where a_i is the ith of m alphabet symbols. Kraft proved that the inequality

$$\sum_{i=1}^{m} 2^{-l(a_i)} \leq 1 \tag{2.4}$$

is satisfied by the binary prefix code. This inequality became known as Kraft's inequality[4]. A counterpart to Kraft's theorem regarding the codewords of a prefix code concerns the construction of a prefix code [16]. This theorem states that it is always possible to construct a prefix code that has codewords with lengths $l(a_i)$ (where $i = 1, 2, \ldots m$), provided that the codeword lengths satisfy Kraft's inequality.

Several years after Kraft's work, McMillan [34] proved that Kraft's inequality is not only satisfied by the codewords of all prefix codes, but by the codewords of all uniquely decodable source codes as well. This theorem implies that one need only consider prefix codes during the construction of variable–length source codes with minimum average codeword length, as any uniquely decodable nonprefix code may be converted to a prefix code with codewords of identical length.

A lower bound on the minimum average codeword length of any uniquely decodable variable–length source code may be derived using Kraft's inequality [11]. Any code that satisfies Kraft's inequality can not have an average codeword length shorter than the source entropy when used to encode symbols from a memoryless source. As all uniquely decodable variable–length source codes satisfy Kraft's inequality, the lower bound

$$\sum_{i=1}^{m} \Pr(X = a_i) l(a_i) \geq H(X) \tag{2.5}$$

applies to all uniquely decodable variable–length source codes.

2.4 Arithmetic source codes

One major shortcoming of extended Huffman codes is that the source encoder has to construct a Huffman codebook with codewords for all source sequences of a certain length prior to encoding any source sequence [16]. Arithmetic source codes do not have this shortcoming, and are computationally feasible source codes.

2.4.1 History

Arithmetic coding has its roots in Shannon's paper of 1948 [12], in which an 'arithmetic process' of source coding was proposed. This precursor to the arithmetic code has

[4]Some authors refer to equation 2.4 as the Kraft–McMillan inequality.

become known as the Shannon–Fano–Elias code[5]. The construction of a Shannon–Fano–Elias code involves the evaluation of the cumulative distribution function (CDF) of the alphabet symbols at each symbol. The codeword of a source symbol is obtained by expressing the corresponding value of the CDF as a binary number and appropriately truncating it.

The average codeword length of a Shannon–Fano–Elias code for a memoryless source is only slightly longer than the entropy of the source, but it is not necessarily minimal [11]. The computational complexity of the original Shannon–Fano–Elias code, like that of the Huffman code, increases rapidly as the code is extended to multiple alphabet symbols. An original Shannon–Fano–Elias code that is extended to a large number of alphabet symbols is therefore impractical.

Despite early work on improving a recursive implementation of the Shannon–Fano–Elias code [35, 36], it wasn't until 1976 that most of the problems associated with this code, including the issue of finite precision, were resolved [4]. Modern arithmetic codes were independently proposed by Rissanen [37] and Pasco [38] in 1976. A more general arithmetic code, as well as a practical implementation of the code, were proposed in 1979 [39]. A subsequent paper made arithmetic codes popular in the source coding community [40].

2.4.2 The source encoder

Unlike Huffman codes, it is unnecessary to construct a codebook that contains a complete set of codewords in order to encode or decode a source symbol using an arithmetic code [4]. The encoder of an arithmetic code encodes a source sequence by dividing a numeric interval repeatedly into subintervals that are proportional in length to the symbols' probabilities of occurrence. The arithmetic coding of symbols from a memoryless source is illustrated in figure 2.3 on page 14. An alphabet of three symbols (`a`, `b` and `c`), as well as the probabilities of occurrence of the symbols, are provided in this figure.

The source coding of the sequence `baca` is demonstrated in figure 2.3. The source encoder sets the initial numeric interval to $[0, 1)$. This interval is divided into subintervals, and each subinterval is associated with one of the alphabet symbols [4]. The length of each subinterval is directly proportional to the probability of occurrence of the symbol it is associated with. In figure 2.3, the initial interval is divided into the subintervals $[0, 0.6)$, $[0.6, 0.9)$ and $[0.9, 1.0)$ according to the probabilities of occurrence of the alphabet symbols `a`, `b` and `c`.

The source encoder proceeds by selecting the numeric subinterval that is associated with the first source symbol, and dividing this interval into subintervals [4]. Each subinterval is associated with one of the alphabet symbols. The length of each subinterval is again proportional to the probability of occurrence of the symbol it is associated with. In figure 2.3, the source encoder selects the subinterval $[0.6, 0.9)$, as it is associated with the source symbol `b`. It divides this interval into three subintervals, as illustrated in the figure.

[5] According to Verdu [11], Elias was not involved in developing the original arithmetic process of source coding. For the purpose of this thesis, his surname is retained in the name of the code in order to distinguish it from the Shannon–Fano code of section 2.1.

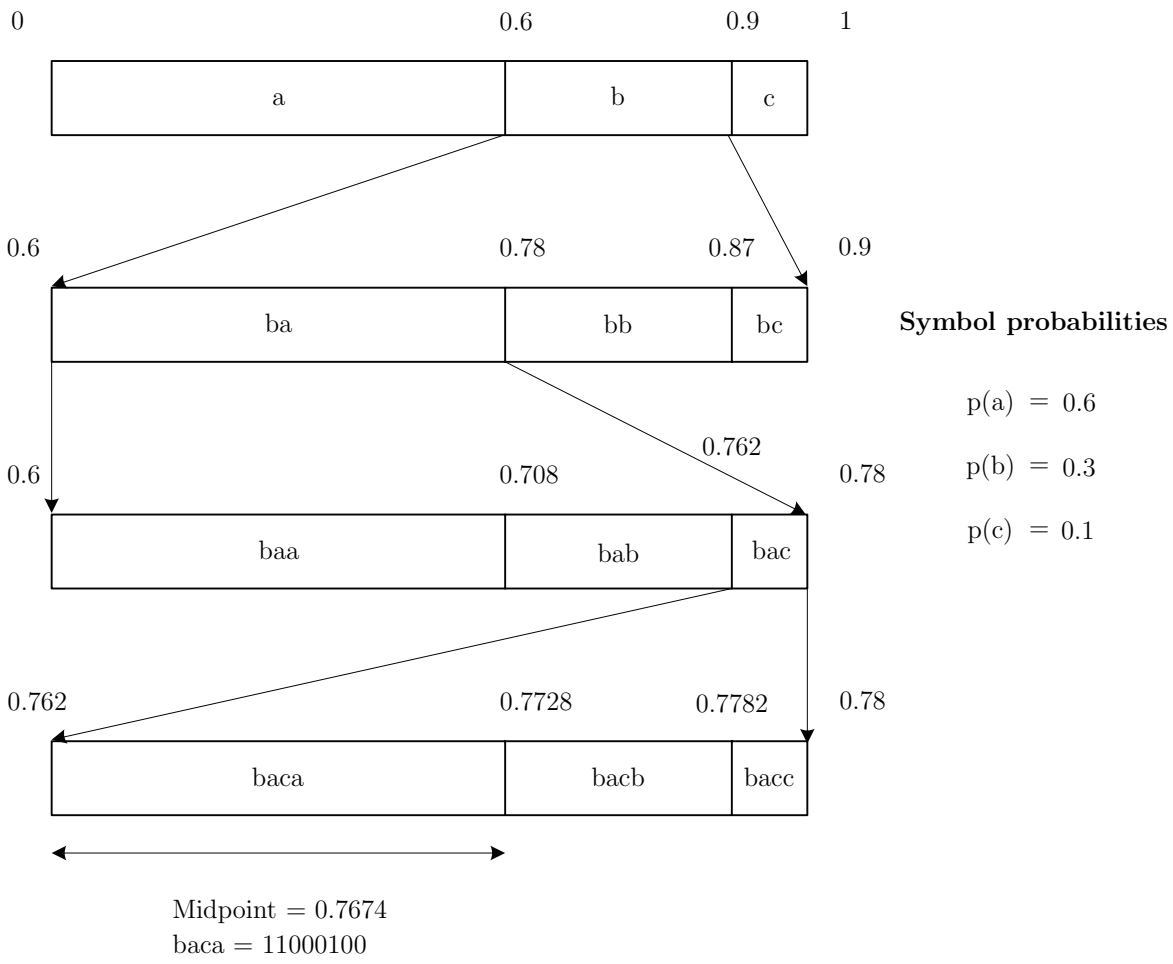


Figure 2.3: Arithmetic coding of the sequence `baca`, as produced by a memoryless source with the symbol alphabet `a`, `b` and `c`. The symbol distribution is provided towards the right–hand side of the figure. The codeword for the sequence `baca` is `11000100`.

Each numeric interval of figure 2.3 is divided according to a first–order probability distribution, as the source symbols are independent from one another. If the source had memory, the intervals may instead have been divided according to the symbol distribution conditioned on those source symbols already encoded [16]. This approach would improve the effectiveness of the source code, as it would enable the source encoder to exploit the source memory.

The process illustrated in figure 2.3 continues with the selection of the numeric interval that is associated with the next symbol in the source sequence, and its division [4]. The selection and division of the intervals repeat until the final source symbol has been considered, and the final interval has been obtained. The source encoder produces a codeword for the source sequence by appropriately truncating the binary representation of any number in the final numeric interval.

A sufficient number of bits must be retained in the codeword in order to guarantee the correct identification of the final interval by the source decoder [4]. If the codeword is obtained by truncating the binary representation of the midpoint between the upper

and lower boundaries of the interval, it is sufficient to retain

$$l(\mathbf{x}^n) = \left\lceil \log_2 \left(\frac{1}{\Pr(\mathbf{x}^n)} \right) \right\rceil + 1 \tag{2.6}$$

bits in the codeword to guarantee the flawless recovery of the final interval. The term $\Pr(\mathbf{x}^n)$ in equation 2.6 denotes the probability of occurrence of the source sequence.

The source encoder obtains the final numeric interval of $[0.762, 0.7728)$ during the source coding of the sequence `baca`, as illustrated in figure 2.3. The midpoint of this interval equals 0.7674. Equation 2.6 implies that a codeword of eight bits is sufficient to guarantee the correct recovery of the final interval and the source sequence. The binary representation of the midpoint, which equals 0.11000100011..., is therefore truncated to eight bits in order to obtain the codeword `11000100` (only the fractional part of the binary number is used).

2.4.3 The source decoder

The decoder of the arithmetic code operates in a similar fashion as the encoder [4]. It first divides the numeric interval $[0, 1)$ in half, and selects one of the subintervals according to the first bit of the codeword. It selects the first subinterval if the first codeword bit equals `0`, and the second subinterval if the first codeword bit equals `1`. The source decoder proceeds by dividing the subinterval that was selected in half, and selecting the next subinterval according to the second bit of the codeword (in the same manner as the first subinterval).

The source decoder continues to select and divide subintervals until it has considered all the bits of the codeword [4]. Let the final subinterval that the source decoder selects be referred to as the codeword interval. After selecting the codeword interval, the source decoder finds the longest symbol sequence with a numeric interval that contains the codeword interval (i.e. the codeword interval lies within the interval associated with the symbol sequence that was found by the source decoder). This symbol sequence is the recovered source sequence.

Consider the numeric intervals that the source decoder selects as it decodes the codeword that was assigned to the source sequence `baca` in figure 2.3. The sequence of intervals are $[0, 1)$, $[0.5, 1)$, $[0.75, 1)$, $[0.75, 0.875)$, $[0.75, 0.8125)$, $[0.75, 0.78125)$, $[0.765625, 0.78125)$, $[0.765625, 0.7734375)$ and $[0.765625, 0.76953125)$. The codeword interval lies within the interval associated with the sequence `baca`, which equals $[0.762, 0.7728)$. The relationship between the intervals may be expressed as

$$[0.765625, 0.76953125) \subset [0.762, 0.7728). \tag{2.7}$$

The source decoder therefore selects the sequence `baca` as the decoded sequence.

It is possible to modify the source decoder to produce the decoded symbols as it calculates successive numeric intervals, instead of producing all the decoded symbols after calculating the codeword interval [4]. The source decoder may produce a decoded symbol once the most recent numeric interval lies within the interval associated with the symbol. It is therefore unnecessary to process the entire codeword prior to recovering the initial symbols of the source sequence. The source encoder may be similarly modified to produce the initial codeword bits prior to considering the entire source sequence.

2.4.4 Performance

If an arithmetic code is used to produce a single codeword for each sequence of n source symbols, its normalized average codeword length is bounded as [4]

$$\frac{H(\mathbf{X}^n)}{n} \leq \frac{L(\mathbf{X}^n)}{n} < \frac{H(\mathbf{X}^n)}{n} + \frac{2}{n}, \tag{2.8}$$

where $\mathbf{X}^n$ is a sequence of n random source symbols with the distribution according to which source sequences are encoded. The upper bound on the normalized average codeword length of an arithmetic code is approximately equal to the upper bound on the normalized average codeword length of an extended Huffman code (refer to equation 2.3). The Huffman code becomes impractical if extended to longer source sequences, however. An arithmetic code does not have this drawback, as its source encoder does not construct the entire codebook with all source sequences of n symbols.

2.4.5 Applications and implementations

Arithmetic coding is arguably the most effective lossless source coding technique that is also practical. Arithmetic codes are used in document compression software [41] as well as command–line compression and archiving software utilities such as `bzip` [42]. Arithmetic codes are also used in applications that involve the source coding of image and video data [43], and are part of several standards [4].

Many adaptive source code implementations use arithmetic codes, as an implementation may perform the source modeling and source coding steps independently from one another if it uses these codes [4]. Arithmetic codes may be used to encode source sequences according to arbitrary probability distributions that change over time. Adaptive arithmetic code implementations need not reconstruct a codebook in order to adapt to a changing symbol distribution [44]. The prediction by partial match algorithm, which is an adaptive, context–based source coding algorithm, uses arithmetic codes [45].

Many papers regarding the implementation of efficient arithmetic codes, as well as solutions to several implementation issues, were published in the literature. Some of these papers are summarized in what follows.

1. The number of bits that are required to represent a numeric interval using a digital computer increases as the interval becomes shorter. As the numeric intervals of arithmetic code implementations become shorter w.r.t. the length of the source sequence that is to be encoded, it follows that the source encoder and decoder require arbitrary–precision arithmetic in order to encode and decode source sequences of any length [4]. This fact complicates the implementation of an arithmetic encoder and decoder on a digital computer, as digital computers cannot represent and manipulate numbers with arbitrary precision. The requirement of arbitrary–precision arithmetic may be eliminated by appropriately rescaling each numeric interval as soon as it becomes too short. An implementation that uses only limited–precision arithmetic is provided in reference [4].

2. In some applications it is necessary to encode a source sequence without the source decoder having apriori knowledge regarding the length of the sequence [16]. Some implementations of the source decoder of an arithmetic code rely on knowledge of the sequence length in order to successfully decode each codeword. An implementation may define an 'end of input' symbol to overcome this problem. The source encoder encodes this symbol at the end of the source sequence. The 'end of input' symbol informs the source decoder that any successive codeword bits belong to a new codeword.

3. The source coding and decoding of a source sequence using an arithmetic code are often illustrated using real number intervals (refer to figure 2.3) [4, 16]. It is possible to implement both the source encoder and decoder of an arithmetic code in such a manner that they use only integer numbers and integer arithmetic. An example of an integer–arithmetic implementation of an arithmetic code is provided in reference [4].

4. Conventional implementations of arithmetic codes multiply several numbers as sequences are encoded and decoded. The frequent multiplication of numbers increases the computational complexity of the source encoder and decoder. This drawback was eliminated with the introduction of multiplication–free arithmetic codes, which are discussed in reference [46].

2.5 The prediction by partial match algorithm

The prediction by partial match (PPM) algorithm is an adaptive, context–based source code implementation that uses arithmetic codes [4, 45]. The algorithm is able to exploit source memory, thereby improving the effectiveness of the source code. It is one of the most effective lossless source code implementations for practical data such as English text [47]. The PPM algorithm was first proposed in reference [45], and has undergone several modifications in order to improve both its effectiveness and efficiency since its introduction — several of the more relevant improvements are summarized in reference [47]. One drawback of PPM implementations with unbounded context order is their high computational complexity, which increases quadratically w.r.t. the length of the source sequence (in the worst case) [47].

The PPM algorithm is effective when used to encode sequences from certain Markov sources. A finite–order Markov source has finite memory, which implies that the probability distribution of each of its source symbols is a function of only a finite number of symbols that precede it in the source sequence [47]. The distribution of a particular letter of English text is strongly influenced by those letters in the same word (and perhaps a few neighbouring words) — distant letters in other sentences typically have little impact on its distribution. It follows that Markov sources are, to a certain degree, suitable for modelling English text. This observation partly explains why the PPM algorithm is able to effectively encode English text.

2.5.1 Source modeling

The source encoder of a PPM implementation processes a source sequence sequentially (i.e. symbol–by–symbol, starting at the first symbol of the sequence) [47]. It constructs a source model according to the frequency counts of the source symbols it encoded previously. The encoder iteratively refines the source model as it encodes consecutive source symbols.

The source model provides an estimate of the probability distribution of the source symbol that follows the most recently processed source symbol [47]. The encoder of the arithmetic code encodes successive source symbols according to the source model. As the source model is constructed according to the frequency counts of the symbols that were previously encoded, the source decoder may construct a source model that is identical to that of the encoder. It follows that the source decoder may recover the source sequence without additional information from the source encoder.

The order–m context of a source symbol x_i is defined as the m–symbol sequence $\mathbf{x}_{i-m}^{i-1}$ that precedes the symbol x_i in the source sequence [47]. The PPM algorithm maintains a list of all contexts, up to some maximum order, that were encountered in the previously encoded source sequence. It records the frequency counts of all symbols that immediately follow each of these contexts. The algorithm derives a probability distribution for the symbol that follows each context according to the symbol frequency counts of that context. PPM implementations may differ w.r.t. the manner in which they derive the symbol distribution associated with each context. Some of the approaches to deriving a symbol distribution from empirical symbol frequency counts that are relevant to the PPM algorithm are summarized in section 2.5.3 on page 19.

In order to source code a symbol, the PPM algorithm selects an appropriate context from its list of contexts, and uses the symbol distribution of that context as it encodes the symbol [47]. The manner in which the PPM algorithm selects an appropriate context is discussed in what follows.

2.5.2 Context selection and context updates

The following summary was adapted from reference [47]. Suppose that the source encoder has to encode the symbol $x_i = \mathtt{a}$. Let the actual context of symbol x_i equal the m symbols that precede it, or $\mathbf{x}_{i-m}^{i-1}$. As the PPM algorithm is unaware of the order of the actual context, it has to select an appropriate context from its list of contexts. The algorithm selects the longest matching context $\mathbf{x}_{i-j_{max}}^{i-1}$ that is present in the list, and considers the frequency counts that it collected for symbols following that context.

There are two possibilities regarding the empirical frequency counts of the symbols that follow the longest matching context. In the first case, the symbol `a` has a nonzero frequency count in the longest matching context. The source encoder encodes the symbol according to the symbol distribution of the longest matching context in this case, and proceeds by updating the empirical frequency counts. In the second case, the symbol `a` has a zero frequency count in the longest matching context. As the source encoder cannot encode the symbol `a` using this context (it has a zero probability of occurring in the longest matching context), it encodes an escape symbol. The escape symbol informs the source decoder that the source symbol did not occur previously in

the current context. An escape symbol is defined in the symbol distribution of each context, and a frequency count is maintained for the escape symbol of each context.

After encoding the escape symbol, the encoder considers the symbol frequency counts that are associated with the second–longest matching context $\mathbf{x}_{i-j_{max}+1}^{i-1}$. It proceeds by encoding the source symbol x_i (where $x_i = \mathtt{a}$) according to the distribution associated with the second–longest matching context, provided that the symbol a has a nonzero frequency count in this context. If the source symbol a has a frequency count of zero in the second–longest matching context, the source encoder encodes a second escape symbol. In the case of the source encoder encoding a second escape symbol, it repeatedly considers shorter contexts until a context is reached in which the source symbol a has a nonzero frequency count, or the zero–order context is reached.

If the source encoder reaches the zero–order context, and finds that the symbol a has a zero frequency count in this context (i.e. the symbol a did not occur previously in the source sequence), it considers the order (-1) context. The order (-1) context has a predefined symbol distribution that includes all symbols of the source alphabet. All symbols of the predefined distribution are equiprobable. The definition of an order (-1) context guarantees that the source encoder of the PPM algorithm will always be able to encode any source symbol, even if it did not occur previously in the source sequence.

After encoding the source symbol a, the source encoder increments the frequency counts of this symbol in all the matching contexts [48]. The manner in which the source encoder updates the frequency counts of the escape symbol depends on the implementation.

2.5.3 Derivation of the context distribution

The PPM algorithm estimates the probability distribution of the symbol that follows each context in its list of contexts [47]. Each estimate is derived according to the frequency counts of the source symbols that were encountered in the context. The selection of an appropriate probability of occurrence for the escape symbol complicates the derivation of each symbol distribution, as there is no canonical method for deriving it [48]. Several methods for deriving the symbol distribution of a context, as well as an appropriate probability of occurrence for its escape symbol, were proposed and assessed. Some of these methods are summarized in what follows. The following summary was adapted from reference [48].

Cleary et. al. [8] proposed a method for estimating the symbol distribution of a context that is derived from Laplace's law of succession. This method assigns the escape symbol ϵ of each context a frequency count of one, regardless of the actual number of times it was encoded. The probability distribution of the symbol X_i, which occurs in the context $\mathbf{s} = \mathbf{x}_{i-j}^{i-1}$, is specified as

$$p(x_i|\mathbf{s}) = \frac{n(x_i, \mathbf{s})}{n(\mathbf{s}) + 1}, \; x_i \neq \epsilon, \tag{2.9}$$

where $n(x_i, \mathbf{s})$ is the frequency count of the alphabet symbol x_i in context $\mathbf{s}$, $n(\mathbf{s})$ is the total frequency count of all symbols in context $\mathbf{s}$, and ϵ denotes the escape symbol.

The probability of the escape symbol is specified as

$$\begin{aligned} p(\epsilon|\mathbf{s}) &= \Pr(X_i = \epsilon|\mathbf{s}) \\ &= \frac{1}{n(\mathbf{s})+1}. \end{aligned} \tag{2.10}$$

This method is known as method A in the literature. The distributions that are obtained using method A are (to a certain degree) inaccurate, as it specifies a constant, nonzero probability of occurrence for the escape symbol. The escape symbol of each context is assigned a nonzero probability of occurrence even if all alphabet symbols were previously encountered in the context, in which case the escape symbol is unnecessary.

Cleary et. al. [8] proposed a second method for defining the symbol distribution of a context. This method is known as method W in the literature. It specifies a more appropriate probability of occurrence for the escape symbol. Method W specifies the symbol distribution of the context $\mathbf{s} = \mathbf{x}_{i-j}^{i-1}$ as

$$p(x_i|\mathbf{s}) = \frac{n(x_i,\mathbf{s})+1}{n(\mathbf{s})+|\mathcal{A}|}, \; x_i \neq \epsilon, \tag{2.11}$$

where $|\mathcal{A}|$ denotes the number of symbols that belong to the source alphabet $\mathcal{A}$, and where $n(x_i,\mathbf{s}) > 0$. If the symbol x_i did not previously occur in context $\mathbf{s}$ (i.e. $n(x_i,\mathbf{s}) = 0$), method W assigns a zero probability of occurrence to the symbol. The probability of occurrence of the escape symbol is specified as

$$\begin{aligned} p(\epsilon|\mathbf{s}) &= \Pr(X_i = \epsilon|\mathbf{s}) \\ &= \frac{|\mathcal{A}| - n_d(\mathbf{s})}{n(\mathbf{s})+|\mathcal{A}|}, \end{aligned} \tag{2.12}$$

where $n_d(\mathbf{s})$ denotes the number of distinct symbols that were encountered in context $\mathbf{s}$. The probability of occurrence of the escape symbol is proportional to the number of distinct alphabet symbols that have not occurred in context $\mathbf{s}$.

Other noteworthy methods for estimating the symbol distribution of a context according to symbol frequency counts are briefly discussed in what follows.

- Cleary et. al. [45] proposed method B for estimating the symbol distribution of a context. PPM implementations that use method B regard an alphabet symbol novel in a certain context if its frequency count in that context is smaller than two.

- Witten et. al. [49] proposed methods P and X for estimating the symbol distribution of a context. These methods use a Poisson process as a model for the symbol occurrences, and derive a context distribution according to this model.

An important observation regarding the estimation of the symbol distribution of a context concerns the exclusion of symbols. Consider the first occurrence of the symbol `a` in a specific context $\mathbf{s}$ of a source sequence. All distinct symbols that previously occurred in the same context would not match the symbol `a`. It follows that the PPM algorithm may exclude all distinct symbols previously encountered in context $\mathbf{s}$ when

deriving a context distribution for the lower–order contexts that are associated with context **s**. Some PPM implementations temporarily rescale the symbol distributions of the lower–order contexts in order to take advantage of this observation.

Moffat [50] investigated the manner in which the PPM algorithm updates symbol frequency counts and proposed a mechanism for improving the effectiveness of the PPM algorithm. This mechanism is known as the update–exclusion mechanism. Moffat stated that updates to the symbol frequency counts should be limited to the contexts that were actually used to encode source symbols, instead of the lower–order contexts in which the symbols were merely observed. The update–exclusion mechanism implements this approach to updating the symbol frequency counts. Symbol distributions that are derived using the update–exclusion mechanism are typically more accurate than symbol distributions that are derived using the conventional approach, and improve the effectiveness of the source code.

2.5.4 Contexts of unbounded order

The sequence of contexts that the PPM algorithm selects from its list as it encodes consecutive symbols of the source sequence should ideally match the true contexts of those symbols [47]. The source encoder is typically unaware of the true contexts of the source symbols. As the source encoder has no knowledge of the true symbol contexts, it encodes each source symbol according to the symbol distribution of the longest context (up to some maximum order) that matches the symbols preceding the source symbol. The maximum context orders[6] of PPM implementations for source coding English text typically do not exceed seven.

The maximum context order of a PPM implementation has a direct impact on its effectiveness [47]. If the maximum context order is too small, the implementation cannot accurately represent the source memory in the source model, which reduces the effectiveness of the source code. If the maximum context order is too large, the source model may be inaccurate when used to encode shorter source sequences. Direct implementations of the PPM algorithm also require significantly more memory to accommodate a greater number of contexts.

PPM* is an implementation of the PPM algorithm that specifies no maximum context order [51]. The PPM* implementation records all symbol contexts that were encountered in the source sequence and maintains a set of symbol frequency counts for each of these contexts. Cleary et. al. [51] observed a rapid increase in the order of those contexts that the PPM* implementation uses as it encodes successive source symbols. It was found that the performance of the PPM* implementation may be improved by resolving this problem.

The aforementioned problem of the PPM* implementation was resolved by modifying its context selection mechanism [47]. Let a symbol context in which only a single symbol appeared be referred to as a deterministic context. A PPM* implementation that uses the modified context selection mechanism encodes a symbol according to the distribution of the shortest deterministic context that matches its preceding symbols.

[6] The name of a PPM implementation often includes its maximum context order (e.g. PPM–5 and PPM–7).

If the source encoder has no matching deterministic context in its list of contexts, it uses the distribution of the longest matching ordinary context as it encodes the symbol. The modified context selection mechanism improves the effectiveness of the PPM* implementation in certain cases [48].

2.5.5 Efficient implementations

Two drawbacks of PPM and PPM* implementations are their high computational complexity and excessive memory requirements [47]. A large fraction of the computational burden of direct PPM implementations involves the search for appropriate contexts, in the list of all recorded contexts, that match the symbols of the source sequence. The substantial memory requirements of direct PPM implementations are a consequence of the rapid increase in the number of possible symbol contexts w.r.t. the maximum context order, as the implementations maintain symbol frequency counts for each context that was encountered. The worst–case memory requirement of direct PPM implementations increases quadratically w.r.t. the length of the source sequence.

Memory–efficient PPM* implementations were proposed in references [51–53]. These implementations store symbol contexts in a context–tree data structure [47, 51]. This data structure represents all the contexts that were encountered in the source sequence as a tree. The zero–order context is represented by the root of the tree, and the first–order contexts are represented by the children of the root. The higher–order contexts are represented by the offspring of the appropriate lower–order nodes.

Each context–tree node with a single descendent may be combined with that descendent in order to reduce the total number of nodes in the tree [47,51]. Context trees that have their nodes combined in this manner are referred to as path–compressed context trees. As the leaves of a context tree correspond to contexts that were observed exactly once in the source sequence, many PPM* implementations maintain a pointer at each leaf of the context tree. Each pointer indicates the position of the corresponding context in the source sequence. A path–compressed context tree with pointers at its leaves has a memory requirement that increases linearly w.r.t. the length of the source sequence.

PPM* implementations that store symbol contexts in a context tree require a computationally efficient algorithm for constructing and expanding it [47]. Bunton [52, 53] proposed the use of a suffix–tree data structure for storing the symbol contexts of a source sequence, as computationally efficient construction and expansion algorithms have been developed for this data structure. A suffix tree is a context tree in which each of the suffixes of the source sequence is represented by a distinct path from the root node of the tree to one of its leaf nodes. McCreight [54] proposed a nonsequential construction algorithm for suffix trees. The computational complexity of this algorithm increases linearly w.r.t. the length of the source sequence.

The source decoder of a PPM* implementation requires sequential construction and expansion algorithms for its suffix tree, because it does not have access to the entire source sequence as it decodes consecutive source symbols [47]. Ukkonen [55] proposed a sequential suffix–tree construction and expansion algorithm with a computational complexity that increases linearly w.r.t. the length of the source sequence. Larsson [56] proposed a PPM* implementation that uses Ukkonen's sequential suffix–tree construc-

tion algorithm. Larsson's PPM* implementation does not have a computational complexity that increases linearly w.r.t. the length of the source sequence, however. This property of Larsson's PPM* implementation is due to the super–linear computational complexity of the PPM* probability estimation routine and context escape mechanism.

Effros [47] proposed a PPM* implementation with a worst–case computational complexity and a worst–case memory requirement that increase linearly w.r.t. the length of the source sequence. This implementation stores symbol contexts in a prefix–tree data structure instead of a suffix–tree data structure. Effros proposed a sequential construction algorithm for prefix trees. The derivation of this algorithm is similar to the derivation of the suffix–tree construction algorithm of McCreight [54]. The computational complexity of Effros' prefix–tree construction algorithm increases linearly w.r.t. the length of the source sequence.

A PPM* implementation that uses a prefix tree instead of a suffix tree may combine each step in adding a new context to the tree with the search for a shorter context in which the most recent source symbol is not novel [47]. The number of calculations that the PPM* implementation of Effros [47] performs is reduced in this manner. Effros' PPM* implementation also limits the number of times that the source encoder may move to a shorter deterministic context as it searches for an appropriate context to use when it encodes a symbol. It was empirically verified that this property of Effros' PPM* implementation reduces its effectiveness by a marginal degree.

2.6 Universal source codes

Following the introduction of Huffman codes in 1952 [17], several researchers investigated the source coding of sequences with unknown distributions (i.e. the source encoder and decoder are unaware of the true distribution of the source sequences). Kolmogorov [11, 57] assigned the term 'universal' to any source code of this type — more precise definitions of universal source codes were to follow in later publications. Universal source codes are of great practical interest, as prior statistical knowledge of a source is often not available in practice. Examples of universal source code implementations include command–line compression and archiving utilities such as `gzip` [58].

The following sections cover two source models that are relevant to typical universal source codes, as well as early publications regarding universal source codes. These sections are followed by a summary of several mathematical definitions and theorems regarding universal source codes. The final section contains a summary of universal Lempel–Ziv source codes.

2.6.1 Composite sources

The source encoder and decoder of a universal source code do not have apriori knowledge of one or more of the source parameters, or in some cases the source type [59]. The goal of the source encoder remains the minimization of the average number of bits that it requires to represent a source sequence without any distortion. A starting point to deriving an effective universal source code is to construct a source model that incorporates the encoder's lack of knowledge regarding the source. Several source models

may be constructed, depending on the encoder's knowledge of the source. Two source models are considered in what follows.

The source encoder may have knowledge of the source type (or class), but not of the values of one or more of its parameters [59]. As an example, consider a stationary binary memoryless source with parameter $\theta = \Pr(X_i = 1)$. A source encoder that is unaware of the value of the parameter may model it as a random variable Θ. The random variable Θ has a distribution that may be known or unknown. This model of the source is known as a composite source model, as the value of the source parameter is 'randomly selected' prior to the source generating the source sequence.

Another source model that incorporates the encoder's uncertainty regarding the source involves the definition of a set of possible sources [59]. The actual source that produces the source sequence is randomly drawn from the set of sources. The source encoder may model the overall source as having a random parameter Ω that represents the actual source that was selected. As an example, consider a set that contains three sources:

1. a stationary binary memoryless source with parameter $\theta = 0.25$,
2. a stationary two–state Markov source with a known transition probability matrix $\mathbf{S}$, and
3. a stationary block–independent source with a fully specified n–dimensional probability distribution $p(x_1, x_2, \ldots x_n)$ for the symbols in each block.

In this example, the value of the random variable Ω is drawn from the set $\{1, 2, 3\}$ according to a known or unknown distribution (each of the set elements corresponds to one of the sources). This model of the source is a composite source model, as the actual source is randomly selected prior to it generating the source sequence.

2.6.2 Early universal codes

Early publications concerning lossless universal source codes appeared between 1965 and 1973 [59]. These papers proved that universal source codes exist for certain source models[7]. Some of these papers are summarized in what follows.

Lynch [5] proposed a universal source code that may be used to encode the indices (or locations) of nonredundant samples in a source sequence. This code is an example of an enumerative source code. Enumerative source codes are summarized in chapter 4 on page 100. Lynch's enumerative source code is summarized in section 4.1.1 on page 101.

Davisson [6] provided a simple analytic basis for the source code proposed by Lynch [5]. Davisson used Lynch's code in a source code for sequences that contain both redundant and nonredundant samples. In order to investigate the performance

[7] These proofs were of a constructive nature. In a constructive proof, the author constructs a source code and proves that it is asymptotically optimal in a certain sense when used to encode sequences from the composite source. The first author that proved general theorems regarding the existence of universal source codes by using concepts from information theory was Davisson [59].

of this source code, Davisson defined a source that produces redundant and nonredundant samples. The redundant and nonredundant samples of this source have fixed probabilities of occurrence, and all samples are statistically independent from one another. Davisson proved that the normalized average codeword length of this source code approaches the entropy of the source in an asymptotic fashion (i.e. as the length of the source sequence tends to infinity). The source code requires no prior knowledge regarding the probabilities of occurrence of the redundant and nonredundant samples. Davisson's enumerative source code is summarized in section 4.1.2 on page 107.

Schalkwijk [60] proposed a universal, variable–to–fixed length enumerative source code. This source code is related to the source codes that were proposed by Lynch and Davisson [5, 6]. Schalkwijk proved that the variable–to–fixed length source code, when used to encode bits from a stationary binary memoryless source, has an average code rate that asymptotically approaches the source entropy. The source code requires no prior knowledge of the source parameter. This source code is summarized in section 4.1.3 on page 108.

Cover [7] proposed a general approach to constructing enumerative source codes. Cover used this approach to construct a fixed–to–variable length source code for sequences from stationary first–order binary Markov sources. The encoder of this source code first encodes the frequency counts of the consecutive pairs of bits that were encountered in the source sequence. It proceeds by encoding the index at which the source sequence would appear if it were present in an ordered list of all bit sequences with the same length and the same frequency counts of bit pairs as the source sequence. Cover proved that this source code is universal by proving that its normalized average codeword length asymptotically approaches the entropy of a first–order Markov source. The source code requires no prior knowledge of the source parameters. It is summarized in section 4.1.4 on page 112.

Shtarkov et. al. [61] proposed a universal source code that is similar to the enumerative code that was proposed by Cover [7]. Shtarkov et. al. proved that this code is universal if used to encode sequences from any stationary finite–order, finite–alphabet Markov source. The normalized average redundancy of the codewords of this source code is proportional to $\log(n)/n$ bits per symbol, where n is the length of the source sequence.

Ziv [62] considered fixed–to–fixed length universal source codes. Ziv used the probability of incorrectly recovering the original source sequence from a codeword as a performance measure for determining whether a fixed–to–fixed length source code is universal. A universal fixed–to–fixed length source code for discrete–time finite–alphabet sources was proposed. The error probability of this source code approaches zero asymptotically, provided that the code rate exceeds the entropy of the source.

2.6.3 Universal coding theorems and performance measures

Davisson [59] laid the foundation for future work concerning universal source codes by publishing an extensive paper regarding the subject. Davisson proved several theorems regarding universal source codes by using concepts from information theory. He unified the constructive techniques of previous authors (as summarized in section 2.6.2) into a general, theoretical framework. Several of the theorems that Davisson proposed are

summarized in this section.

The definition of a universal source code is relevant to the theorems that are presented in this section. Davisson [59] defined a universal source code as a code that

1. does not require access to past or future sequences from the source as its encoder is used to encode a source sequence, and

2. meets a certain performance specification asymptotically (w.r.t. the length of the source sequence).

Davisson [59] introduced several performance measures, as well as conditions regarding the existence of universal source codes (where universality is defined according to the performance measures). In the case of fixed–to–variable length source codes, the performance measure that Davisson's definition of universality refers to is ordinarily selected as the redundancy of the code. If the redundancy of a fixed–to–variable length source code approaches zero asymptotically, it is considered universal.

The redundancy of a source code has more than one mathematical definition. Davisson [59] proposed conditions that source codes have to satisfy in order to be universal w.r.t. each definition of redundancy. The definitions of redundancy that Davisson considered, as well as theorems regarding universality w.r.t. these definitions, are provided in what follows.

2.6.3.1 Minimum average redundancy

Consider a source code $\mathscr{C}$ that is used to encode an n–symbol random sequence $\mathbf{X}^n$ from a source with an unknown parameter Θ [59]. Let the conditional average codeword length of the source code be defined as

$$L_{\mathscr{C}}(\mathbf{X}^n|\theta) = \sum_{\mathbf{x}^n \in \mathcal{A}^n} l_{\mathscr{C}}(\mathbf{x}^n)\Pr(\mathbf{x}^n|\theta), \tag{2.13}$$

where $l_{\mathscr{C}}(\mathbf{x}^n)$ is the length of the codeword assigned to source sequence $\mathbf{x}^n$, $\mathcal{A}$ is the symbol alphabet, and $\Pr(\mathbf{x}^n|\theta)$ is the probability of occurrence of the source sequence $\mathbf{x}^n$, conditioned on the source parameter θ. The normalized redundancy of the source code $\mathscr{C}$, conditioned on the source parameter θ, is defined as

$$R'_{\mathscr{C}}(\mathbf{X}^n, \theta) = \frac{1}{n}\left[L_{\mathscr{C}}(\mathbf{X}^n|\theta) - H(\mathbf{X}^n|\theta)\right], \tag{2.14}$$

where $H(\mathbf{X}^n|\theta)$ is the conditional entropy of the source. The normalized minimum average redundancy of any sequence of source codes that is used to encode symbol sequences from a composite source with a parameter density function

$$w(\theta) = \Pr(\Theta = \theta) \tag{2.15}$$

is defined as

$$R'_{min,ave} = \lim_{n\to\infty} \inf_{\mathscr{C}\in\mathcal{C}^n} \int_{\Lambda} R'_{\mathscr{C}}(\mathbf{X}^n, \theta) w(\theta) d\theta, \tag{2.16}$$

where $\mathcal{C}^n$ is the set of all uniquely decodable fixed–to–variable length source codes (for sequences of n symbols), and Λ is the set of all possible values that the source parameter θ may assume. If the normalized minimum average redundancy associated with source coding sequences from a certain source is zero, a sequence of source codes that attains this limit (i.e. with a zero normalized minimum average redundancy) is known collectively as a weighted universal source code.

Davisson [59] proved that a necessary and sufficient condition for the existence of weighted universal source codes is that the normalized average mutual information between $\mathbf{X}^n$ and Θ tends to zero asymptotically. This condition may be expressed as

$$\begin{aligned} \lim_{n\to\infty} \frac{1}{n} I(\mathbf{X}^n;\Theta) &= \lim_{n\to\infty} \frac{1}{n}\Big[H(\mathbf{X}^n) - H(\mathbf{X}^n|\Theta)\Big] \\ &= 0. \end{aligned} \tag{2.17}$$

If a weighted universal code for sequences from a certain source exists, it may be obtained by constructing a Huffman code for sequences with the so–called 'mixture' distribution. This distribution is defined as

$$\Pr(\mathbf{x}^n) = \int_\Lambda \Pr(\mathbf{x}^n|\theta) w(\theta) d\theta. \tag{2.18}$$

2.6.3.2 Strong minimax redundancy

The normalized strong minimax redundancy associated with source coding sequences from a composite source (with parameter Θ) is defined as [59]

$$R'_s = \lim_{n\to\infty} \inf_{\mathscr{C}\in\mathcal{C}^n} \sup_{\theta\in\Lambda} R'_{\mathscr{C}}(\mathbf{X}^n,\theta), \tag{2.19}$$

where $R'_{\mathscr{C}}(\mathbf{X}^n,\theta)$ is defined in equation 2.14. If the normalized strong minimax redundancy associated with source coding sequences from a certain source is zero, a sequence of source codes that attains this limit is collectively referred to as a strong minimax universal source code.

Davisson [59] derived a necessary and sufficient condition for the existence of strong minimax universal source codes for a source. This condition is the existence of a sequence of probability mass functions $\{q(\mathbf{x}^n)\}_{n=1}^{\infty}$ such that the normalized Kullback–Leibler divergence between $\Pr(\mathbf{x}^n|\theta)$ and $q(\mathbf{x}^n)$ tends to zero asymptotically and uniformly over $\theta \in \Lambda$. This condition may be expressed as

$$\begin{aligned} \lim_{n\to\infty} \frac{1}{n} \sup_{\theta\in\Lambda} D(\Pr(\mathbf{x}^n|\theta)||q(\mathbf{x}^n)) &= \lim_{n\to\infty} \frac{1}{n} \sup_{\theta\in\Lambda} \sum_{\mathbf{x}^n\in\mathcal{A}^n} \Pr(\mathbf{x}^n|\theta) \log\left[\frac{\Pr(\mathbf{x}^n|\theta)}{q(\mathbf{x}^n)}\right] \\ &= 0. \end{aligned} \tag{2.20}$$

Uniform convergence implies that for any $\epsilon > 0$, there exists some n_0 such that, for any $n > n_0$ and any θ,

$$\frac{1}{n} D(\Pr(\mathbf{x}^n|\theta)||q(\mathbf{x}^n)) < \epsilon. \tag{2.21}$$

Uniform convergence therefore implies that the choice of n_0 is independent from the parameter θ.

If $q(\mathbf{x}^n)$ is selected as the mixture distribution $\Pr(\mathbf{x}^n)$ of equation 2.18, a sufficient condition for the existence of strong minimax universal source codes may be derived [59]. The sufficient condition is that the normalized average mutual information between $\mathbf{X}^n$ and Θ tends to zero asymptotically and uniformly over $\theta \in \Lambda$. This condition may be expressed as

$$\lim_{n\to\infty} \frac{1}{n} \sup_{\theta\in\Lambda} I(\mathbf{X}^n;\theta) = 0. \tag{2.22}$$

2.6.3.3 Weak minimax redundancy

The normalized weak minimax redundancy associated with source coding sequences from a composite source (with parameter Θ) is defined as [59]

$$R'_w = \inf_{\mathscr{C}\in\mathcal{C}} \sup_{\theta\in\Lambda} \lim_{n\to\infty} R'_{\mathscr{C}}(\mathbf{X}^n,\theta). \tag{2.23}$$

If the normalized weak minimax redundancy associated with source coding sequences from a certain source is zero, a source code that attains this limit is referred to as a weak minimax universal source code.

Davisson [59] proved that the necessary and sufficient condition for the existence of strong minimax universal source codes is also a necessary and sufficient condition for the existence of weak minimax universal source codes, except that the convergence of the normalized Kullback–Leibler divergence to zero need not be uniform over $\theta \in \Lambda$. In the case of weak minimax universal source codes, the convergence of the normalized Kullback–Leibler divergence need only be pointwise. Pointwise convergence implies that, for any $\epsilon > 0$ and any θ, there exists some $n_0(\theta)$ such that, for any $n > n_0(\theta)$, equation 2.21 is satisfied. Pointwise convergence therefore implies that the choice of n_0 may be dependent on θ.

2.6.4 Universal coding of composite sources with denumerable parameter values

Davisson [59] proved that weighted universal source codes exist for composite sources with a parameter Θ that may assume only a denumerable number of values. If the parameter Θ satisfies this condition and has a finite number of possible values, both strong and weak minimax universal source codes exist for the source[8]. This theorem implies that all types of universal source codes considered up to this point exist for a composite source that randomly selects the actual source from a finite set of sources.

2.6.5 Lempel–Ziv source codes

Ziv et. al. [63, 64] proposed the first and second universal Lempel–Ziv source codes in 1977 and 1978, respectively. Ziv et. al. proved that the encoders of the Lempel–Ziv source codes, when used to encode sequences from ergodic sources, produce sequences

[8]These statements assume that the source satisfies other conditions over and above those stated. The interested reader is referred to reference [59] for details regarding these conditions.

with an average length that asymptotically approaches the source entropy [16]. These source codes are therefore universal.

Lempel–Ziv source codes are dictionary–based source codes [16]. A dictionary–based source code maintains a dictionary (i.e. a list) of sequences that are known as phrases, as well as a codeword for each phrase. The encoder of a Lempel–Ziv source code encodes a sequence by progressively dividing it into phrases that correspond to the phrases of the dictionary. The division of the source sequence into phrases is referred to as parsing. The source encoder progressively encodes the phrases of the source sequence by replacing each phrase with its codeword.

The source decoder of a Lempel–Ziv source code requires access to the same dictionary as the encoder in order to correctly decode successive codewords [16]. This requirement forces the source encoder to consider only those source symbols that it encoded previously when it determines how dictionary phrases and codewords should be updated. The source decoder is able to correctly update its dictionary and successfully decode the sequence of codewords by remaining in step with the source encoder[9].

Dictionary–based source codes vary regarding the manner in which their dictionaries are updated and parsing is performed [16]. The Lempel–Ziv source codes parse source sequences from the start of the sequence to the end of the sequence. The encoder of a Lempel–Ziv source code updates its dictionary with new phrases as parsing proceeds, and considers those symbols encoded previously as it formulates changes to the dictionary. Dictionaries that are updated in this manner are referred to as dynamic dictionaries.

One favourable property of Lempel–Ziv source codes is the low computational complexity of their implementations [16]. The computational complexity of typical Lempel–Ziv source code implementations increases linearly w.r.t. the length of the source sequence. This property of Lempel–Ziv source code implementations led to their widespread use in applications that require the rapid source coding and decoding of sequences. The Lempel–Ziv source codes are widely used in compression and archiving software utilities such as `winzip` [65] and `gzip` [58].

The universal Lempel–Ziv source codes of 1977 [63] and 1978 [64] are summarized in what follows.

2.6.5.1 Lempel–Ziv 1977

Ziv [63] proposed their first dictionary–based universal source code in 1977. This source code is commonly abbreviated as LZ77 in the literature [16]. The first implementations of the LZ77 algorithm had a computational complexity that increased quadratically w.r.t. the length of the source sequence. Rodeh et. al. [66] later developed a LZ77 implementation with computational complexity that increases linearly w.r.t. the length of the source sequence.

The source encoder of the LZ77 code updates its dictionary according to the part of the source sequence it parsed [16]. The phrases of the dictionary consist of all substrings of the parsed sequence. The source encoder parses the remainder of a source sequence by finding its longest prefix that matches a phrase in the dictionary. The source encoder

[9]The source encoders of some dictionary–based source codes represent the dictionary as part of the encoded data (i.e. the dictionary is encoded). These source codes are not regarded in this thesis.

replaces each prefix with a codeword from its dictionary. The codeword contains three fields, namely

1. the symbol index, in the parsed section of the source sequence, at which the prefix starts,
2. the length of the prefix, and
3. the first symbol that follows the prefix in the remaining source sequence.

If the dictionary of the source encoder does not contain a phrase that matches a prefix of the remaining source sequence, the index and length fields of the codeword are set to zero [16]. This situation occurs if the first symbol of the remaining source sequence does not appear in the parsed source sequence. The source decoder obtains the novel symbol from the final field of the codeword in this case. The source encoder disregards the first symbol of the remaining source sequence as it continues to parse the sequence.

2.6.5.2 Lempel–Ziv 1978

Ziv et. al. [64] proposed their second dictionary–based universal source code in 1978. This source code is abbreviated as LZ78 in the literature [16]. The LZ78 source code has a straightforward implementation with a computational complexity that increases linearly w.r.t. the length of the source sequence. The normalized redundancy of the sequence produced by the encoder of the LZ78 source code tends to zero more rapidly w.r.t. the source sequence length than the normalized redundancy of the LZ77 source code [67].

The source encoder of the LZ78 source code parses a source sequence in the same manner as the LZ77 source code (i.e. it finds the longest prefix in the remaining source sequence that matches a phrase in its dictionary) [16]. The format of the codewords, as well as the manner in which the source encoder updates its dictionary, differ from the LZ77 source code. The dictionaries of both the encoder and decoder of the LZ78 source code initially contain all distinct single–symbol phrases. Each dictionary phrase is assigned an integer as a codeword, and successive phrases that are added to the dictionary are assigned incrementally larger integers as codewords.

The source encoder encodes the longest matching prefix of the remaining source sequence by replacing it with its codeword [16]. The encoder proceeds by adding the phrase that was replaced to the dictionary, but with the symbol that follows it in the source sequence concatenated at its end. The source encoder leaves the first symbol of the remaining source sequence uncoded as it continues to parse the sequence.

2.6.5.3 Performance and implementations

The Lempel–Ziv source codes were empirically demonstrated as being less effective than source code implementations such as PPM when used to encode practical data [47]. The main advantage of the Lempel–Ziv codes is the low computational complexity of their implementations [16]. Lempel–Ziv source codes are used in both hardware and software applications that require the rapid source coding and decoding of sequences.

The older V.42bis modem standard [68], as well as the more recent V.44 modem standard [69] incorporate variants of the LZ78 source code. Lempel–Ziv source codes are also widely used in compression and archiving software utilities such as `gzip` [58] (which uses a variant of the LZ77 code) and `compress` [70] (which uses a variant of the LZ78 code). The specification of the GIF [71] image file format incorporates a variant of the LZ78 source code for image compression.

CHAPTER 3

The Burrows–Wheeler transform

This chapter presents an in–depth study of the literature that is relevant to the Burrows–Wheeler transform and its application to source coding. The chapter contains a summary of the forward and reverse Burrows–Wheeler transforms, as well as efficient implementations of these algorithms. A discussion of the statistical properties of the forward Burrows–Wheeler transform output is provided, and these properties are expressed mathematically.

The chapter includes a summary of the recency–rank code, which is a transform often used in Burrows–Wheeler–based source code implementations. The chapter concludes with a summary of an elementary Burrows–Wheeler–based source code and its components. Several of the variations of the elementary source code that were proposed in the literature are also considered.

3.1 The Burrows–Wheeler transform

One of the more recent advances in the field of source coding is that of the Burrows–Wheeler transform, or BWT [9]. The purpose of the BWT is to enable the effective source coding of sequences from complex sources using simple source codes, but without redesigning the source codes. This objective is achieved by transforming each 'complex' source sequence (i.e. a sequence of symbols that are statistically dependent on one another) into a 'simpler' sequence (i.e. a sequence of symbols with an asymptotically piecewise independent distribution) prior to encoding it [10].

To illustrate the role of the BWT, suppose that a simple source code is only effective when used to encode i.i.d. symbols with a biased distribution. When the BWT is applied to source sequences with higher–order redundancy (i.e. symbols that are strongly dependent on their preceding contexts), sequences of asymptotically piecewise i.i.d. symbols are frequently produced [10]. The distribution of the symbols in each piecewise segment is typically biased, and each segment of the transformed sequence may be effectively encoded using the simple source code.

The BWT is reversible, and slightly increases the length of the source sequence [10]. This increase is proportional to $\log_2(n)$ bits, where n is the length of the source sequence that is to be encoded. Implementations of both the forward and reverse transforms with computational complexity that increases linearly with respect to the length of the source sequence exist. The forward and reverse transforms are discussed in greater detail in the following sections.

Table 3.1: The first table associated with the forward BWT of the sequence `bananas` (adapted from reference [10]). The first symbol of the original sequence (`b`) is printed in bold.

b	a	n	a	n	a	s
a	n	a	n	a	s	**b**
n	a	n	a	s	**b**	a
a	n	a	s	**b**	a	n
n	a	s	**b**	a	n	a
a	s	**b**	a	n	a	n
s	**b**	a	n	a	n	a

3.1.1 The forward transform

This section contains a description of the forward BWT, as well as a summary of several forward BWT implementations. The distribution of the BWT output sequence is also considered.

3.1.1.1 Description

The forward transform is best described by means of an example. Suppose that the forward transform is to be applied to the source sequence `bananas` [10]. The first step of the forward transform involves the construction of a table (refer to table 3.1). The original sequence is inserted in the first row of the table. The second row of the table equals the first row of the table that is cyclically shifted to the left. Similarly, the third row equals a cyclically left–shifted version of the second row. A table with an equal number of columns and rows is obtained by repeating the shift operation a number of times equal to the length of the source sequence minus one.

Let the following context of a specific symbol be defined as the suffix that follows that symbol in the source sequence. Each row of the table starts with a following context of the original sequence, and the last column contains the symbol that precedes each following context (with the exception of the first row). A prefix of the source sequence follows each following context in the second row to the last row of the table.

The second step of the forward transform consists of lexicographically sorting the rows of the first table relative to one another (refer to table 3.2 on page 34) [10]. During this step, all following contexts that are lexicographically similar are placed in adjacent rows of the table. The final column of the sorted table contains the symbols that precede the sorted contexts — those symbols that precede lexicographically similar following contexts are located adjacent to one another in this column. This column is the output sequence of the forward transform. The forward BWT of the sequence `bananas` is the sequence `bnnsaaa`, as indicated in table 3.2.

An algorithm for reversing the transformed sequence requires additional information regarding the row–sorted table in order to successfully perform the reversal [10]. An integer index is sufficient for this purpose, as demonstrated in section 3.1.2.1. This index equals the number of the row in the sorted table that contains the original source sequence, and is known as the BWT index. In the example, the fourth row of table

Table 3.2: The row–sorted table associated with the forward BWT of the sequence `bananas` (adapted from reference [10]). The output sequence of the forward BWT is printed in bold.

a	n	a	n	a	s	**b**
a	n	a	s	b	a	**n**
a	s	b	a	n	a	**n**
b	a	n	a	n	a	**s**
n	a	n	a	s	b	**a**
n	a	s	b	a	n	**a**
s	b	a	n	a	n	**a**

3.2 contains the original sequence `bananas`. The BWT index of the example therefore equals four — the sequence `bananas` is transformed to the double (`bnnsaaa`,4), or

$$\text{BWT}(\texttt{bananas}) = (\texttt{bnnsaaa}, 4). \tag{3.1}$$

3.1.1.2 Implementation

Computationally efficient implementations of both the forward and reverse Burrows–Wheeler transforms are required in order for these transforms to have any practical use. The computational complexity as well as the memory requirement of each implementation have to be considered. The reverse transform is straightforward to implement [16]. Its implementation has a low computational complexity and requires little memory. It is significantly more difficult to develop an efficient implementation of the forward transform.

A rudimentary implementation of the forward transform might use tables, as demonstrated in the example of section 3.1.1.1. This implementation is impractical due to its memory requirement of $O(n^2)$, where n is the length of the source sequence. A practical implementation has to represent the following contexts of the source sequence in a more efficient manner.

The computationally intensive routine of the forward transform of a sequence is typically the sorting of the following contexts of the sequence [16]. To motivate this statement, consider a digitized colour photograph, which typically contains very long runs of the same pixel in its raw, uncompressed image format. When sorting contexts that contain long runs of identical symbols, a typical sort algorithm frequently has to compare many of these identical symbols in order to resolve comparison ties between the contexts. The large number of symbol comparisons renders the worst–case computational complexity of the algorithm unacceptable. Several solutions for this problem were proposed — these solutions, as well as other techniques for reducing the computational complexity of the forward transform, are discussed in what follows.

Resolution of comparison ties One approach to improving the worst–case computational complexity of the sort algorithm lies in the resolution of comparison ties that occur during the comparison of sequences that contain long runs of identical symbols.

The symbol runs may be replaced with run lengths (i.e. the source sequence is run–length coded) prior to applying the forward transform [10]. This approach shortens the time that the sort algorithm requires to compare similar contexts as it may use the run lengths to resolve the comparison ties. The use of a run–length code reduces the effectiveness of the BWT, as it removes symbol contexts from the source sequence [16].

Another approach to reducing the worst–case computational complexity of the sort algorithm involves the declaration of a maximum sort length [10]. A sort algorithm compares two contexts of the source sequence up to a number of symbols that equals the maximum sort length in this approach. If two contexts are identical up to the maximum sort length, the comparison tie is resolved according to the contexts' position in the original source sequence.

Both the run–length code and the maximum sort length approach reduce the computational complexity of the forward transform, but at the cost of reducing the effectiveness of the BWT and therefore a BWT–based source code [16]. Other techniques for reducing the computational complexity of the forward transform, but which involve no loss in effectiveness, are summarized in what follows.

Efficient sort algorithms A more efficient sort algorithm may be used in the implementation of the forward transform, instead of relying on a mechanism that resolves comparison ties during the execution of a conventional sort algorithm [16]. Several sort algorithms that are relevant to the implementation of the forward transform are presented in what follows. Sadakane [72] wrote an informative summary of these algorithms — some of the material from this summary is used in the following discussion.

The Quicksort algorithm [73], which is arguably one of the most popular sort algorithms currently in use, is fairly efficient when used during the forward transform of most source sequences [9]. Implementations of the Quicksort algorithm also require relatively little memory. The worst–case computational complexity of this algorithm is excessive, however. Alternative sort algorithms with improved worst–case performance should be considered for use in implementations of the forward transform.

The Bentley–Sedgewick algorithm [74] is a practical, general–purpose algorithm for sorting symbol sequences. It may be interpreted as a combination of the Quicksort [73] and the most significant symbol (MSS) radix–sort algorithms. The algorithm sorts a set of sequences in a recursive fashion and with a pivot symbol that is selected at the start of each instance of the algorithm.

The MSSs of the unsorted sequences are compared to the pivot symbol during the initial steps of the Bentley–Sedgewick algorithm [74]. The sequences are subsequently divided into three groups. The groups contain sequences with MSSs that are (respectively) smaller than, equal to, or larger than the pivot symbol. The sequences in each of these groups are sorted recursively according to their MSSs, except for the sequences of the 'equal to' group, which are sorted according to their second–most significant symbols.

The Bentley–Sedgewick algorithm eliminates unnecessary comparisons between sequences with identical prefixes by placing these sequences within the same group [74]. The efficiency of the algorithm depends on the selection of appropriate pivot symbols, however. The Bentley–Sedgewick algorithm is used in the BWT–based compression

and archiving utility `bzip2` [42].

The Andersson–Nilsson algorithm [75] is an iterative radix–sort algorithm for a set of symbol sequences. The algorithm changes the order of the sequences in such a manner that they are sorted w.r.t. their prefixes of i symbols after the ith iteration. The algorithm assigns the sequences to data structures of two types, namely groups and buckets. At the end of the ith iteration of the algorithm, each group contains only sequences that are identical up to their ith MSSs (i.e. the sequences have identical prefixes of i symbols). Each group may therefore be associated with a distinct prefix of i symbols at the end of the ith iteration, and no two groups are associated with the same prefix of this length.

All unsorted sequences are assigned to a single group at the start of the algorithm's first iteration [75]. During the ith iteration of the algorithm, all sequences with ith MSSs that are identical are placed within the same bucket, so that each bucket contains only sequences with the same ith MSSs. The buckets are traversed in lexicographical order w.r.t. the ith MSSs of their sequences. The sequences in a bucket are returned to the back of their groups as the bucket is traversed, thereby sorting the sequences in each group **according to** their ith MSSs. As the $i-1$ MSSs of the sequences in each group are identical, these sequences are sorted **up to** their ith MSSs at the end of the ith iteration of the algorithm.

The algorithm splits each group into subgroups at the end of each iteration [75]. Each of the subgroups contains only sequences with the same prefix of i symbols at the end of the ith iteration. The subgroups are maintained in lexicographic order w.r.t. the prefixes shared by their sequences. The algorithm continues to iterate until each group contains a single sequence. At this point, the sorted set of sequences is obtained by traversing the groups. Sadakane [72] observed that the computational complexity of this algorithm is low, despite its conceptual simplicity.

Suffix–sort algorithms The forward transform may be modified in order to reduce its computational complexity [10]. One such modification involves attaching an 'end of file' (`EOF`) symbol to the end of the source sequence that is to be transformed. The `EOF` symbol may not appear elsewhere in the source sequence, and it is considered to be the last symbol in the lexicographic order of the symbol alphabet.

Attaching an `EOF` symbol to the source sequence has two benefits [10]. The first benefit is the elimination of edge effects. Some contexts of a source sequence may be sorted w.r.t. symbols that transcend the end of the sequence during its forward transform. This property is not beneficial to the transform as it implies that the symbols at the end of the source sequence precede the symbols at the beginning of the sequence. A comparison between any two contexts would terminate at the `EOF` symbol should it be reached. Attaching an `EOF` symbol to the source sequence therefore delimits the contexts at the end of the sequence.

The second benefit of attaching an `EOF` symbol to the source sequence is that it simplifies the forward transform of the sequence [10]. Instead of sorting all the cyclic shifts of the original source sequence, the forward transform of the source sequence that is appended with an `EOF` symbol involves sorting the n distinct suffixes of the original source sequence of length n. An efficient implementation of this forward transform may

be developed, as efficient techniques for sorting the suffixes of a symbol sequence are available.

Baron et. al. [76] proposed three novel suffix–sort algorithms, and used each of these algorithms in an implementation of the forward transform. Each algorithm has a different degree of worst–case computational complexity. The worst–case computational complexity of each algorithm does not increase linearly w.r.t. the length of the source sequence, but at a greater rate. The empirical performance of each algorithm was found to be competitive with the performance of other suffix–sort algorithms, however. The algorithms make use of a memory–efficient suffix–list data structure, and are antisequential (i.e. each source sequence is processed in reverse order).

The Karp–Miller–Rosenberg algorithm [77] locates repeating patterns in a symbol sequence, and may be used to sort the suffixes of a source sequence. The algorithm is similar to the Andersson–Nilsson algorithm (refer to page 36). It exploits a certain property of a suffix set in order to sort it more efficiently.

The Karp–Miller–Rosenberg algorithm assigns the suffixes of a source sequence to groups [77]. It iteratively divides the groups in such a manner that each group contains only suffixes with the same prefix. Only those suffixes that share longer prefixes remain in the same group as the division of the groups proceeds. After dividing a group, its subgroups are arranged in lexicographical order according to the prefixes shared by their suffixes. The groups are divided repeatedly until each group contains only a single suffix. At this point, the sorted suffix set is obtained by traversing the groups in lexicographical order.

The algorithm assigns each subgroup a unique integer that represents its rank among the sorted subgroups at the end of each iteration [77]. It is able to divide the subgroups with greater efficiency by using these integer ranks. Let the suffixes of a source sequence be referred to as its primary suffixes. Instead of dividing the subgroups according to the ith MSSs of their primary suffixes during iteration i, the algorithm divides the subgroups according to the suffixes of their primary suffixes. As the suffix of a primary suffix is another primary suffix, the algorithm compares primary suffixes according to the integer ranks of the subgroups that contain their suffixes during the division of a subgroup.

In contrast to the Andersson–Nilsson algorithm, the Karp–Miller–Rosenberg algorithm sorts the suffixes of each group according to more than one symbol during each iteration [77]. The Karp–Miller–Rosenberg algorithm possesses the so–called 'doubling' property, which states that the suffixes of each group are sorted according to a prefix with a length that doubles with each consecutive iteration of the algorithm. The suffixes of a source sequence may be sorted in $\lceil \log_2(n) \rceil$ iterations of the algorithm, where n is the length of the source sequence.

The Manber–Myers algorithm [78] uses a variant of the Karp–Miller–Rosenberg algorithm [77] to sort a suffix set. It assigns the suffixes of a source sequence to groups, and iteratively divides these groups into smaller subgroups in such a manner that the suffixes of each subgroup share the same prefix. The subgroups are maintained in lexicographic order according to the prefixes shared by their suffixes. The algorithm possesses the doubling property — the suffixes of each group are sorted according to a prefix with a length that doubles with each consecutive iteration of the algorithm.

The Manber–Myers algorithm uses what is referred to as a suffix array to divide each group of suffixes efficiently [78]. Each element of this array is a pointer to the start of a distinct suffix in the source sequence. The elements of the suffix array are maintained in lexicographic order according to the prefixes of the distinct suffixes.

The algorithm traverses the suffix array I from its first element to its last element prior to dividing the groups of suffixes during each iteration [78]. The first element of the suffix array, $I(0)$, points to a suffix that belongs to the group with the smallest lexicographical rank. The suffix that starts at index $I(0) - \lfloor 2^{i-1} \rfloor$ of the source sequence has the smallest prefix of 2^i symbols of all suffixes in its group at the start of the $(i+1)$th iteration. This suffix is moved to the front of its group. The suffix starting at index $I(k) - \lfloor 2^{i-1} \rfloor$ of the source sequence, for each consecutive value of k, has the next smallest prefix of 2^i symbols of all suffixes in its group at the start of the $(i+1)$th iteration. It is inserted behind the last suffix that was moved towards the front of its group during iteration $i + 1$.

The suffixes of each group are sorted up to their prefixes of 2^i symbols after the traversal of the suffix array during the $(i + 1)$th iteration [78]. The algorithm divides each group according to these prefixes, and arranges the subgroups in lexicographic order according to the prefixes of their suffixes. It rearranges the elements of the suffix array in lexicographic order prior to starting the next iteration.

The pointers to the suffixes of a source sequence may be stored in a tree data structure instead of an array data structure [54]. A suffix tree is a data structure with a root node, internal nodes and leaf nodes. The nodes of the tree are connected with edges, and each edge is associated with a subsequence of the source sequence. Each edge is labelled with the subsequence that is associated with it.

Each leaf node of a suffix tree is associated with a distinct suffix of the source sequence — the suffix is equal to the concatenation of the edge labels along the path from the root node of the suffix tree to the leaf node [54]. The internal nodes of the suffix tree are arranged in such a manner that those suffixes with a common prefix share the edges associated with the prefix. The suffix tree of the sequence `abcabcabc$`, where `$` denotes the `EOF` symbol, is presented in figure 3.1.

The suffixes of a source sequence may be sorted by using the suffix tree of the sequence [10]. Let all edges that depart from the same internal node of the suffix tree of a source sequence be ordered lexicographically according to their labels. If the suffix tree is traversed in a depth–first manner from its root node to its leaf nodes, the ith leaf node that is encountered during the traversal is associated with the ith smallest suffix of the sequence.

If all edges that depart from the same internal node of a suffix tree are not ordered lexicographically, the labels of these edges have to be compared in order for the edges to be traversed in a lexicographical order. All comparisons between edges that depart from the same node of the suffix tree are trivial, as each of these edges has a label with a distinct MSS. All edges that depart from the same internal node of the tree have labels with distinct MSSs, as edges with the same parent node and with labels that share the same MSS would have been merged during the construction of the tree.

McCreight [54] proposed an efficient, iterative algorithm for the construction of the suffix tree of a source sequence. The algorithm inserts a distinct suffix of the source

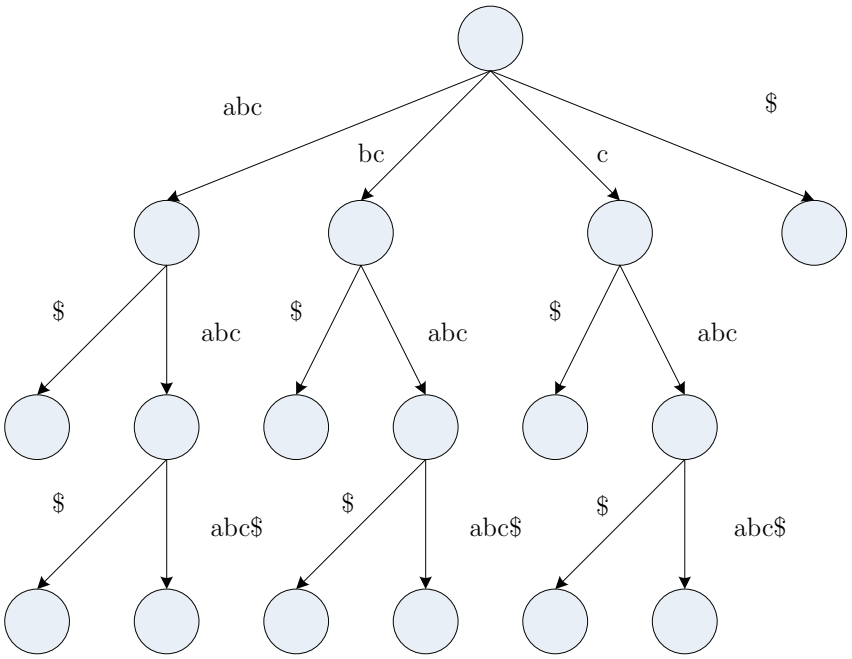


Figure 3.1: The suffix tree of the sequence `abcabcabc$`.

sequence into an initially empty tree during each iteration. The suffixes are inserted into the tree in ascending length order. The algorithm inserts certain suffixes more efficiently into the suffix tree by exploiting certain properties of suffixes with identical prefixes, and by incorporating additional information into the tree structure.

The construction and the depth–first traversal of a suffix tree may be performed with a worst–case computational complexity of $O(n)$ and with $O(n)$ memory, where n is the length of the source sequence [54]. Suffix trees typically require three to five times as much memory as suffix arrays [78]. Both suffix trees and suffix arrays require memory that increases only linearly with respect to the length of the sequence, however.

3.1.1.3 Output distribution

Source codes are designed to effectively encode sequences from a certain probabilistic information source or a certain class of sources. Sources vary from those that are very simple to those that have a more complex structure. A simple source may produce i.i.d. symbols according to some distribution, while some of the more complex sources produce symbols that are statistically dependent on one another.

The performance of a source code that is used to encode sequences from an abstract source provides a general indication of its performance should it be used to encode real–life data. Two of the more complex source models are introduced in this section. These models produce symbols with statistical properties that match those of real–life data such as English text more closely than many of the simpler sources [10].

Finite state machine sources A finite state machine (FSM) source is characterized by a finite set of states $\mathcal{S}$, a finite alphabet $\mathcal{A}$, a next–state function $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, and

the conditional probability mass functions $p(x|\mathbf{s})$, $x \in \mathcal{A}$, $\mathbf{s} \in \mathcal{S}$ [10]. The probability of the source producing the sequence $\mathbf{x}^n = \{x_1, x_2, \ldots x_n\}$, conditioned on the initial state $\mathbf{s}_0^{(t)}$, equals

$$\Pr(\mathbf{x}^n|\mathbf{s}_0^{(t)}) = \prod_{i=1}^{n} p(x_i|\mathbf{s}_{i-1}^{(t)}), \tag{3.2}$$

where $\mathbf{s}_i^{(t)} = f(\mathbf{s}_{i-1}^{(t)}, x_i)$, and $\mathbf{s}_i^{(t)}$ denotes the state in which the source resides during the ith time instant.

Finite–order finite state machine (FSMX) sources constitute a subset of the class of FSM sources [10]. The m most recent symbols produced by an order–m FSMX source uniquely determine the current state of the source. The state set $\mathcal{S}$ of a FSMX source is a minimum suffix set of sequences — this implies that, for every symbol x where $p(x|\mathbf{s}) \neq 0$ with $\mathbf{s} \in \mathcal{S}$, the sequence $\mathbf{s}x$ has exactly one suffix that is present in $\mathcal{S}$. The next–state function of the FSMX source may be expressed as

$$\begin{aligned} \mathbf{s}_i^{(t)} &= f(\mathbf{s}_{i-1}^{(t)}, x_i) \\ &= \text{suf}(\mathbf{s}_{i-1}^{(t)} x_i), \quad 1 \leq i \leq n, \end{aligned} \tag{3.3}$$

where $\text{suf}(\mathbf{s}_{i-1}^{(t)} x_i)$ is the suffix of the concatenation of the sequence $\mathbf{s}_{i-1}^{(t)}$ and the symbol x_i.

Finite–memory and tree sources The transition to the next state of both FSM and FSMX sources depends only on the most recent state and the most recent symbol produced by the source. Certain authors consider this condition to be restrictive and unnecessary [10]. Source models that are not restricted by this condition were proposed in the literature [79, 80]. Two of these source models are discussed in what follows[1].

Effros et. al. [10] made use of what is referred to as finite–memory sources in analyzing BWT–based source codes. This source class may be regarded as a generalization of the class of FSMX sources. The next–state function of an order–m finite–memory source depends on at most the m most recent symbols produced by that source. This relationship may be expressed as

$$\mathbf{s}_i^{(t)} = \text{suf}(x_{i-m+1} x_{i-m+2} \ldots x_i). \tag{3.4}$$

The states $\mathbf{s}_i^{(t)}$ are the finite–length symbol contexts of the source, and each context has a certain symbol distribution associated with it.

Finite–memory sources are very similar to tree sources. A tree source is characterized by a tree data structure [81]. This data structure has a root node, internal nodes and leaf nodes. Parent nodes are connected to child nodes with edges, and each edge is associated with a single symbol of the source alphabet. Any edge symbol is unique among the symbols of the edge's siblings. Each node of a q–ary tree source has at most q child nodes. All the internal nodes of a full q–ary tree source, including the root node, have exactly q child nodes.

Assume that each tree node is associated with a symbol sequence that equals the concatenation of the edge symbols that are encountered upon traversing the tree from

[1] This thesis uses the definitions of these source models that appear in references [10, 81].

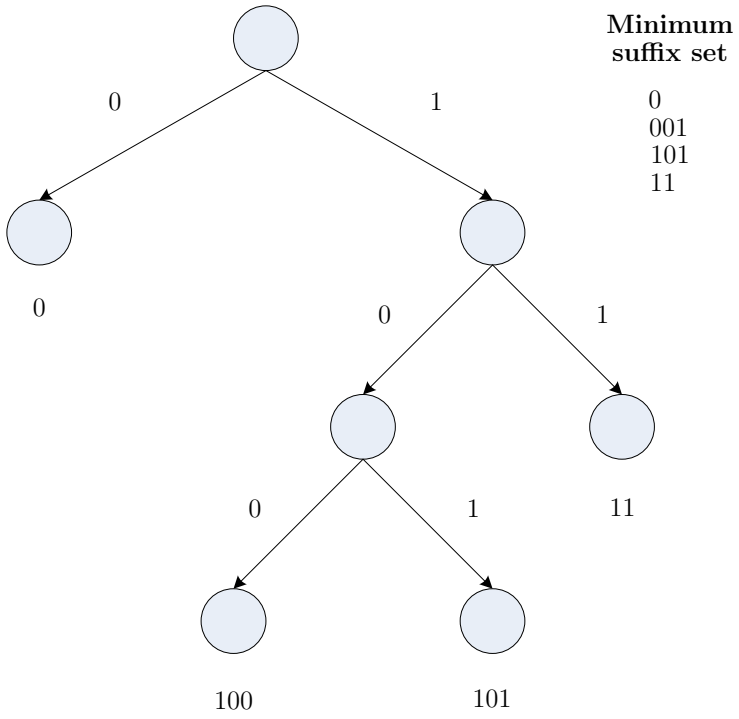


Figure 3.2: An example of a full binary tree source and its minimum suffix set [81].

its root to the specific node [81]. The sequences that are associated with the leaf nodes of a full q–ary tree may be used to construct a minimum suffix set. The suffixes of this set are equal to the sequences that are associated with the leaf nodes, but in reverse order. An example of a full binary tree source, with a minimum suffix set that was derived from it, are presented in figure 3.2 on page 41.

The sequences that are associated with the leaves of the full q–ary tree source are referred to as the states of the source [81]. Each state of a full q–ary tree source of depth m has a maximum length of m symbols. Each state of the tree source is assigned a certain probability mass function $p(x|\mathbf{s})$, where $x \in \mathcal{A}$ and $\mathbf{s} \in \mathcal{S}$. A source symbol is generated according to the probability mass function of the state in which the source currently resides.

A unique state–transition pattern is associated with each sequence produced by the source [81]. The next state of the source depends on at most the previous m symbols produced by the source. This relationship may be expressed as

$$\mathbf{s}_i^{(t)} = \mathcal{R}(\text{suf}(x_{i-m+1}, x_{i-m+2}, \ldots x_i)), \tag{3.5}$$

where $\mathcal{R}(\cdot)$ denotes sequence reversal. This equation is equivalent to the expression for finite–memory sources (refer to equation 3.4) [10, 81]. It follows that the minimum suffix set of the tree source is equal to the source's set of preceding contexts.

The probability of a tree source producing a certain sequence $\mathbf{x}^n$ equals

$$\Pr(\mathbf{x}^n|\mathbf{s}_0^{(t)}) = \prod_{i=1}^{n} p(x_i|\mathbf{s}_{i-1}^{(t)}). \tag{3.6}$$

This equation is identical to the equation for finite–memory sources [10, 81].

The primary difference between a tree source and an FSM source is the fact that a state transition of an FSM source depends only on the most recent state and the most recent symbol that the source produced [10,81]. A state transition of a tree source may depend on up to m of the most recent symbols that the source produced, where m is the depth of the tree. As an example, consider the tree source of figure 3.2, which has a depth equal to three. The states of this tree source are `0`, `100`, `101` and `11`. The preceding contexts that are associated with the states are `0`, `001`, `101` and `11`. Assume that the tree source is in context `0` at some point in time, and that it produces the symbol `1` while in that context. The length–three suffix of the symbol sequence equals x_{i-2}`01`, where x_{i-2} is the last symbol that was produced prior to entering state `0`. The transition to the next state of the tree source clearly depends on the symbol x_{i-2} and not only on the most recent context (`0`) and symbol (`1`), as the next context may only equal `001` or `101`.

If a tree source cannot be represented as an FSM source, it may be extended in order to represent it as an equivalent FSM source [81]. The extension process consists of adding additional nodes to the tree source. Several definitions that are required in order to summarize the extension process are provided in what follows.

Assume that the set $\mathcal{T}$ defines a tree source with tree nodes $t \in \mathcal{T}$ [81]. A tree defined by $\mathcal{T}'$ is said to be a refinement of the tree defined by $\mathcal{T}$ if $\mathcal{T} \subseteq \mathcal{T}'$. Let the set of leaf nodes of the tree defined by $\mathcal{T}$ be expressed as $\mathcal{L}_{\mathcal{T}}$. Furthermore, for each $l \in \mathcal{L}_{\mathcal{T}}$, let $\text{suf}_{\text{c}}(l)$ denote a set that contains all the suffixes of the sequence that is associated with the leaf node l, and let $\mathcal{E} = \bigcup_{l \in \mathcal{L}_{\mathcal{T}}} \text{suf}_{\text{c}}(l)$.

A refined tree $\mathcal{T}'$ that is defined as $\mathcal{T}' = \mathcal{T} \cup \mathcal{E}$ is referred to as the FSM closure of the tree source defined by $\mathcal{T}$ [81]. The FSM closure of a tree source $\mathcal{T}$ is the smallest possible extension of the tree $\mathcal{T}$ that may be successfully represented as an equivalent FSM source. All state transitions of the extended tree source depend on only the most recent state of the source, as well as the most recent symbol that the source produced.

To illustrate the tree extension process, consider the tree source of figure 3.2 [81]. The union of all the suffixes that are associated with all the leaf node sequences is given by $\mathcal{E} = \{$`0`, `1`, `11`, `100`, `101`, `00`, `01`$\}$. The only sequences in this set that are not associated with nodes in the tree source of figure 3.2 are `00` and `01`. The FSM closure of the tree source is obtained by adding the leaves that are associated with these sequences to the tree. The FSM closure of the tree source of figure 3.2 is presented in figure 3.3. The extended tree may be represented as an equivalent FSM source using a state–transition diagram, as demonstrated in figure 3.4 on page 43.

The forward transform of finite–memory source sequences Researchers investigated the distribution of the BWT output sequence in order to motivate the performance of BWT–based source codes. Effros [82] and Viswesvariah et. al. [83] investigated the distribution of the BWT output sequence when the transform is applied to sequences from various sources. In a comprehensive paper regarding BWT–based source codes, Effros et. al. [10] statistically characterized the output of the forward BWT when applied to source sequences produced by finite–memory sources. This section summarizes these characteristics.

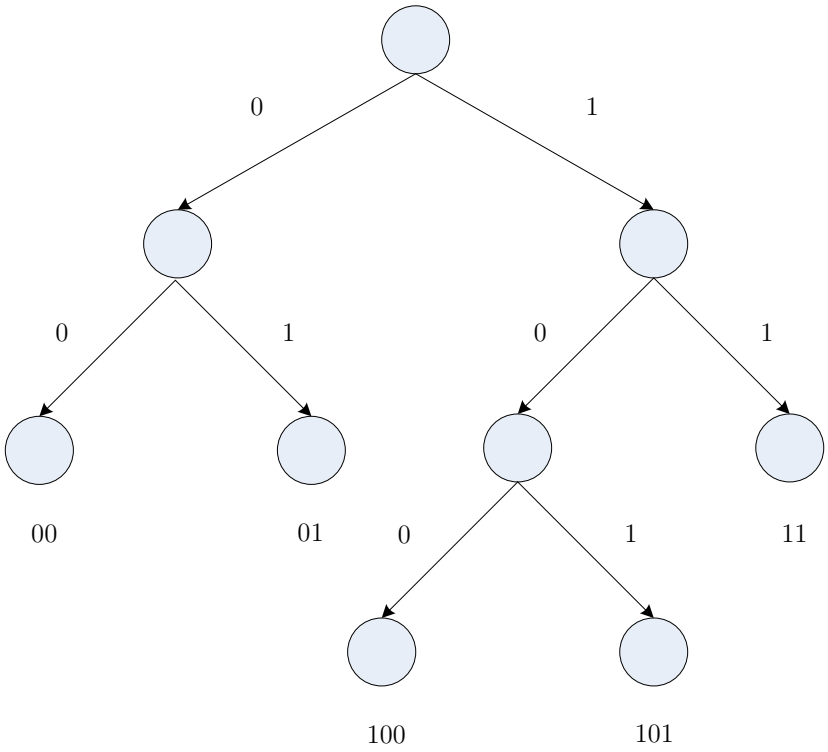


Figure 3.3: FSM closure of the binary tree source of figure 3.2 [81].

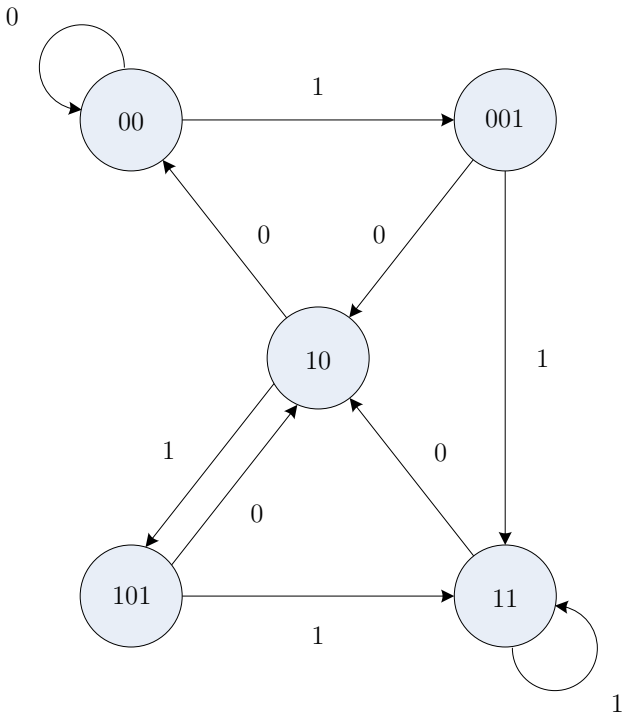


Figure 3.4: The state–transition diagram of the FSM closure of figure 3.3.

The theoretical investigation of the BWT, as performed by Effros et. al. [10], assumes that the source sequences are reversed and subsequently appended with the EOF symbol before being transformed. The reversal of a source sequence causes the reversed preceding contexts of the sequence to be sorted relative to one another during the forward transform[2]. The forward transform is expressed as

$$(\mathbf{Z}^{n+1}, \mathcal{I}) = \text{BWT}(\mathcal{R}(\mathbf{X}^n)\$), \tag{3.7}$$

where \$ represents the EOF symbol, $\mathbf{X}^n$ is the source sequence, $\mathcal{R}(\cdot)$ denotes sequence reversal, $\mathbf{Z}^{n+1}$ is the transformed source sequence, and $\mathcal{I}$ is the index of the original source sequence in the sorted transform table (i.e. the BWT index).

As row $\mathcal{I}$ of the sorted table contains the original sequence, the EOF symbol is the final symbol in this row. The $\mathcal{I}$th symbol of the transformed sequence therefore equals the EOF symbol. The n–symbol sequence $\mathbf{W}^n$, which is defined as

$$\begin{aligned} \mathbf{W}^n &= \{Z_1, Z_2, \ldots Z_{\mathcal{I}-1}, Z_{\mathcal{I}+1}, \ldots Z_{n+1}\} \\ &= \{\mathbf{Z}_1^{\mathcal{I}-1}, \mathbf{Z}_{\mathcal{I}+1}^{n+1}\}, \end{aligned} \tag{3.8}$$

does not contain the EOF symbol. The source sequence $\mathbf{X}^n$ may be obtained from the sequence $\mathbf{W}^n$ by first inserting the EOF symbol at index $\mathcal{I}$ of this sequence, and subsequently performing the reverse transform on this sequence. The probability distribution of the sequence $\mathbf{W}^n$ is derived in this section.

In order to derive a probability distribution for the BWT output sequence, it is necessary to investigate how the addition of the EOF symbol to the reversed source sequence affects the forward transform [10]. The change in the forward transform is explained in what follows. Observe that there are m rows in the sorted transform table with the EOF symbol present among the first m symbols of the row, and $n-m+1$ rows that do not share this property (m is the length of the longest context of the source). As the minimum suffix set associated with the finite–memory source is complete, a prefix of each of the $n-m+1$ rows with no EOF symbol among the initial m symbols equals a reversed version of one of the suffixes in the minimum suffix set of the source. Next consider one of the m rows with the EOF symbol present among its initial m symbols. It is possible that no prefix of this row equals a reversed version of any suffix in the minimum suffix set of the source. If this is the case, the last symbol of this row will be associated with a context that is not related to one of the contexts of the original source during the forward transform of the sequence. Let the set $\mathcal{Q}^c$ be defined as

$$\begin{aligned} \mathcal{Q}^c = \{(X_{i-1}, X_{i-2}, \ldots X_2, X_1, \$) \quad &: \\ 1 \leq i \leq m \quad \wedge \quad &[(X_{i-1}, X_{i-2}, \ldots X_1) \neq \mathcal{R}(\mathbf{s}) \ \forall \ \mathbf{s} \in \mathcal{S}]\}, \end{aligned} \tag{3.9}$$

and let $\mathcal{Q}' = \mathcal{Q} \cup \mathcal{Q}^c$, where $\mathcal{Q} = \{\mathcal{R}(\mathbf{s}) : \mathbf{s} \in \mathcal{S}\}$. It follows that

$$|\mathcal{Q}'| \leq |\mathcal{S}| + m. \tag{3.10}$$

[2]The conclusions that follow also apply to source sequences that are not reversed prior to the forward transform [10]. This observation is a consequence of the fact that any reversed finite–memory source sequence is equivalent to a sequence from a (possibly different) finite–memory source.

Effros et. al. [10] proved that the distribution of the BWT output sequence, when applied to a finite–memory source sequence, equals

$$\Pr(\mathbf{W}^n, \mathcal{I}) = \prod_{j=1}^{C} \prod_{i=T_j}^{T_{j+1}-1} p(W_i|\mathcal{R}(\mathbf{q}'_j)), \tag{3.11}$$

where $C = |\mathcal{Q}'| \leq |\mathcal{S}| + m$, $\mathbf{q}'_j \in \mathcal{Q}'$, and the sequences $\mathbf{q}'_j$ are sorted in ascending order w.r.t. the integer j. The distribution $p(W_i|\mathcal{R}(\mathbf{q}'_j))$ is the conditional distribution associated with state $\mathcal{R}(\mathbf{q}'_j)$ of the finite–memory source. The integer T_j be defined as

$$T_j = 1 + \sum_{i=1}^{n} \sum_{k=1}^{j-1} 1(\text{pre}(\mathbf{X}_i^1\$) = \mathbf{q}'_k), \tag{3.12}$$

where pre$(\cdot)$ denotes the prefix of a sequence, and $1(\cdot)$ equals unity if its argument is true (and zero if not). The integer T_j equals the index of the first symbol in the BWT output sequence that precedes context $\mathbf{q}'_j$ in the reversed source sequence.

Equation 3.11 resembles the distribution of a piecewise independent and identically distributed (p.i.i.d.) symbol sequence [10]. Each piecewise segment consists of only those symbols that occur within a certain source context, or state. A remarkable aspect of the BWT is that it is able to produce output sequences with the distribution of equation 3.11, but without the algorithm requiring any apriori knowledge of the finite–memory source or its parameters.

Effros et. al. [10] observed that the distribution of equation 3.11 differs from that of a true p.i.i.d. symbol sequence in certain aspects — the interested reader is referred to reference [10] for details regarding the differences. Viswesvariah et. al. [83] investigated the normalized Kullback–Leibler divergence between the BWT output distribution and a p.i.i.d. distribution, and found that it converged to zero as the source sequence length tended to infinity. It was proved that

$$\frac{1}{n} D(\Pr(\mathbf{Y}^n)||R) \leq \frac{c}{\sqrt{n}} \tag{3.13}$$

for some constant c, where R is a p.i.i.d. symbol distribution, $D(\cdot)$ is the Kullback–Leibler divergence, and $\mathbf{Y}^n$ is the output sequence of the BWT when applied to a reversed source sequence.

3.1.2 The reverse transform

This section is a summary of the reverse BWT and its implementation.

3.1.2.1 Description

The original source sequence may be recovered from the BWT output sequence and the BWT index by using two observations regarding the transformed sequence [16]:

1. The transformed sequence is a permutation of the original sequence.

Table 3.3: The first (a) and second (b) tables of the reconstruction of the original sequence `bananas` (adapted from reference [10]).

						b
						n
						n
						s
						a
						a
						a

→

a						b
a						n
a						n
b						s
n						a
n						a
s						a

Table 3.4: The third (a) and fourth (b) tables of the reconstruction of the original sequence `bananas` (adapted from reference [10]).

b	a					
n	a					
n	a					
s	b					
a	n					
a	n					
a	s					

→

a	n					b
a	n					n
a	s					n
b	a					s
n	a					a
n	a					a
s	b					a

2. The first symbol of each following context may be obtained by sorting the transformed sequence.

The reverse transform is carried out by iteratively reconstructing the table that contains the sorted contexts (refer to table 3.2 on page 34). Once the table is reconstructed, the BWT index is used to obtain the original sequence from the table.

The reverse transform is illustrated by the same example that was used to illustrate the forward transform [10]. The first step of the reverse transform is to place the transformed sequence `bnnsaaa` in the final column of an empty table (refer to table 3.3(a) on page 46). As the first column of the original table contains the sequence in lexicographical order, and the final column is a permutation of the original sequence, the first column equals the final column that is sorted lexicographically.

As the rows of the original table of sorted contexts are the cyclic shifts of the original sequence, the symbols in the final column of the table precede those in the first column [10]. The second table (table 3.3(b)) therefore contains all pairs of symbols that appear next to each other in the original sequence. If these symbol pairs are sorted, the first two columns of the original table are obtained. The columns of the second table are therefore cyclically shifted to the right (refer to table 3.4(a)), the rows of this table are sorted, and the transformed sequence is placed in its final column. The result is the fourth table of the reverse transform (refer to table 3.4(b)).

From this point onwards, the remainder of the original table may be reconstructed by iteratively [10]

1. shifting all columns of the table one column to the right (in a cyclic fashion),

Table 3.5: The fifth (a) to twelfth (h) tables of the reconstruction of the original sequence `bananas` (adapted from reference [10]). The recovered sequence is equal to the fourth row of the final table (`bananas`), and is printed in bold.

b	a	n					a	n	a				b
n	a	n					a	n	a				n
n	a	s					a	s	b				n
s	b	a					→	b	a	n				s
a	n	a					n	a	n				a
a	n	a					n	a	s				a
a	s	b					s	b	a				a
							↙						
b	a	n	a				a	n	a	n			b
n	a	n	a				a	n	a	s			n
n	a	s	b				a	s	b	a			n
s	b	a	n				→	b	a	n	a			s
a	n	a	n				n	a	n	a			a
a	n	a	s				n	a	s	b			a
a	s	b	a				s	b	a	n			a
							↙						
b	a	n	a	n			a	n	a	n	a		b
n	a	n	a	s			a	n	a	s	b		n
n	a	s	b	a			a	s	b	a	n		n
s	b	a	n	a			→	b	a	n	a	n		s
a	n	a	n	a			n	a	n	a	s		a
a	n	a	s	b			n	a	s	b	a		a
a	s	b	a	n			s	b	a	n	a		a
							↙						
b	a	n	a	n	a		a	n	a	n	a	s	b
n	a	n	a	s	b		a	n	a	s	b	a	n
n	a	s	b	a	n		a	s	b	a	n	a	n
s	b	a	n	a	n		→	**b**	**a**	**n**	**a**	**n**	**a**	**s**
a	n	a	n	a	s		n	a	n	a	s	b	a
a	n	a	s	b	a		n	a	s	b	a	n	a
a	s	b	a	n	a		s	b	a	n	a	n	a

2. sorting the rows of the table, and

3. inserting the transformed sequence into the final column of the table.

The reconstruction of the original table that contains the sorted contexts of the sequence `bananas` is completed in tables 3.5(a) to 3.5(h) on page 47 [10]. Upon complete reconstruction of the original table, the BWT index is used to obtain the original sequence from the table — in the case of the example, the index equals four. The recovered sequence is therefore the fourth row of the reconstructed table, or `bananas`.

The reverse transform may be expressed as

$$\mathrm{BWT}^{-1}(\texttt{bnnsaaa}, 4) = \texttt{bananas}. \tag{3.14}$$

3.1.2.2 Implementation

The reverse transform has a straightforward and computationally simple implementation [16]. The computational complexity of the implementation increases linearly w.r.t. the length of the source sequence. Instead of iteratively reconstructing the columns of the sorted table as demonstrated in section 3.1.2.1 and tables 3.3(a) to 3.5(h), the reverse transform may be carried out by iteratively traversing the symbols of the transformed sequence.

The key to developing an efficient implementation of the reverse transform is an observation regarding the first and final columns of the sorted table (refer to table 3.2 on page 34) [16]. The ith occurrence of any alphabet symbol `a` in the final column of the table has the same index in the source sequence as the ith occurrence of the same alphabet symbol `a` in the first column of the table. To prove this statement, observe that all the `a` symbols in the first column are lexicographically sorted relative to one another according to the symbols that follow them. The proof of the statement follows from the fact the symbols of the last column precede the symbols in the first column, and that all the `a` symbols in the final column are sorted relative to one another according to the symbols that follow them.

The iterative traversal of the transformed symbols is carried out as follows [16]. The transformed sequence is sorted to obtain a sequence equal to the first column of the sorted table. The BWT index that was produced during the forward transform is subsequently used. The symbol that occurs at this index of the sorted sequence is the first symbol of the original sequence. The symbol that matches this first symbol is located in the transformed sequence. Suppose that this symbol is present at index m of the transformed sequence. As the symbol at index m of the sorted sequence follows the first symbol of the original sequence, the second symbol of the original sequence is recovered. The symbol that matches this second symbol is subsequently located in the transformed sequence. These steps repeat until the entire sequence has been recovered.

3.2 The recency–rank code

This section contains a summary of the recency–rank code, which was proposed by Elias [84][3]. The recency–rank code forms part of many BWT–based source codes, and is typically used to encode the BWT output sequence [16]. The recency–rank code is also known as the move–to–front code or the book–stack code in the literature.

The recency–rank code is not a conventional source code as it does not assign shorter codewords to certain sequences, and longer codewords to others [84]. It replaces each symbol of the input sequence with an integer, and is reversible. The recency–rank code

[3]Elias [84] actually proposed two related codes, namely the recency–rank code and the interval code. The interval code is of lesser importance than the recency–rank code as it has certain negative properties. It is only mentioned briefly in this thesis.

is therefore equivalent to a transform — the terms 'recency–rank code' and 'recency–rank transform' are used interchangeably in the remainder of this thesis.

The purpose of the recency–rank code is to produce a sequence of integers with a distribution that is more stationary than the symbol distribution of its input sequence [85]. It converts the local stationarity of certain input sequences to a global stationarity [86]. The integer sequences may be encoded with reasonable effectiveness using a nonadaptive source code, as the source code need not constantly adapt to a changing symbol distribution.

The operation and implementation of the recency–rank encoder and decoder are summarized in this section. The distribution of the recency–rank encoder output sequence, in the case of the code being applied to both i.i.d. and p.i.i.d. sequences, is also investigated.

3.2.1 The forward transform

The forward recency–rank transform is summarized in this section. The summary includes a description of the transform and its implementation. The distribution of the recency–rank encoder output sequence is also considered.

3.2.1.1 Description

The encoders of the interval and recency–rank codes, when applied to the input sequence $\mathbf{x}^n = \{x_1, x_2, \ldots x_n\}$, produce the n–element integer sequences $\{f_{int}(x_1), f_{int}(x_2), \ldots f_{int}(x_n)\}$ and $\{f_{rr}(x_1), f_{rr}(x_2), \ldots f_{rr}(x_n)\}$ respectively [10]. The integer $f_{int}(x_i)$ of the interval coded sequence equals the number of symbols that have occurred since the previous occurrence of symbol x_i in the input sequence. The integer $f_{rr}(x_i)$ of the recency–rank coded sequence equals the number of distinct symbols that have occurred since the previous occurrence of symbol x_i in the input sequence. The integers of the transformed sequences may be mathematically expressed as

$$f_{int}(x_i) = \min\{k \geq 1 : x_{i-k} = x_i\} \tag{3.15}$$

and

$$f_{rr}(x_i) = |\{x_k : i - f_{int}(x_i) < k \leq i\}|. \tag{3.16}$$

The forward transforms are carried out in a symbol–by–symbol fashion, starting with the first symbol in the input sequence, and ending with the final symbol in the sequence.

Three relevant observations regarding interval and recency–rank codes are provided below.

1. Interval coding produces integers that may be as large as the length of the input sequence [84]. In contrast to interval coding, recency–rank coding produces a maximum of $|\mathcal{A}|$ distinct integers, where $|\mathcal{A}|$ equals the number of symbols in the source alphabet.

2. It was proved that $f_{rr}(x_i) \leq f_{int}(x_i)$ for all i [10]. Any source code for integer sequences that produces codewords with lengths proportional to the magnitude of the integers will therefore have a shorter or equal average codeword length when

applied to recency–rank coded sequences, instead of interval coded sequences (assuming both codes are applied to the same sequences).

3. The definitions of both the interval and recency–rank codes assume that all symbols in the alphabet have been encountered previously at all indices of the input sequence. This assumption does not hold true at the beginning of the input sequence. The issue may be resolved by assuming that all alphabet symbols occur, in order from the lexicographically largest to the lexicographically smallest, immediately prior to the first symbol of the input sequence.

The remainder of this thesis will follow the example of Effros et. al. [10] and disregard interval codes, due to their negative properties.

3.2.1.2 Implementation

The forward recency–rank transform has a straightforward yet efficient implementation [16]. Assume that the source alphabet contains k symbols. The implementation first initializes a k–element symbol array with element i equal to the $(i+1)$th lexicographically smallest symbol of the alphabet, where $0 \leq i \leq k-1$. This array is referred to as the rank array for the remainder of this thesis. The elements of the rank array are updated after transforming each input symbol so that the symbol at index i of the array has a recency rank of i, where $0 \leq i \leq k-1$. The symbol at the front of the rank array is the most recent alphabet symbol in the input sequence, and the symbol at the back of the rank array the least recent alphabet symbol in the input sequence.

The implementation processes the symbol sequence iteratively in a symbol–by–symbol fashion [16]. At the start of the ith iteration, it locates symbol x_i in the rank array. Suppose that symbol x_i is located at index j of the rank array. The integer j, which equals the rank of symbol x_i, is the ith output integer produced by the implementation. The rank array is subsequently updated by moving symbol x_i to the front of the rank array (i.e. it is assigned a rank of zero). All elements of the rank array with index smaller than j are shifted one step towards the back of the array. Symbol x_i therefore becomes the symbol with the lowest rank, and all symbols that were initially of lower rank have their ranks incremented by one. The algorithm proceeds by transforming the remaining symbols of the input sequence, and updating the rank array after transforming each symbol.

Instead of searching for the location of symbol x_i in the rank array during iteration i, the algorithm may 'look up' the location of the symbol using another array [16]. To illustrate the symbol lookup process, suppose that the algorithm maintains a separate k–element lookup array. Element $i-1$ of this lookup array contains a pointer to the location, in the rank array, of the ith lexicographically smallest alphabet symbol. This approach allows for swift access to the symbols in the rank array. It has the drawback of having to update the lookup array at the end of each iteration, which increases the computational complexity of the forward transform. It was demonstrated that an implementation which uses the lookup array approach requires more processing time to transform typical sequences than the original implementation, provided the symbol alphabet is not excessively large.

3.2.1.3 Output distribution

The distribution of the recency–rank encoder output sequence depends largely on the distribution of the input sequence to which it is applied. The transform of i.i.d. input symbols, as well as p.i.i.d. input symbols is considered in this thesis.

The forward transform of i.i.d. input symbols Suppose that a certain source produces a sequence of i.i.d. symbols. Upon applying the recency–rank transform to the sequence, a dependency is typically introduced between each symbol of the sequence and one or more of the symbols that precede it in the sequence [10]. The rank that is assigned to each symbol is therefore a function of the symbols that precede it in the sequence. This property also applies to the probability distribution that is associated with the rank of each symbol.

The fact that a symbol depends on its predecessors in the input sequence implies that the rank probability distribution changes from symbol to symbol [10]. In order to effectively encode an i.i.d. sequence that was transformed using the recency–rank code, a source code that is able to resolve the dependency between a source symbol's rank and the ranks of its predecessors is required. If a rudimentary source code for i.i.d. sequences is used to encode the recency–rank encoder output, the code may be less effective compared to the case where the source code is applied to the untransformed sequence.

The forward transform of p.i.i.d. input sequences Suppose that the recency–rank transform is applied to a sequence of p.i.i.d. symbols, and that the symbol distribution of each piecewise segment is highly biased. The ranks assigned to the symbols of each piecewise segment of this sequence will typically have a nonincreasing probability distribution over increasing rank, due to the bias of each segment symbol distribution [16]. As each segment symbol distribution becomes more biased, the rank distribution associated with the symbols within each segment typically becomes more biased towards lower ranks.

The transitions between the piecewise segments of a p.i.i.d. symbol sequence are of interest to the characterization of the recency–rank encoder output sequence [16]. If two neighbouring segments have similar symbol probability distributions, the rank array will not be significantly reordered as the implementation begins to transform the second segment. This observation suggests that the rank distribution does not change significantly between the two segments. If the two neighbouring segments have dissimilar symbol distributions, significant reordering of the rank array typically occurs as the implementation begins to transform the second segment. This reordering of the rank array produces a short burst of high–valued ranks at the initial symbols of the second segment. As the transform of the second segment proceeds, less significant changes are made to the rank array, and the rank distribution again assumes a nonincreasing profile.

If the recency–rank transform is applied to a sequence of piecewise i.i.d. symbols, the distribution of its output ranks is typically nonincreasing [16]. The sequence of ranks typically contains short bursts of high–valued ranks at the transition points between some of the piecewise segments. Some authors neglect to mention the bursts of

high–valued ranks in their characterization of the output sequence, and state that the recency–rank transform shifts the local stationarity of the input sequence to a global stationarity [86], or that the rank distribution is more stationary than the distribution of the input symbols [85]. While these statements are true, it is important to remain aware of the bursts of high–valued ranks between segments, as these bursts are problematic for some statistical encoders.

3.2.2 The reverse transform

The reverse recency–rank transform is summarized in this section. The summary includes a description of the reverse transform and its implementation.

3.2.2.1 Description

The original sequence is recovered symbol–by–symbol from the sequence of ranks, starting with the first rank and ending with the final rank in the sequence [16]. To reverse transform the rank j at index i of the transformed sequence, the algorithm finds the $(j+1)$th distinct symbol in the recovered sequence, moving backward from symbol $i-1$ to the first symbol of the recovered sequence. The $(j+1)$th distinct symbol equals the recovered symbol at index i of the recovered sequence, or x_i.

The reverse transform assumes that the alphabet symbols appear in order from the lexicographically largest to smallest immediately prior to the first symbol in the original input sequence [16]. This assumption guarantees the successful recovery of the input sequence, provided the forward transform was carried out under the same assumption.

3.2.2.2 Implementation

In order to recover the symbol x_i from the rank j at index i of the transformed sequence, an implementation of the reverse transform may search for the $(j+1)$th unique symbol in a backward direction starting at index $i-1$ of the recovered sequence. An alternative to searching for each symbol is to use the rank array [16]. The rank array is initialized at the start of the reverse transform, and in the same manner as in the forward transform.

The implementation of the reverse transform operates as follows [16]. Each rank is reverse transformed, starting at the first rank of the transformed sequence and proceeding to the last rank of the sequence. Upon encountering the rank j at index i of the transformed sequence, the symbol at index j of the rank array is accessed. The symbol present at this index of the array is the recovered symbol at index i of the recovered sequence. The rank array is updated in an identical fashion to the rank array of the forward transform after recovering each symbol.

3.3 BWT–based source codes

This section concerns source codes that make use of the Burrows–Wheeler transform. A block diagram of an elementary BWT–based source code is used to illustrate the design of these codes, and to present the basic properties of these codes. Many alterations and additions to the elementary BWT–based source code were proposed in the literature.

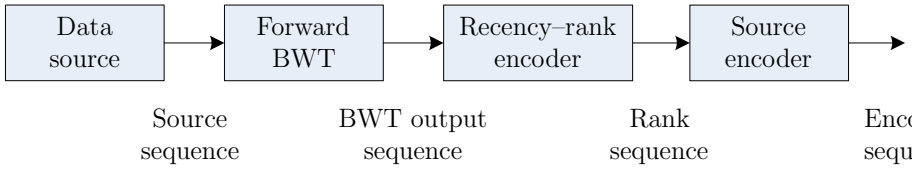


Figure 3.5: A block diagram of an elementary BWT–based source code [16].

The purpose of these modifications is to improve the performance of the elementary code — the more significant modifications are summarized at the end of this chapter.

3.3.1 The elementary BWT–based source code

The elementary BWT–based source code that is discussed in this section is similar to a source code that was examined by Fenwick [16]. A block diagram of the elementary source code is presented in figure 3.5 on page 53. Each block of the elementary source code is discussed in what follows.

3.3.1.1 The data source

This thesis is ultimately concerned with the source coding of sequences from abstract data sources with memory. Tree sources are considered, as this type of source produces symbol sequences with statistical properties that are similar to those of real–life data such as English text [10]. Sequences from those sources that produce independent symbols do not accurately represent this type of data. The discussion of the elementary BWT–based source code assumes that an appropriate tree source produces the source sequence that is to be encoded.

Effros et. al. [10] stated that the symbol contexts present in practical data such as English text are typically quite long. To illustrate this property, consider reading the first half of a very long sentence and having to predict the next word of the sentence. The next word of the sentence is best predicted by examining as many of the words of the first half of the sentence as possible. If one only reads the final few words of the first half of the sentence, one typically cannot predict the next word with great accuracy.

Effros et. al. [10] observed that few distinct symbols typically appear in a symbol context of English text. This property implies that an accurate model of the practical data would define a biased distribution for the symbols that occur in each context, or keep many of the alphabet symbols from occurring in each context.

3.3.1.2 The forward BWT

The first step that the elementary source code carries out is the application of the forward Burrows–Wheeler transform to the source sequence [16]. The symbols of the BWT output sequence are asymptotically (w.r.t. the length of the sequence) p.i.i.d., as the source symbols that are produced in each context of the source are i.i.d. [10]. The forward transform places those source symbols that share a common context in the same piecewise segment of the transformed sequence.

The usefulness of the BWT lies in two properties of its output sequences [10]. The first property is the statistical independence between the symbols in each piecewise segment of the output sequence. The second property is the stationary nature of the symbol distribution within each piecewise segment of the output sequence. The effective source coding of the BWT output sequence is (typically) computationally less complex and more straightforward than the effective source coding of the original source sequence. The BWT provides elementary source codes the ability to remove a large fraction of the redundancy from the source sequences, without having to construct and maintain a complex model of the source.

The BWT transforms source sequences without any prior knowledge of the source statistics [10]. In practice the source encoder has no prior knowledge of the symbol distribution that is associated with the sequence it is to encode. Without knowledge of the underlying source model, the locations of the transition points between the piecewise segments of the BWT output sequence are unknown. The source coding of the BWT output, when applied to real–life data, is complicated by the fact that the transition points between its piecewise segments are unknown. Many BWT–based source codes maintain a simple model of the BWT output that depends exclusively on the local statistics of the BWT output sequence [85].

If the BWT is applied to sequences from accurate source models of English text, its output typically contains runs of identical symbols within one or more of its piecewise segments [16]. This is a consequence of the fact that only a few distinct alphabet symbols occur in many of the contexts of a source that accurately models English text [10]. This property may be exploited to improve the effectiveness of the BWT–based source code.

The forward transform is ideally carried out over the entire source sequence that is to be encoded. This approach is not necessarily practical, as the memory required by the forward BWT implementation is proportional to the source sequence length, and may become excessive [10]. Practical implementations of BWT–based source codes, such as the `bzip2` compression and archiving software utility, uniformly divide the source sequence into blocks of a certain length [87]. Each block is transformed independently from the other blocks, after which the entire sequence is source coded [10].

3.3.1.3 The recency–rank encoder

The second step that the elementary source code carries out is the recency–rank coding of the BWT output sequence [16]. The recency–rank encoder transforms the locally stationary output of the BWT into a sequence that is nearly stationary over its entire length [86]. It transforms runs of identical symbols in the BWT output sequence into runs of the rank zero [16]. Long segments that contain only a few distinct symbols are transformed into segments of equal length that contain lower–valued ranks. The symbol distribution of the recency–rank encoder output sequence is typically biased towards lower–valued ranks.

The symbol distributions of the piecewise segments in the BWT output sequence can often be distinguished from one another, whereas the rank distributions of the piecewise segments are often indistinguishable. Some source encoders for the recency–rank encoder output sequence encode all ranks according to a source model with a

single, zero–order context [16]. The rank distribution of the model is updated to reflect the local distribution of the sequence as its source coding proceeds. The recency–rank encoder output may, in some cases, be encoded effectively using a source code that does not adapt to any changes in the rank distribution of the sequence. An example of this type of source code is an integer code, which was considered by Effros et. al. [10] for encoding the recency–rank encoder output.

Fenwick [16] observed that the recency–rank encoder output consists of low–entropy intervals and short, high–entropy intervals located in between the low–entropy intervals. The high–entropy intervals correspond to the transition points between the piecewise segments of the BWT output sequence. These intervals contain distinct high–valued ranks that are produced as the rank array is rearranged to match a new symbol distribution during recency–rank coding. The rank array is typically rearranged when the recency–rank encoder moves from one segment in the BWT output sequence to another. The short bursts of high–valued ranks in the recency–rank encoder output are problematic to certain adaptive source codes. These issues are discussed in section 3.3.1.4.

While the recency–rank encoder greatly simplifies the coding of the BWT output sequence, it does have a drawback. It introduces a dependency between each symbol in the BWT output sequence and all the symbols that precede it in the sequence [10]. An optimally effective source code would need to resolve the dependency between the symbols in order to source code the sequence. This requirement implies a less straightforward and computationally more complex source code. Effros et. al. [10] suggested that the recency–rank encoder only be used if it can produce a nonasymptotic gain in coding effectiveness.

The previous observation regarding the optimal source coding of the recency–rank encoder output motivated the development of BWT–based source codes that omit the recency–rank encoder. Effective BWT–based source codes that do not use the recency–rank encoder were demonstrated [16,88]. The recency–rank encoder nevertheless forms part of many BWT–based source codes. Several improvements and alternatives to the basic recency–rank encoder are presented in section 3.3.2.3 on page 65.

3.3.1.4 The source encoder

Early BWT–based source codes used arithmetic codes with a variety of source models to encode the recency–rank encoder output sequence (refer to figure 3.5). Fenwick [16] initially used the CACM arithmetic encoder of Witten et. al. [89], and subsequently used an improved version of the same encoder (which was developed by Moffat et. al. [90]). The improved version of the CACM encoder was intended for use in a PPM implementation, and proved to be significantly less effective when used to encode the recency–rank encoder output sequence than the original CACM encoder [16].

The poor performance of the improved CACM arithmetic encoder of Moffat et. al. [90] was eventually attributed to its assumption of a multiple–context source model [16]. The output of the recency–rank encoder is not accurately modeled as having a large number of rank contexts. Fenwick [16, 91] reported that a source model with a single zero–order context is more effective when used during the source coding of the recency–rank encoder output.

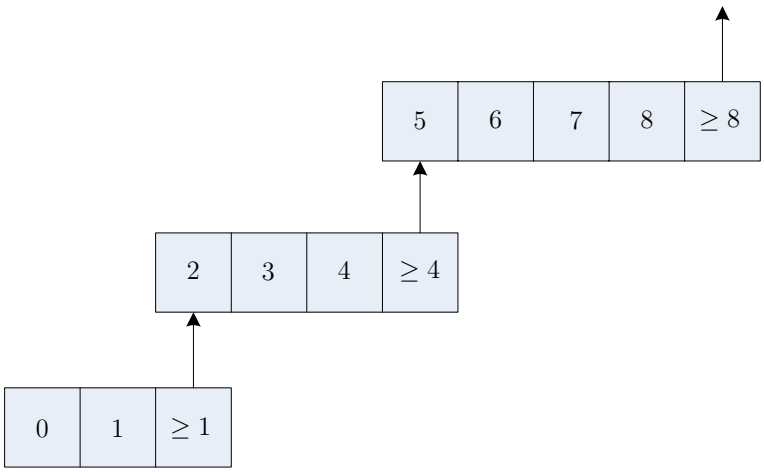


Figure 3.6: The cascaded source model [16, 88].

The approach of using a source model with a single zero–order context to encode the recency–rank encoder output is reasonable, as the rank distributions of many piecewise segments in the BWT output sequence are often indistinguishable. However, the presence of high–entropy intervals between the low–entropy intervals of the recency–rank encoder output is problematic to adaptive source codes [16]. Two problems need to be addressed by an appropriate source code for encoding the recency–rank encoder output.

1. The source encoder must have the ability to accommodate ranks with both low and high probabilities of occurrence [16]. Lower–valued ranks appear much more frequently in the recency–rank encoder output sequence than higher–valued ranks. This requirement forces the source model of the encoder to maintain a significant difference between its rescaling limit for the accumulated symbol counts and the per–symbol increment.

2. The source encoder must adapt rapidly as it moves from a low–entropy segment to a high–entropy segment in the recency–rank encoder output sequence [16]. This requirement forces the source model to maintain a small ratio of the rescaling limit for the accumulated symbol counts to the per–symbol increment, as the frequent rescaling of symbol counts is required in order to rapidly adapt to the new distribution.

These two problems are in conflict regarding the requirements they impose on the source model.

The conflicting requirements that are imposed on the source model with a single zero–order context may be resolved by using a different source model. Fenwick [16, 88] proposed a source model that consists of multiple contexts, and referred to this model as the cascaded model. The contexts of this model are arranged in levels, and each context contains a number of ranks (refer to figure 3.6). The lower–level contexts contain those ranks with a relatively high probability of occurrence, while the higher–level contexts contain ranks with a lower probability of occurrence. Each distinct rank is assigned

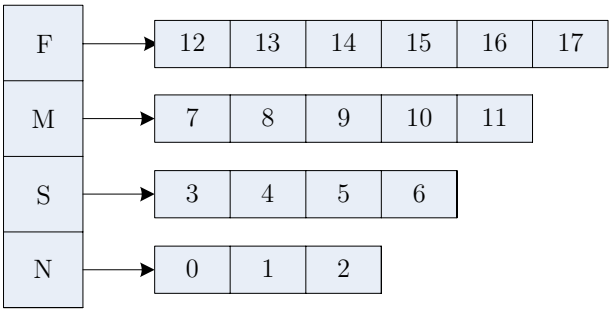


Figure 3.7: The structured source model [16, 92].

to a single level only. The contexts of adjacent levels are connected with an escape symbol mechanism, which is represented by the arrows of figure 3.6. By placing the likely and unlikely ranks in different contexts, the problem of maintaining a wide range of accurate probabilities of occurrence for the ranks is resolved.

A source encoder uses the cascaded model as follows [16]. The encoder always considers the context at the bottom level of the model when it starts to encode a new rank, regardless of any previous ranks that it encoded. To encode a rank, the encoder first determines the level of the context in which the rank resides. It then produces a sequence of escape symbols to signify that it is moving to the appropriate context. If the rank is located in the bottom–level context, no escape symbols are produced. The rank is subsequently encoded according to the distribution that is associated with the context.

The structured model is another multiple–context source model that may be used to effectively encode the recency–rank encoder output [16, 92]. The model may be likened to a tree trunk with branches (refer to figure 3.7). The trunk is divided into levels, and each level is associated with only a single branch. Each branch corresponds to a single context, and each distinct rank is assigned only to a single branch.

An encoder uses the structured model as follows [16]. The encoder first determines the trunk level of the branch that contains the rank that is to be encoded. It next produces the level number as output. The rank is subsequently encoded according to the distribution of the context that is associated with the branch.

The contexts of the structured source model may adapt at different rates to a change in the distribution of the recency–rank encoder output[4] [16]. The rates of adaptation are specified by assigning different values to the rescaling limits and the per–symbol increments of the branch contexts. With an appropriate assignment of ranks to branches, and the proper selection of context parameters, the requirements of rapid adaptation and the accommodation of a wide range of rank probabilities may be met.

Fenwick [16] reported a significant gain in source coding performance when using the cascaded and structured source models. The design of appropriate source models for encoding the recency–rank encoder output is discussed in greater detail in section 3.3.2.4 on page 77.

Fenwick [93] suggested that a run–length encoder be used to encode the runs of

[4]This observation also holds for the contexts of the cascaded model.

zero–valued ranks in the recency–rank encoder output. The run–length code provides an effective representation of the runs of zero–valued ranks. It is also beneficial to certain source encoders that follow it in the elementary BWT–based source code. By removing long runs of zero–valued ranks, it ensures that the source model of an adaptive code does not overestimate the probability of the zero–valued rank immediately after each run has been encoded (i.e. the source model may become overly biased towards the zero–valued rank due to the large number of zero–valued ranks that were encountered).

3.3.2 Improvements and additions to the elementary code

This section presents a summary of several improvements and additions to the elementary BWT–based source code of figure 3.5. The summary is divided into sections, and each section deals with one block of the elementary source code. A section regarding the preprocessing of the BWT input sequence is also included in the summary.

3.3.2.1 Preprocessing the BWT input sequence

The output at each stage of the BWT–based source code may be altered by processing the source sequence prior to its transformation by the BWT implementation [16]. The intelligent preprocessing of the BWT input sequence may improve the effectiveness and/or efficiency of the overall source code. Several preprocessing techniques are summarized in what follows.

Run–length coding Run–length coding the BWT input sequence, as stated in section 3.1.1.2, improves the efficiency of some BWT implementations [16]. The run–length code is detrimental to the effectiveness of the BWT, as it removes certain contexts from the source sequence. By run–length coding the input of the BWT implementation, its output sequence loses some of its structure, and becomes more difficult to encode effectively. Run–length coding therefore has a negative impact on the effectiveness of the overall source code.

Fenwick [88, 92] used a run–length code in which a sequence of six identical source symbols signals a run. The six symbols are followed by a length codeword, which represents the number of symbols that remain in the run. Fenwick found that run–length coding reduces the effectiveness of the overall source code by around 0.1%. This figure was derived from the source coding of real–life data using a BWT–based source code [16]. Balkenhol et al. [94] suggested that the use of a run–length code should be avoided unless it shortens the source sequence by more than 30%.

If the forward BWT implementation uses suffix trees or suffix arrays, the preprocessing of the BWT input with a run–length encoder is unnecessary, as it does not drastically improve the efficiency of these implementations [16].

Partial input alphabets Source sequences frequently contain only a small fraction of the distinct alphabet symbols that a source may produce [16]. English text, for example, typically contains the uppercase and lowercase letters of the Latin alphabet, the digits 0 to 9, and punctuation marks such as the period, comma, etc. English text documents rarely contain all of the control symbols that are defined in the original

128–symbol ASCII alphabet. This characteristic of source sequences may be exploited in order to improve the effectiveness of a source code.

If the source alphabet contains q distinct symbols, and only r of these symbols appear in a source sequence, a maximum of $\lceil \log_2(r) \rceil$ bits are required to uniquely represent each symbol in the sequence, where $1 < r < q$. To successfully use the partial alphabet, the source encoder has to inform the source decoder as to which of the alphabet symbols appear in the sequence and which are absent [16]. One method of informing the source decoder appends a bit vector to the source–coded data. This bit vector contains one bit for each distinct alphabet symbol — a bit equals one if the symbol is present in the sequence, and zero if it is absent.

The use of a partial input alphabet may also improve the performance of the recency–rank encoder, as well as some source encoders [16]. If a partial input alphabet is used, only those symbols that appear in the source sequence are present in the rank array of the recency–rank encoder. Any movement in the rank array would initially generate ranks with a smaller range of possible values than would be the case with the full source alphabet. This rank sequence may be encoded more effectively than the ordinary rank sequence.

Some adaptive source encoders also benefit from the use of a partial input alphabet [16]. The source encoder need not incorporate the unused symbols in its model of the source. The omission of the unused symbols from the source model may improve the effectiveness of the source code, depending on the source encoder and its implementation.

Fenwick [16] observed that the overhead of a partial alphabet often exceeds the gain in performance that it grants. The overhead of the partial input alphabet refers to the mechanism that the source encoder uses to inform the source decoder as to which symbols are present in the sequence. Fenwick proposed two bit sequences that the source encoder may produce in order to inform the source decoder, provided that ASCII text is being source coded. These sequences are

- a bit vector with one bit for each distinct alphabet symbol (as discussed previously), and
- a bit that indicates full mode or half mode (i.e. a ‘mode’ bit).

If more than 240 distinct symbols are present in the source sequence, the full symbol alphabet may be used — this mode of operation is referred to as the full mode. Only the first seven bits of each ASCII symbol is used in the half mode (i.e. the ASCII symbols with nonzero most significant bits are omitted from the alphabet). The use of a mode bit involves significantly less overhead than the use of a bit vector. A source encoder may choose between a bit vector and a mode bit, depending on which one is more appropriate for each source sequence.

Balkenhol et. al. [95] investigated the overhead of partial alphabets, as well as the use of multiple partial alphabets (i.e. different partial alphabets are used in different intervals of the source sequence, or alternatively the BWT output sequence). The use of multiple partial alphabets is reasonable, as each piecewise segment of the BWT output sequence typically contains only a very small fraction of the total number of

distinct alphabet symbols. These segments may be encoded more effectively if the source encoder is aware of the partial alphabet of each segment.

Balkenhol et. al. [95] considered the use of a set of sliding windows in defining the partial alphabet for each piecewise segment of the BWT output sequence. This approach is necessary as the exact transition points between the piecewise segments are unknown in practice. The lengths of the sliding windows are proportional to the average length of the segments, and are therefore proportional to the length of the source sequence.

Permutation of the symbol alphabet The BWT of a source sequence is dependent on the lexicographical order of the symbols in the symbol alphabet [16]. If the lexicographical order of the symbols changes, the BWT typically produces a different output sequence[5]. The process of changing the lexicographical order of the alphabet symbols is referred to as the permutation of the symbol alphabet. The permutation of the symbol alphabet changes the statistical characteristics of the BWT output sequence, as the symbol contexts of the source sequence are sorted differently if the lexicographical order of the symbols changes.

Suppose that contexts with similar symbol distributions are defined as being lexicographically similar to one another. The distributions of these lexicographically similar contexts would belong to segments that neighbour one another in the BWT output sequence. The source model of an adaptive source code would remain accurate during the source coding of these segments, as the empirical distribution of the symbols does not change significantly from segment to segment. The improved accuracy of the source model, which is a consequence of the permutation of the symbol alphabet, improves the effectiveness of the source code.

The permutation of the symbol alphabet was investigated by Chapin et. al. [96], as well as Kruse et. al. [97]. Chapin et. al. [96] observed a reduction in the effectiveness of a BWT–based source code that may be as high as 20% when the alphabet symbols are randomly permuted. This observation motivates a more structured approach to the permutation of the symbol alphabet.

One straightforward approach to the permutation of the ASCII alphabet is to arrange all the vowels next to one another in the lexicographical order of the alphabet symbols [96]. Multiple permutations of the alphabet symbols exist in which the vowels are arranged next to one another. One of these permutations starts with the symbols `a e i o u b c d f g`. The lowercase vowels of this permutation are the lexicographically smallest letters among the lowercase letters of the permutation. As similar letters often follow the vowels in words, alphabet reordering causes the BWT to position these letters next to one another in the BWT output sequence.

A second approach to the permutation of the symbol alphabet assumes that the source sequence contains only first–order contexts (i.e. the context of a symbol is its preceding symbol) [96]. In this approach, a histogram is constructed for the symbols that appear in each first–order context of the source sequence. The histograms are used to determine the degree of dissimilarity between the symbol distributions of contexts. A cost is associated with each pair of contexts — the cost is proportional in magnitude

[5]The BWT may produce the same output sequence in certain cases.

to the dissimilarity between the empirical symbol distributions of the two contexts. The alphabet symbols are permuted in order to reduce the total cost that is associated with lexicographically traversing the contexts. The optimal permutation minimizes the total cost.

The search for the optimal alphabet permutation of the second approach may be defined in terms of the traveling salesman problem [96]. Suppose that the contexts are associated with the cities that the salesman visits, and that the costs associated with all context pairs are calculated. The minimum cost solution to the traveling salesman problem implies an effective alphabet permutation. Four different definitions of the cost were considered by Chapin et. al. [48, 96]. These definitions include the Kullback–Leibler divergence between two distributions, as well as the number of symbol swaps that are necessary to change the frequency–ranked symbol order of one distribution to another.

Chapin et. al. [96] investigated the performance of BWT–based source codes that use the two approaches to alphabet permutation. The authors observed that the overhead associated with the complete reordering of the alphabet typically exceeds any improvement in the effectiveness of the source code. The first approach to alphabet permutation improves the effectiveness of the overall source code slightly, and it involves very little overhead [16].

Balkenhol et. al. [95] proposed an alphabet permutation technique that is a modified version of a technique that was proposed by Chapin et. al. [96]. The performance gain that is associated with this technique is relatively small, however. The technique may only be applied to certain types of source sequences, and is therefore inflexible.

The preprocessing of text data If the source encoder is informed that English text in the form of ASCII symbols is to be source coded, it may apply additional transforms to the source sequence in order to improve the effectiveness of the source code[6] [98]. These transforms are only appropriate when applied to ASCII text data and have a detrimental effect on performance when applied to data that is not ASCII text. General–purpose source codes and their implementations typically do not use these transforms. The text–specific transforms that are summarized in this section are not implemented in any of the source codes associated with this thesis, due to their restrictive assumption regarding the source sequence type.

Grabowski [99] proposed several 'reversible filters' that may be applied to a text sequence prior to it being encoded. The first filter is the capital conversion filter. This filter converts any uppercase letter at the start of a word to lowercase, and sets a flag to indicate the change. A second filter inserts a space symbol after each 'end of line' (`EOL`) ASCII symbol in the source sequence. The `EOL` symbol is typically used to terminate a paragraph of ASCII text. The motivation behind the insertion of the space symbol concerns the preceding context of the first letter of the paragraph. The `EOL` symbol is artificial, and not an appropriate context for the first letter of the paragraph. The insertion of the space symbol implies that the first letter of the paragraph is also the first letter of a word. The space symbol, when used as the preceding context of the first letter in a paragraph, may be used to predict the letter with greater accuracy.

[6] The summary presented in this section was adapted from reference [98].

The third filter proposed by Grabowski [99] concerns phrase substitution. This filter replaces groups of letters (i.e. phrases) with certain symbols. The symbols indicate which phrases the filter replaced — each distinct symbol is associated with one distinct phrase. The filter may associate the alphabet symbols that do not appear in the source sequence with phrases. The selection of an appropriate set of phrases typically improves the performance of the source code.

Grabowski [99] proposed several text–specific improvements in addition to those discussed. Deorowicz [98] claimed that these improvements have only a marginal impact on the performance of the source code. It was empirically demonstrated that the application of the three reversible filters reduced the size of the source–coded Calgary corpus text files [100] by up to 3%. However, the three reversible filters significantly degraded the performance of the source code when applied to binary files.

Awan et. al. [101,102] proposed a text–specific transform known as the length index preserving transform, or LIPT. The LIPT requires that the source encoder be informed that the source sequence is text, and that the language of the text is known. If the source encoder has a precompiled dictionary for the language of the text, the LIPT may be applied to the source sequence. The precompiled dictionary contains a list of words, and associates a codeword with each of the words.

The LIPT processes the source sequence word–by–word, starting with the first word of the sequence [101,102]. If a word of the source sequence is present in the dictionary, it is substituted with its codeword. Deorowicz [98] stated that the LIPT technique improves the effectiveness of a source code that is applied to the text files of the Calgary corpus [100] by approximately 5%. This is a significant improvement.

The preprocessing of binary data Several preprocessing techniques that are beneficial to the source coding of certain binary data files were proposed [98]. These techniques are detrimental to the performance of a source code when applied to a file that does not contain the correct type of binary data. Due to the wide variety of binary data files, these techniques are not considered in the remainder of this thesis.

3.3.2.2 The forward BWT

The BWT is a fertile research topic. Researchers proposed several improvements to the transform [16,94,96,103], and also implemented the transform in source codes for different types of data [104–107]. Some researchers proposed transforms that are related to the BWT — each of these transforms may be used as an alternative to the ordinary BWT [108–110].

Improvements to the transform The BWT may sort the cyclic shifts of the source sequence in a forward direction or a backward direction. If one direction consistently brings about better source coding performance than the other, the BWT may be improved to take advantage of this trend.

The direction in which the BWT sorts the cyclic shifts of the source sequence was investigated in the literature [16,94]. The cyclic shifts are ordinarily sorted in the forward direction by initially comparing the leading symbols of the cyclic shifts (the MSSs), followed by the second symbols of the cyclic shifts, up to the trailing symbols

(the LSSs) of the cyclic shifts. The cyclic shifts may also be sorted in the backward direction by initially comparing the trailing symbols of the cyclic shifts (the LSSs), followed by the second–to–last symbols of the cyclic shifts, and ending at the leading symbols (the MSSs) of the cyclic shifts. If the cyclic shifts are sorted in a forward direction, the source symbols that precede the same context are placed within the same piecewise segment of the BWT output sequence. If the sort is performed in the backward direction, the source symbols that follow the same context are placed within the same piecewise segment of the BWT output sequence.

It may seem appropriate to group together those symbols that follow identical contexts in data such as English text, as many letters of a sentence may be accurately predicted by examining the letters that precede them. Shannon [111] proved that there is very little difference between using contexts that precede letters or contexts that follow letters for the purpose of accurately predicting English text. This fact suggests that a source code would be more or less equally effective when applied to the output sequences of the forward–sort BWT and the backward–sort BWT, provided the source sequences are English text.

The choice of the sort direction may impact the performance of source codes that are used to encode sequences from finite–memory sources [10]. If a source sequence from any finite–memory source is reversed, the reversed sequence is equivalent to an ordinary source sequence from a different finite–memory source. This finite–memory source may be defined over a greater number of states than the original finite–memory source. As the normalized average redundancies of some universal source codes are proportional to the number of source states, the choice of the sort direction may impact the performance of a universal source code when applied to sequences from finite–memory sources.

Fenwick [16] compared the performance of BWT–based source codes w.r.t. the sort direction. The sizes of the source–coded files of the Calgary corpus [100] were used as a measure of performance. Both sort directions and both a basic and an advanced implementation of a source code were considered. Fenwick observed no significant difference between the sizes of the source–coded files that correspond to both sort directions. This observation holds true for the basic source code, as well as the advanced source code. The source coding of the `geo` file of the Calgary corpus is significantly more effective if the backward sort direction is used, however. This file contains floating–point numbers, and has a particular structure that accounts for the difference in performance. This file is a special case.

Balkenhol et. al. [94] stated that the backward sort direction should be used if the source alphabet contains approximately 256 symbols. Fenwick [16] argued that it is misleading to choose a sort direction based on the size of the alphabet. Fenwick proposed that each file be source coded using both sort directions in the BWT, and that the sort direction that brings about the best performance be used. In this approach the source decoder has to be informed regarding the choice of sort direction to successfully decode each file.

Chapin et. al. [96] proposed the reflexive–sort BWT, which is an improved version of the conventional BWT. Instead of sorting the cyclic shifts of the source sequence lexicographically in either ascending or descending order, the reflexive–sort BWT alternately uses the ascending and descending lexicographical sort orders.

The reflexive–sort BWT sorts the cyclic shifts of the source sequence as follows [96]. Assume that the BWT uses a radix–sort algorithm to sort the cyclic shifts. The cyclic shifts are first placed in a group and sorted in ascending lexicographical order according to the MSS of each cyclic shift. The group is then divided into subgroups according to the MSSs of the cyclic shifts. Each subgroup is associated with the identical MSSs of the cyclic shifts it contains. The cyclic shifts in each subgroup are subsequently sorted according to the second MSS of each cyclic shift, but only the cyclic shifts of every second subgroup (in the ordered list of subgroups) are sorted in ascending order. The cyclic shifts of the remaining subgroups are sorted in descending order. These reflexive–sort BWT proceeds by repeatedly dividing the subgroups and sorting the cyclic shifts within each subgroup until each subgroup contains only a single cyclic shift. The sorted cyclic shifts are obtained by traversing the final subgroups.

By alternating the sort order, the cyclic shifts of the source sequence are sorted in an order that resembles the sequence of codewords of a Gray code [16]. Each successive codeword of a Gray code differs from the previous codeword with respect to a single symbol — the remaining symbols of the codewords are identical. The goal of the reflexive–sort BWT is to place symbols from contexts with more similar symbol distributions next to one another in the BWT output sequence. This improvement has a beneficial effect on the performance of the source encoder that encodes the transformed sequence.

Butterman et. al. [103] proposed an error–resistant version of the BWT that is useful in the transmission of data over noisy communication channels. Adaptive lossless source codes typically suffer from error propagation. Error propagation refers to a single bit error in the source–coded sequence causing several of the remaining bits in the sequence to be incorrectly decoded. Source code implementations that make use of the BWT typically discard those source–coded blocks that contain one or more bit errors, as the reverse BWT scatters bit errors throughout these blocks [42].

The error–resistant version of the BWT, as proposed by Butterman et. al. [103], encodes additional information as part of the transformed sequence. The additional information enables the reverse BWT to recover at least certain sections of the source sequence if bit errors are present in the transformed sequence. The error–resistant nature of this version of the BWT was demonstrated empirically by source coding files from the Calgary [100] and Canterbury [112] corpora, introducing bit errors to the source–coded files, and decoding the corrupted files.

Isal et. al. [106] proposed an implementation of the BWT that is applied to sequences of phrases, instead of symbols. The source encoder compiles a dictionary that contains all the distinct phrases appearing in the source sequence. It then substitutes each phrase in the source sequence with an integer that equals its index in the dictionary. The BWT is finally applied to the sequence of integers. The output of the BWT contains piecewise segments of integers that are asymptotically i.i.d. The overall source code is flexible, as the dictionary entries may be defined as words, symbol digrams, syllables, or any other combination of symbols and letters. The dictionary of the source encoder is encoded and constitutes part of the source–coded sequence.

The transformation of different types of data Some researchers successfully implemented the BWT in source codes for data that is not text. Elsayed et. al. [104] implemented the BWT in a lossless source code for audio signals. The source encoder applies the BWT to integer samples that represent the audio signal. A recency–rank code is applied to the BWT output sequence, after which a Huffman encoder or arithmetic encoder is used to encode the recency–rank encoder output sequence. The BWT–based source code outperforms several lossless audio source codes, including a source code that uses the integer discrete cosine transform, as well as a source code that uses the integer wavelet transform.

Adjeroh et. al. [105] implemented the BWT in a source code for DNA sequences. The BWT is applied to the source sequence, and its output sequence is expressed as a suffix tree. The suffix tree is used to identify patterns that repeat in the DNA sequence. The source encoder may encode the repeating patterns in a more effective manner, thereby improving its performance.

Alternative transforms Mantaci et. al. [108] proposed a transform that may be regarded as an extension of the BWT. The proposed transform is applied to a multiset of words. All cyclic shifts of the words are sorted lexicographically. If a comparison tie between two words is not resolved at the end of one of the words, that word is repeated, and the comparison continues. The final letters of the lexicographically–sorted cyclic shifts constitute the output sequence of the transform.

Arnavut et. al. [109, 110] proposed a series of transforms that may be regarded as generalizations of the Burrows–Wheeler transform. The first transform [109] is known as the lexical permutation sorting algorithm. Arnavut et. al. [110, 113] generalized the lexical permutation sorting algorithm to multiset permutations, and introduced the linear–order transform.

Arnavut et. al. [113] demonstrated that the linear–order transform is computationally less complex than the BWT. The linear–order transform of certain types of data, such as pseudo–colour images, may be source coded with nearly the same degree of effectiveness as the Burrows–Wheeler transform of the data. Arnavut et. al. [114] proposed a source code for electrocardiogram (ECG) signals that uses the linear–order transform. This source code is more effective than a similar BWT–based source code [114], the `gzip` algorithm [58], as well as the `shorten` lossless waveform encoder.

3.3.2.3 The recency–rank encoder

This section contains a summary of several topics concerning the recency–rank code. The recency–rank code has received considerable attention in the literature. Researchers proposed a large number of improvements and modifications to the recency–rank code. Some researchers investigated the possibility of omitting the recency–rank encoder from BWT–based source codes. Several alternatives to the recency–rank code were also proposed.

The aim of the summary is to highlight some of the more significant improvements and alternatives to the recency–rank code, and not to be exhaustive. To motivate some of the improvements to the recency–rank code, it is first necessary to introduce the list update problem.

The list update problem This summary of the list update problem was adapted from the summary of reference [98]. The list update problem (LUP) is defined as follows [115]. A list of items is defined, and a sequence of requests regarding the items in the list is specified. A request may be an insertion, deletion or access to an item in the list. Only the requests to access an item in the list need be considered for the purpose of this thesis.

A cost is associated with servicing each request in the sequence. The cost of servicing a request to access an item in the list equals the item's index in the list. After servicing each request, the requested item may be moved any number of positions closer to the top of the list, and at no cost. This move is referred to as a free transposition. Any other list item that was located between the original position of the requested item and the bottom of the list may also be moved closer to the top of the list, but at unity cost. This move is known as a paid transposition.

An algorithm for solving the LUP services the requests to access items in the list, and produces a sequence of transpositions of items in the list. The total cost of a solution equals the sum of the number of paid transpositions and the total cost of servicing the requests. An appropriate algorithm for solving the LUP should attempt to minimize the total cost of its solution.

An online algorithm for solving the LUP does not have access to the entire sequence of requests before it starts to service them. In contrast, an offline algorithm has access to the entire sequence of requests before it starts to solve the LUP. An optimal offline algorithm for solving the LUP produces a solution that has the smallest total cost among all possible solutions of the LUP. The optimal offline algorithm has a computational complexity that increases exponentially with respect to the length of the request sequence.

Online algorithms for solving the LUP may be divided into two classes. A deterministic online algorithm for solving the LUP always produces the same solution in response to the same sequence of requests. A randomized online algorithm for solving the LUP may produce different solutions if the same sequence of requests is presented to it repeatedly.

The theory behind the LUP and its solution was not initially applied in the field of source coding [48]. It was used to develop procedures for updating identifier lists and hash tables, thereby enabling efficient access to the data these structures contain. It was also applied to efficiently update simpler data structures that are employed in situations where sufficient memory is not available for the use of more sophisticated data structures. The LUP and the recency–rank encoder were only recently used to develop more effective source codes [116].

The recency–rank encoder may be interpreted as an algorithm for solving the LUP. The relationship between the LUP and the recency–rank encoder may be clarified by interpreting the output sequence of the BWT as a sequence of requests to access symbols in an ordered list, which corresponds to the rank array. The recency–rank encoder produces the sequence of costs (ranks) that is associated with the sequence of requests, and moves the requested symbol to the front of the list after each request. As the recency–rank encoder only moves the requested symbol to the front of the rank array, it does not perform any paid transpositions.

The goal of the recency–rank encoder is to produce a sequence of integers with a distribution that is as biased as possible, thereby improving the effectiveness of the source encoder that is applied to its output [16]. One approach to producing a sequence of costs with a distribution that is biased towards zero involves the minimization of the total cost of servicing the requests for items in the list. This approach does not guarantee a sequence of biased costs, but its application typically does produce this type of sequence.

The conventional LUP does not specify a distribution for the sequence of requests. The recency–rank encoder may reasonably assume that the BWT output is p.i.i.d., however. Prior knowledge regarding the distribution of the requests may be used to improve the performance of an algorithm for solving the LUP.

Deorowicz [98] observed that the recency–rank encoder is remarkably effective when applied in a BWT–based source code, despite its simplicity. The effectiveness of the recency–rank encoder is a consequence of the distribution of the symbols in the BWT output sequence — those symbols that were recently requested in the rank array are likely to be requested again in the near future.

Improvements to the recency–rank code One characteristic of the recency–rank encoder that was addressed in the literature is that it moves an improbable symbol to the front of the rank array immediately upon encountering it in the BWT output sequence [16]. An improbable symbol will likely remain in the front half of the rank array for a considerable period of time, despite the fact that it is unlikely to appear again in the remainder of the BWT output sequence. Unlikely symbols may clutter the front of the rank array, momentarily pushing some of the more probable symbols towards the back of the array.

The clutter of unlikely symbols at the front of the rank array has a balancing effect on the distribution of the ranks that the recency–rank encoder produces [16]. This effect is detrimental to the performance of the source encoder that is applied to the sequence of ranks. Burrows et. al. [9] suggested that the recency–rank encoder might be more effective if it refrains from immediately moving all symbols that it encounters to the front of the rank array.

Several authors proposed algorithms that avoid moving each symbol to the front of the rank array upon encountering it. These algorithms may be interpreted as variants of the recency–rank encoder. Some of the more noteworthy algorithms are discussed in what follows.

The ‘move–one–from–front’ algorithm [16, 94] attempts to reduce the clutter of unlikely symbols at the front of the rank array. A symbol that is not present in the first two positions of the rank array is moved to the second position of the rank array as it is encountered in the BWT output sequence. The symbol in the second position of the rank array is moved to the front of the rank array upon being encountered in the BWT output sequence.

The move–one–from–front algorithm delays the movement of symbols to the front of the rank array, but it does not completely eliminate the clutter at the front of the rank array [16, 94]. The usefulness of the move–one–from–front algorithm is evident when processing long runs of a certain alphabet symbol that are separated by a few distinct

symbols. The run symbol remains at the front of the rank array as this sequence is transformed, and more zero–valued ranks are produced than would be the case had the conventional recency–rank code been used.

The modified move–one–from–front algorithm [48, 95, 117] is identical to the move–one–from–front algorithm, except regarding the movement of symbols in the second position of the rank array. If the symbol in the second position of the rank array is encountered in the BWT output sequence, it is moved to the front of the rank array provided that the previous rank that the algorithm produced does not equal zero. If the previous output rank equals zero, the encountered symbol remains in the second position of the rank array. This modification stops any distinct symbol within a long run of identical alphabet symbols from moving the run symbol to the second position of the rank array.

The 'move–one–from–front–2' algorithm [16, 48] is a variation of the move–one–from–front algorithm. All symbols that are encountered are moved to the front of the rank array, except if

- the encountered symbol is not present in the first two positions of the rank array, or
- at least one of the previous two ranks produced by the algorithm equals zero.

If at least one of these conditions is satisfied, the encountered symbol is moved to the second position of the rank array. If the encountered symbol is present in the first position of the rank array, it remains in this position.

The 'sticky' move–to–front algorithm [16] attempts to avoid clutter at the front of the rank array by moving certain symbols towards the back of the rank array. Upon encountering a symbol, the algorithm moves it to the front of the rank array. If the next symbol in the BWT output sequence does not equal the symbol at the front of the rank array, the symbol at the front of the rank array is moved towards the back of the rank array by approximately 40% of the number of positions it moved to reach the front of the rank array. It appears that the value of 40% was selected through trial and error.

The 'move–to–front–when–even' algorithm [118] selectively updates the rank array in an attempt to delay the movement of symbols to the front of the rank array. The algorithm records the number of times each alphabet symbol was encountered in the BWT output sequence as it transforms this sequence. If a certain alphabet symbol is encountered in the BWT output sequence, and that symbol had been encountered $i-1$ times previously in the sequence, the symbol is moved to the front of the rank array provided that i is an even number. If i is not an even number, the symbol remains at its current position in the rank array.

The 'move–fraction' algorithm [93, 118, 119] updates the rank array by moving the encountered symbol only a fraction of the distance to the front of the rank array. Suppose that the algorithm encounters the ith symbol of the rank array in the BWT output sequence. This symbol is moved a total of $\lceil i/k \rceil - 1$ positions towards the front of the rank array, where k is an integer parameter of the algorithm (with $k \geq 1$). Bachrach et. al. [119] implemented the move–fraction algorithm with $k \geq 2$ in a BWT–based source code, and found that the code performed poorly. Fenwick [93] observed a

drop in source coding performance upon replacing the recency–rank encoder with the move–fraction algorithm in a BWT–based source code with k equal to 8 or 32.

The 'most recent item' algorithm [48, 120] updates the rank array according to those alphabet symbols that occurred at least once since the previous occurrence of the symbol that is to move in the rank array. Suppose that the algorithm encounters the symbol `a` in the BWT output sequence. It compiles a list of all symbols that were encountered since the previous occurrence of symbol `a` in the BWT output sequence, and finds the symbol in the list with the highest rank. The symbol `a` is placed immediately behind this symbol in the rank array. If a symbol is encountered twice in succession, it is moved to the front of the rank array.

The 'best x of $2x-1$' algorithm [16, 121] updates the rank array differently than other variants of the recency–rank encoder. Suppose that the algorithm has access to a list of all possible alphabet symbol pairs. This list also contains, for each symbol pair, the number of times each of the two symbols appeared in the previous $2x-1$ occurrences of either of the two symbols. Upon encountering a certain symbol in the BWT output sequence, the algorithm examines all symbol pairs in the list that contain the alphabet symbol that was encountered. All symbol pairs in which the encountered symbol has a count of x or more are marked. The algorithm subsequently finds the symbol with the lowest rank among all the symbols in the marked pairs. The encountered symbol is moved in front of this symbol in the rank array.

The 'transpose' algorithm [48,120] is a well known deterministic algorithm for solving the LUP, and may be used as an alternative to the recency–rank code. It updates the rank array in an extremely conservative fashion. After encountering a symbol in the BWT output sequence, the algorithm moves the symbol a single position closer to the front of the rank array. The more likely symbols slowly move to the front of the rank array as the transformation of the sequence proceeds, while the less likely symbols eventually drift to the back of the array.

The 'time–stamp' algorithm [98,122] is an algorithm for solving the LUP that may be used as an alternative to the recency–rank code. The deterministic version of the time–stamp algorithm, which is referred to as TS(0) in the literature, is relevant to source coding. In order to illustrate the time–stamp algorithm, assume that it encounters the alphabet symbol `a` in the BWT output sequence. The time–stamp algorithm first produces the index of the symbol `a` in the rank array as output, and next updates the rank array. The rank array is updated by first compiling a list of all alphabet symbols that occurred at most once since the previous occurrence of the symbol `a` in the BWT output sequence. The algorithm finds the symbol in the list with the lowest rank, and moves the symbol `a` in front of this symbol in the rank array. Upon encountering a distinct alphabet symbol for the first time in the BWT output sequence, the time–stamp algorithm does not update the rank array.

Albers et. al. [123] investigated the performance of the TS(0) algorithm analytically, and stated that it is typically more effective than the recency–rank encoder for the purpose of source coding. The TS(0) algorithm was implemented in a BWT–based source code, but this implementation proved to be less effective than a similar implementation that uses the recency–rank code. The effectiveness of each implementation was judged according to the total size of the source–coded files that it produces when

applied to the files of the Calgary corpus [100].

The 'sort–by–rank' (SBR) algorithm [118, 124] is a deterministic algorithm for solving the LUP that is relevant to source coding. It defines a rank array of distinct alphabet symbols that is updated after transforming each symbol of the BWT output sequence. The algorithm calculates two integers for each symbol of the alphabet prior to updating the rank array.

After producing the rank of the ith symbol of the BWT output sequence, the sort–by–rank algorithm calculates the integers $w_1(y, i)$ and $w_2(y, i)$ for each alphabet symbol y [118, 124]. The first integer, $w_1(y, i)$, equals the index of the last occurrence of the alphabet symbol y in the BWT output sequence, considering all symbols up to the ith symbol of the sequence. The second integer, $w_2(y, i)$, is similarly defined, except that the index of the second–to–last occurrence of the symbol y is used, instead of its last occurrence. If the symbol y is not present in the first i symbols of the BWT output sequence, the integers $w_1(y, i)$ and $w_2(y, i)$ are set to zero.

The SBR algorithm subsequently calculates the number of symbols that have occurred since the last occurrence of each symbol of the alphabet [118, 124]. This quantity may be expressed as $s_1(y, i) = i - w_1(y, i)$. The number of symbols that have occurred since the second–to–last occurrence of each alphabet symbol is defined as $s_2(y, i) = i - w_2(y, i)$. This quantity is also calculated for each symbol of the alphabet.

The SBR algorithm recalculates the rank of each symbol of the alphabet, and updates the rank array accordingly [118, 124]. Each alphabet symbol y is assigned a weight that is calculated according to the equation

$$r_\alpha(y, i) = (1 - \alpha)s_1(y, i) + \alpha s_2(y, i)^2, \tag{3.17}$$

where α is a parameter between zero and one, and i is the index of the symbol in the BWT output sequence that was most recently transformed. Upon calculating the weights $r_\alpha(y, t)$ for all alphabet symbols, the updated symbol ranks (and the updated rank array) are obtained by sorting the symbols in nondecreasing order according to their weights.

The SBR algorithm may be interpreted as a generalization of the recency–rank encoder and the time–stamp algorithm [118]. If the parameter α is set to zero, the output of the SBR algorithm equals the output of the recency–rank encoder. The output of the time–stamp algorithm is equal to the output of the SBR algorithm if the parameter α is set to unity. For values of α that lie between zero and unity, the output of the SBR algorithm has characteristics of both the recency–rank encoder output and time–stamp algorithm output.

Dorrigiv et. al. [118] implemented the SBR algorithm in a BWT–based source code. The effectiveness of the implementation, as a function of the parameter α, was investigated. The implementation encoded the majority of source sequences most effectively if the parameter α was set equal to either zero or unity, and not a fractional value. Dorrigiv et. al. [118] suggested that the source encoder should choose between the values of zero and unity for the parameter α by first determining which of these two values brings about the shortest source–coded sequence. The value of α that is associated with the shortest source–coded sequence is saved as part of this sequence, as the source decoder requires the correct value of α in order to successfully recover the BWT output sequence.

Alternatives to the recency–rank code Some researchers proposed alternatives to the recency–rank encoder. These alternative transforms have the same purpose as that of the recency–rank encoder, which is to convert the output of the BWT into a sequence of integers with a biased, stationary distribution. The alternative transforms differ significantly from the recency–rank encoder regarding their approaches to the conversion, however. The more relevant alternatives to the recency–rank encoder are summarized in this section.

Frequency count transform The frequency count transform [118] is an algorithm for solving the LUP. The algorithm defines an array that contains the distinct alphabet symbols, and records the number of times each distinct alphabet symbol appears in the BWT output sequence. The frequency counts are recorded as the transform of the BWT output sequence proceeds in a symbol–by–symbol manner.

The algorithm replaces each symbol of the BWT output sequence with its index in the array of alphabet symbols [118]. The algorithm maintains the array of alphabet symbols in nonincreasing order according to the symbol frequency counts. The array is updated after the transformation of each consecutive symbol.

The symbols that are more likely to occur in the BWT output sequence occupy the front of the array. This fact suggests that the frequency count transform would likely produce a large number of small integers as output. The frequency count transform is often slow in adapting to a change in the symbol distribution of the BWT output sequence, however. Its slow rate of adaptation is due to the fact that it only uses the frequency counts of the symbols during each update of the array, and that it does not consider the recency of the symbols at all.

Dorrigiv et. al. [125] found that an implementation of a BWT–based source code that uses the frequency count transform is less effective than identical implementations that use the time–stamp algorithm (refer to page 69) or the recency–rank encoder. The effectiveness of each implementation was judged according to the total size of the source–coded files that it produces when applied to the files of the Calgary corpus [100]. This observation suggests that an effective alternative to the recency–rank encoder should incorporate a symbol recency metric in its updates of the array.

Weighted frequency count transform Deorowicz [98, 126] observed that the recency–rank encoder does not take the frequency count of any symbol into consideration as it updates the rank array. By disregarding the frequency counts, the recency–rank encoder may move an unlikely symbol to the front of the rank array upon encountering it in the BWT output sequence. The movement of unlikely symbols to the front of the rank array may be reduced by considering both the frequency counts and the recency of the symbols during each update of the array.

Deorowicz [98, 126] proposed an alternative to the recency–rank encoder that updates the rank array according to the frequency counts and the recency of the symbols. This novel transform is referred to as the weighted frequency count (WFC) transform.

The WFC transform updates the rank array prior to transforming each consecutive symbol of the BWT output sequence [98, 126]. The rank of each alphabet symbol in the rank array is recalculated using a mathematical function. If the WFC transform

recalculates the symbol ranks with an appropriate mathematical function, the distribution of the ranks it produces may have a greater bias than the distribution of the recency–rank encoder output.

The WFC transform associates a weight with each symbol of the alphabet [98, 126]. Its initial step in updating the rank array is the recalculation of the weight of each alphabet symbol. The alphabet symbols are subsequently ranked according to nonincreasing weight. The rank array is finally updated according to the newly–calculated symbol ranks.

The weight of each alphabet symbol equals the sum of several smaller weights [98, 126]. Each of the smaller weights corresponds to one occurrence of the particular symbol in the BWT output sequence. The smaller weights are proportional to the recency of the symbols. Immediately prior to transforming the ith symbol of the BWT output sequence, the WFC transform calculates the weight of the jth distinct alphabet symbol a_j according to the function

$$W_i(a_j) = \sum_{k=1}^{i-1} w(i-k)\delta(a_j, x_k), \tag{3.18}$$

where x_k is the kth symbol in the BWT output sequence, and

$$\delta(p, q) = \begin{cases} 1 & \text{if } p = q, \\ 0 & \text{otherwise.} \end{cases} \tag{3.19}$$

The term $w(i-k)$ of equation 3.18 represents the smaller weight that is associated with the kth symbol of the BWT output sequence. Each of the smaller weights is calculated according to the equation

$$w(t) = w_f(t), \tag{3.20}$$

where $w_f(t)$ is referred to as the weight function.

Deorowicz [126] considered several weight functions. In order to select a suitable weight function from among those considered, Deorowicz implemented the WFC transform in identical BWT–based source codes, and specified a different weight function for each code. The files of the Calgary corpus [100] were encoded using the source codes, and the effectiveness of each weight function was judged from the sizes of the source–coded files. One of the most effective weight functions that was considered is expressed as

$$w_f(t) = \begin{cases} 1 & \text{if } t = 1, \\ \frac{1}{pt} & \text{if } 1 < t \leq 64, \\ \frac{1}{2pt} & \text{if } 64 < t \leq 256, \\ \frac{1}{4pt} & \text{if } 256 < t \leq 1024, \\ \frac{1}{8pt} & \text{if } 1024 < t \leq t_{\max}, \\ 0 & \text{if } t > t_{\max}, \end{cases} \tag{3.21}$$

where p is a parameter, and $t_{\max}$ specifies the total number of symbols of the BWT output sequence that are assigned nonzero weights. The parameters p and $t_{\max}$ were assigned values of 4 and 2048 during the source coding of the Calgary corpus files.

The WFC transform is computationally more complex than the recency–rank encoder, as it updates the entire rank array after transforming each symbol [86, 98].

Deorowicz [98] proposed that the range of the weight function be quantized to integer powers of two in order to reduce the computational complexity of the transform. By quantizing the range of the weight function, several of the smaller weights of the symbols in the BWT output sequence do not change between consecutive updates of the rank array. These weights need not be recalculated. The quantization of the range of the weight function does sacrifice some of the precision of the WFC transform, however. The loss of precision may reduce the effectiveness of the source code that uses the WFC transform.

Abel [86] proposed a more effective weight function than the function of equation 3.21. The effectiveness of the weight function was judged according to the performance of a BWT–based source code that uses the WFC transform and the weight function. The proposed weight function is defined over twelve nonzero levels, instead of the five nonzero levels of equation 3.21.

Abel [86] proposed a variation of the WFC transform that is known as the advanced weighted frequency count (AWFC) transform. The AWFC transform introduces a dependency between the calculation of the smaller weights and the symbol distribution of the input sequence of the WFC transform. The purpose of this dependency is to improve the source coding of individual source sequences.

Abel [86] proposed an adaptive weight function with three parameters. Two of the parameters were selected through trial and error according to the empirical performance of the source code. The third parameter equals the percentage of distinct alphabet symbols with individual frequency counts that exceed twice the symbol allocation. The symbol allocation is defined as the fraction n/m, where n equals the length of the source sequence, and m is the number of distinct symbols in the alphabet.

Inversion frequencies transform Arnavut et. al. [127] proposed the inversion frequencies transform (IFT) as an alternative to the recency–rank code. This summary of the IFT was adapted from references [86, 98].

The IFT associates a codeword with each symbol in the BWT output sequence. Each codeword depends on the symbol's index in the BWT output sequence, as well as the symbol's position in the lexicographical order of distinct alphabet symbols. Suppose that the codeword of symbol x_i in the BWT output sequence is to be calculated, and assume that $x_i =$ `c`. The codeword of this symbol equals the number of symbols that are lexicographically larger than symbol `c`, but which occur between symbol x_i and the previous occurrence of the symbol `c` in the BWT output sequence.

Instead of substituting each symbol of the BWT output sequence with its codeword, the IFT first produces a single integer for each distinct symbol of the alphabet. Each integer equals the number of appearances of its alphabet symbol in the BWT output sequence — these integers are required by the source decoder to successfully perform the reverse transform. The IFT finally produces, for each distinct alphabet symbol in lexicographical order, the codewords that are associated with all occurrences of the alphabet symbol in the BWT output sequence. In order to correctly reverse the transform, the codeword of the first occurrence of each distinct alphabet symbol in the BWT output sequence is set equal to the index of the symbol in the BWT output sequence. Efficient implementations of both the forward and reverse transforms were

presented in the literature [128].

Identical symbols that are located in the vicinity of one another in the BWT output sequence are assigned integer codewords with small values by the IFT. The IFT therefore produces integers that are biased towards zero when applied to typical BWT output sequences. There is a significant difference between the distribution of the IFT output and the distribution of the recency–rank encoder output, however.

The difference between the output distributions of the IFT and the recency–rank encoder is due to the fact that the IFT codewords of lexicographically large symbols are typically small integers, regardless of their position in the BWT output sequence. This statement follows from the fact that there are few distinct alphabet symbols that are lexicographically larger than these symbols. As the codewords of lexicographically large symbols are located next to one other in the IFT output sequence, the source encoder may encode these symbols effectively. Deorowicz [126] remarked, however, that the overhead associated with the representation of the symbol frequency counts may limit the usefulness of the IFT to longer sequences only.

Abel [86] proposed an improvement to the IFT that involves a change in the lexicographical order of the symbol alphabet. Assume that the lexicographical order of the symbol alphabet is changed so that the frequently occurring symbols of the BWT output sequence are considered lexicographically smaller than the infrequently occurring symbols. As the BWT output sequence contains fewer symbols that are lexicographically large, the less frequent symbols are assigned smaller integer codewords. The BWT output sequence also contains a greater fraction of symbols that are lexicographically small, however — the IFT may possibly produce a greater fraction of integers with larger values than before. It is not immediately obvious whether to assign lexicographical ranks to symbols in the ascending or descending order of symbol frequency counts in order to bias the distribution of the IFT output integers towards zero.

Abel [86] implemented the IFT in a BWT–based source code, and assigned lexicographical ranks to symbols that are arranged in both the increasing and decreasing order of frequency counts. This implementation was used to determine which method of assigning lexicographical ranks to symbols typically produces the most effective source code. The implementation of the source code was used to encode the files of the Calgary corpus [100]. The majority of the Calgary corpus files were source coded more effectively when lexicographical ranks were assigned to symbols in the ascending frequency count order. The only files of the Calgary corpus that were encoded more effectively when lexicographical ranks were assigned to symbols in the order of descending frequency counts were the binary files `geo` and `obj1`.

Abel [86] proposed a heuristic for selecting an appropriate order in which to assign lexicographical ranks to symbols. The heuristic states that lexicographical ranks should be assigned to symbols in the order of decreasing frequency counts if the percentage of alphabet symbols with individual frequency counts exceeding twice the average symbol allocation is less than 10%[7]. If this is not the case, the lexicographical ranks are assigned to symbols in the order of ascending frequency counts.

[7]Refer to the discussion regarding the AWFC (page 73) for the definition of the average symbol allocation.

Distance coding transform Binder [129] proposed the distance coding transform (DCT) as an alternative to the recency–rank code. The DCT is related to the interval code that was proposed by Elias [84]. This summary of the DCT was adapted from reference [98].

The DCT is applied to the BWT output sequence, and produces two concatenated integer sequences as output. The transform initially finds the first occurrence of each distinct alphabet symbol in the BWT output sequence. The integer indices of the first occurrences of all distinct alphabet symbols in the BWT output sequence constitute the first integer output sequence of the DCT. The integers of the first output sequence are typically arranged in lexicographical order w.r.t. the symbols they represent.

The second output sequence of the DCT contains one integer for each symbol of the BWT output sequence. The DCT processes the BWT output sequence in a symbol–by–symbol fashion, starting at the front of the sequence and ending at the back of the sequence. The transform finds the number of symbols that are located between each symbol of the BWT output sequence and its next occurrence in the BWT output sequence. If a total of k symbols are located between a certain symbol and the next occurrence of the same symbol in the BWT output sequence, the transform produces the integer $k+1$ as output. Upon encountering the final instance of any symbol in the BWT output sequence, the transform produces the integer zero.

The DCT produces small integers when applied to identical alphabet symbols that appear in close proximity to one another. The distribution of the DCT output, assuming that the DCT is applied to typical BWT output sequences, is biased towards the smaller integers. This is due to the fact that the BWT output sequence consists of p.i.i.d. symbols, and that each piecewise segment has a biased symbol distribution (it is assumed that the BWT is applied to a source sequence from a finite–memory source or a tree source).

Binder [129] proposed three improvements to the DCT, as summarized by Deorowicz [98]. The first improvement concerns the integers of the second output sequence of the DCT. The DCT may exclude all symbol pairs that it already encountered when counting the number of symbols that appear between consecutive occurrences of the same symbol in the BWT output sequence. The BWT output sequence may be successfully recovered from the DCT output sequence in this case, as the reverse transform has knowledge of the locations of the symbol pairs it encountered previously.

The second improvement concerns the final run of zeros at the end of the transformed sequence [98]. These trailing zeros are superfluous, and may be omitted. The third improvement concerns runs of identical symbols in the BWT output sequence. The DCT need not produce any integers for these symbols, and may proceed to the final symbol in the run. This improvement yields a reversible transform, as the reverse DCT is aware that no other symbols appear within the run. It may therefore insert the correct number of missing run symbols.

Switching algorithms Chapin [48, 121] proposed that different algorithms for solving the LUP be used to transform different intervals of the BWT output sequence. This proposal is motivated by the observation that some algorithms for solving the LUP are more effective when applied to certain intervals of the sequence than other

algorithms.

Chapin [48, 121] recommended that the conventional recency–rank encoder be applied to shorter intervals. The best two of three algorithm was proposed for the purpose of transforming longer intervals. In order to transform the BWT output sequence effectively, the implementation of the source code should switch between the algorithms in a dynamic fashion. The switching should be carried out in such a manner that the source decoder is able detect each switch, as this is necessary to successfully recover the sequence.

Chapin [48, 121] employed a switching algorithm that was proposed by Volf et. al. [130] to switch between the recency–rank encoder and its alternatives. The switching algorithm estimates the lengths of the source–coded intervals that would be obtained if each transform was applied to each interval [48]. A dynamic program is used to keep track of the estimates. The switching algorithm finally selects the transforms that brought about the shortest source–coded sequence.

Postprocessing of the recency–rank encoder output Balkenhol et. al. [95] proposed a technique for processing the output of the recency–rank encoder. The postprocessing of the recency–rank encoder output has the purpose of improving the effectiveness of the source encoder that follows the recency–rank encoder in the BWT–based source code.

The postprocessing of the recency–rank encoder output produces two rank sequences [95, 98]. The first sequence is identical to the recency–rank encoder output sequence, but with all ranks higher than two replaced with the rank two. The first sequence correctly identifies those symbol ranks that are equal to zero and one, as well as the positions of the symbol ranks that are higher than one in the recency–rank encoder output sequence. The second sequence is equal to the recency–rank encoder output sequence, but with all ranks of zero and one omitted from the sequence. The second sequence contains the true values of those symbol ranks that were identified as being higher than one in the first sequence.

The ratio of the frequency counts of the most likely rank and the least likely rank in the recency–rank encoder output sequence is typically quite large [16]. By replacing all ranks in the recency–rank encoder output sequence that are higher than two with a rank of two, the alphabet associated with the first postprocessed sequence is reduced to three ranks (i.e. zero, one and higher than one). The frequency count associated with the third rank of the alphabet (i.e. the higher–than–one rank) equals the sum of the frequency counts of all ranks that are higher than one. The ratio of the frequency counts of the most likely rank and the least likely rank is significantly smaller in the first postprocessed sequence than in the ordinary recency–rank encoder output sequence.

The abovementioned property of the first postprocessed sequence is beneficial to adaptive source encoders [16]. Adaptive source encoders need to maintain a good trade-off between rapid adaptation to changing source statistics and the accuracy of the rank probability estimates. An additional advantage of the postprocessing step is that the source coding of the first and second postprocessed sequences may be performed (and optimized) separately from each other [95].

One advantage of the postprocessing step concerns the transition points between the

piecewise segments of the BWT output sequence [95]. Each transition point usually appears as a short burst of the higher–than–one rank in the first postprocessed sequence. The locations of the transition points may in some cases be estimated from the first postprocessed sequence, and used to improve the source coding of the postprocessed sequence.

The postprocessing step has the disadvantage of lengthening the sequence of ranks that is to be source coded [95]. The number of additional ranks that need to be encoded equals the number of ranks in the second postprocessed sequence

3.3.2.4 The source encoder

Researchers applied several source encoders to the recency–rank encoder output sequence of BWT–based source code implementations with varying degrees of success. These encoders include universal integer encoders [16], adaptive Huffman encoders [9], and adaptive arithmetic encoders such as PPM encoders [45] and Witten's CACM encoder [89].

Fenwick [16] observed that source encoders with multiple–context models (such as the PPM encoder [45]) are typically ineffective when applied to the recency–rank encoder output. A thorough characterization of the BWT and recency–rank encoder output sequences is necessary in order to design effective source encoders for BWT–based source codes, and to understand why certain source encoders perform poorly. This section is a summary of several source encoders and their implementation in BWT–based source codes.

Initial source encoders Burrows et. al. [9] proposed that either a Huffman encoder or an arithmetic encoder be used to encode the recency–rank encoder output in the original BWT–based source code. Burrows et. al. implemented a Huffman source encoder in the original BWT–based source code. The Huffman encoder updates the Huffman tree for each consecutive 16 kilobyte block of the rank sequence. This source encoder is able to adapt to changes in the rank distribution, provided the changes do not occur too rapidly.

Fenwick [16, 93] investigated a BWT–based source code that uses the CACM encoder of Witten et. al. [89] to encode the sequence of ranks. The effectiveness of the proposed code is within 1.6% of the effectiveness of the PPMC implementation [50], where effectiveness refers to the reduction in the size of the Calgary corpus data set [100].

Fenwick [93] investigated the performance of a BWT–based source code that uses the DCC–95 implementation of an arithmetic encoder [90]. The BWT–based source code that uses the DCC–95 arithmetic encoder proved to be significantly less effective than the BWT–based source code that uses the CACM encoder [89]. The DCC–95 encoder was designed for use in PPM source code implementations — this assumption regarding its use is the reason for its lack of performance in BWT–based source code implementations [93].

A PPM source code implementation defines multiple contexts in its model of the source [16, 93]. The DCC–95 encoder infrequently rescales the symbol counts in order to maintain an accurate multiple–context source model. The CACM implementation

that Fenwick [93] investigated assumes that a single–context source model is used. This encoder rescales the symbol counts more frequently.

A BWT–based source code implementation that uses the CACM encoder is more successful at adapting to statistical changes in the recency–rank encoder output sequence than an implementation that uses the DCC–95 encoder, due to its frequent rescaling of the symbol counts [93]. Recall that recency–rank encoder output sequences contain several high–entropy bursts, which correspond to transitions between certain piecewise segments of the BWT output sequence. The CACM encoder is more effective than the DCC–95 encoder when used to encode these sequences in a BWT–based source code implementation, as it adapts more successfully to the high–entropy bursts.

The conventional multiple–context source model is not an accurate representation of the recency–rank encoder and its output [16]. Some BWT–based source code implementations maintain only a single–context source model, and rely on the source encoder to update the model in an appropriate fashion.

The bzip2 and bred3 implementations This summary of the `bzip2` and `bred3` source code implementations was adapted from reference [48]. The predecessor of the `bzip2` command–line compression and archiving utility, `bzip`, encodes the rank sequence using an arithmetic code [87]. The subsequent version of the utility, `bzip2`, encodes the rank sequence using Huffman codes, due to patent–related restrictions on the use of arithmetic codes. The `bzip2` implementation is closely related to the `bred3` implementation [131], which is summarized in this section.

The `bred3` implementation is similar to the elementary BWT–based source code of figure 3.5 on page 53. The recency–rank encoder output sequence of the `bred3` implementation is run–length encoded. The output of the run–length encoder is uniformly partitioned into blocks of a certain length.

The `bred3` implementation maintains a set of several Huffman codes. It is assumed for the purpose of this discussion that the set contains eight Huffman codes. The `bred3` algorithm iterates over the blocks of integers and selects the most appropriate Huffman code that should be used to encode each block. The decision regarding which Huffman code to use for source coding a block of integers is based on a goodness–of–fit metric between the code and the block.

The `bred3` algorithm encodes each block of integers using the appropriate Huffman code, and records the length of each encoded integer block. It adds the overall length of the source–coded integer blocks to the length of the Huffman code tables, as well as the lengths of the selectors and delimiters that are associated with each block. This total may be interpreted as the cost of encoding the sequence of blocks.

After calculating the cost of encoding the blocks, the `bred3` algorithm merges some of the original, uncoded blocks of integers according to the particular Huffman codes that were assigned to the blocks. The algorithm proceeds by creating a total of eight new Huffman codes according to the distribution of the integers in the blocks.

After creating the new Huffman codes, the `bred3` algorithm repeats its initial steps of matching the integer blocks to Huffman codes, and calculating the total cost of encoding the sequence of blocks with the new codes. If the algorithm observes a significant reduction in the cost of encoding the sequence of blocks, it again merges some of the

blocks, and creates a new set of Huffman codes according to the integers of the blocks. These steps continue to repeat until the reduction in the cost of encoding the sequence of blocks becomes insignificant. The `bred3` algorithm concludes by source coding the blocks of integers using the final set of Huffman codes.

The Shannon encoder of Fenwick This summary was adapted from reference [98]. Fenwick [93] proposed a BWT–based source code that uses an arithmetic encoder and a multiple–context source model. The source encoder processes the recency–rank encoder output sequence before encoding it with the arithmetic encoder. The recency–rank encoder output sequence is processed by replacing each rank with a certain codeword of a prefix code. The codewords of the prefix code are provided in table 3.6.

Each codeword of the prefix code has a binary prefix [93]. Some codewords also have a decimal suffix, which is represented by the bracketed number in certain rows of table 3.6. The structure of each prefix may be interpreted in terms of a test subject that predicts a certain rank in the recency–rank encoder output sequence. The prefix is equivalent to a sequence of answers in response to the predictions of the test subject — a zero–valued bit indicates a correct prediction, and a nonzero–valued bit an incorrect prediction.

As the zero–valued rank is most likely, the test subject initially predicts that the rank equals zero. If this is not the case, the test subject predicts that the rank equals one, as this is the second–most likely rank. The sequence of predictions continues in the order of decreasing rank likelihood. The prediction of the rank resembles an experiment that Shannon [111] performed to determine the entropy of English text. This resemblance motivated Fenwick's choice of name for the algorithm.

Upon replacing each rank with its codeword, the binary prefix and the decimal suffix of each codeword are encoded separately from each other. The bits of the binary prefixes are encoded using an arithmetic code and a multiple–context source model. The nth bits of all the prefixes are always encoded in the same context, with the exception of the first bit of each prefix. The first bit of each prefix is encoded in one of two contexts — the choice of context depends on whether the previously encoded rank equals zero or not. The decimal suffixes of the codewords are encoded using an arithmetic code with a single–context model.

Arithmetic codes and the structured source model The structured source model was introduced in section 3.3.1.4 on page 57. This section is a summary of several implementations that use this model.

Fenwick [16] proposed a BWT–based source code implementation that uses arithmetic codes, as well as the structured source model of figure 3.7 on page 57. The structured model has nine levels, and each consecutive level contains twice the number of ranks of the preceding level, with the exception of the first two levels (which contain one rank each). Fenwick's implementation of the BWT–based source code encodes both the level numbers and the ranks in zero–order contexts.

Fenwick [16] assumed that the source uses the 8–bit ASCII alphabet, which contains 256 symbols. This assumption implies that the recency–rank encoder may only produce a maximum of 256 distinct ranks. Fenwick assigned these ranks to the levels of the

Table 3.6: The prefix code that is applied to the recency–rank encoder output sequence by the Shannon encoder [93].

Rank	Codeword
0	0
1	10
2	110
3	1110
4	1111(4)
5	1111(5)
⋮	⋮
255	1111(255)

Table 3.7: Fenwick's assignment of ranks to the levels of the structured source model [16].

Level number	Ranks	Number of ranks
0	0	1
1	1	1
2	2–3	2
3	4–7	4
4	8–15	8
5	16–31	16
6	32–63	32
7	64–127	64
8	128–255	128

structured source model as indicated in table 3.7.

A level count is defined as the total number of ranks that appear in the recency–rank encoder output, but which also belong to a specific level of the structured source model. Fenwick [16] assigned ranks to the levels of the structured source model in order to reduce the difference between the level counts, as well as the difference between the frequency counts of the ranks within each level [98]. This assignment of ranks to levels facilitates the accurate estimation of each rank's probability of occurrence. Wirth [48] stated that Fenwick's assignment of ranks to levels is optimal if the ranks follow Zipf's law. Zipf's law [132] implies that a rank's probability of occurrence is inversely proportional to the magnitude of the rank.

Fenwick [93] used the implementation of the BWT–based source code and the structured source model of table 3.7 to source code the files of the Calgary corpus [100]. The implementation was found to be less effective than the implementation that uses Fenwick's Shannon encoder. The performance of the proposed implementation was improved by applying a run–length code to the recency–rank encoder output sequence prior to source coding it.

Balkenhol et. al. [95] used a structured source model in an implementation of a

BWT–based source code. The implementation postprocesses[8] the recency–rank encoder output sequence, and encodes the postprocessed sequence using arithmetic codes.

The postprocessed recency–rank encoder output consists of two sequences [95]. The implementation proposed by Balkenhol et. al. [95] encodes the two sequences separately. The source coding of the first postprocessed sequence is discussed in section 3.3.2.4 on page 89. This section concerns the source coding of the second postprocessed sequence, as well as the structured source model that is used during the source coding of this sequence.

The second postprocessed sequence contains all the ranks of the original recency–rank encoder output sequence that are higher than one [95]. Two approaches may be followed to encode the source data that these ranks represent. The first approach is to source code the ranks of the second postprocessed sequence directly, using an arithmetic code with an appropriate source model for the ranks. The second approach is to source code the symbols of the BWT output sequence that correspond to the ranks of the second postprocessed sequence.

Balkenhol et. al. [95] used the second approach to encoding the second postprocessed sequence in the implementation of a BWT–based source code. The second approach was found to be effective, as the properties of the second postprocessed sequence may be used to develop an appropriate structured source model for the relevant symbols of the BWT output sequence. This approach is summarized in what follows.

Balkenhol et. al. [95] proposed a structured source model with four levels. The implementation of the source code assigns all distinct alphabet symbols[9] to each level of the structured source model. The frequency counts of the symbols in each level are updated as source coding proceeds. The fourth level is an order (-1) context, as it maintains an equiprobable distribution over all symbols, and does not update any frequency counts.

The implementation of the BWT–based source code halves all the symbol frequency counts of a level as soon as one of its symbol frequency counts exceeds a certain count limit [95]. The halving of the frequency counts implicitly defines a sliding window over the second postprocessed sequence — the symbols corresponding to the ranks that are inside the window remain part of the frequency counts of the level. The length of the sliding window is not static, as it depends on the frequency counts of the symbols, and therefore the symbol sequence.

Balkenhol et. al. [95] specified different count limits for the first three levels of the structured source model. The rate at which the estimate of a context symbol distribution adapts to changes in the local distribution of the BWT output sequence depends on the count limit that is assigned to the level. Balkenhol et. al. proposed count limits that are linear functions of the source sequence length.

The four levels of the structured coding model of Balkenhol et. al. [95] are named according to their rates of adaptation to changes in the BWT output sequence [16]. The first three levels are referred to as the fast adaptation, medium adaptation and slow adaptation levels. The symbol frequency counts of the fourth level are not updated

[8] Refer to section 3.3.2.3 on page 76 for a summary of postprocessing.

[9] The implementation may also use a partial input alphabet, thereby eliminating unused symbols from the source model.

— the fourth level is therefore referred to as the no adaptation level.

The implementation initially considers each rank of the recency–rank encoder output sequence in a rank–by–rank fashion, starting at the front of the sequence [95]. If the rank equals zero, the implementation does not update any frequency count of the structured source model, and the source symbol that corresponds to the rank is not encoded. If the rank equals one, it increments the frequency count of the symbol that is associated with the rank in all levels of the structured source model. It does not encode the symbol, however, as it is not part of the second postprocessed sequence. If the implementation encounters a rank that is higher than one, it first encodes the symbol that corresponds to the rank. The implementation subsequently updates the frequency count of the symbol in all levels of the structured source model.

The implementation produces two outputs for each symbol of the BWT output sequence that it encodes [95]. The first output equals the level number of the context that was used to encode the symbol. The second output equals the encoded symbol.

The implementation of Balkenhol et. al. [95] first attempts to use the symbol distribution estimate that is associated with the fast adaptation level when encoding each successive symbol. If the symbol has a frequency count of zero in this context, it attempts to encode the symbol according to the distribution estimate of the medium adaptation level. In the event of the symbol having a frequency count of zero in this context, the implementation attempts to use the symbol distribution estimate of the slow adaptation level. If the symbol has a zero frequency count in all contexts of the first three levels, it is encoded according to the uniform distribution of the no adaptation level.

An arithmetic code is used to encode the level numbers [95]. Frequency counts are maintained for the level numbers — these frequency counts are used to define a distribution according to which the level numbers are encoded. A threshold is associated with the level–number counts. Level–number counts are halved as soon as any level–number count exceeds the threshold.

By disregarding all ranks that equal zero during the update of the structured source model, the implementation avoids an unfavourable situation in which the source model is excessively biased by a run of the same symbol [95]. Balkenhol et. al. refer to this situation as the ‘pressure of runs’. The structured source model of Balkenhol et. al. simultaneously satisfies the conflicting requirements of rapid adaptation to changing source statistics, and the maintenance of an accurate source model with a wide range of symbol probabilities.

Arithmetic codes and the cascaded source model The cascaded source model was introduced in section 3.3.1.4 on page 56. Fenwick [16, 88] developed a BWT–based source code implementation that uses an arithmetic code with a cascaded source model. The implementation encodes the ranks of the recency–rank encoder output sequence using the arithmetic code. The cascaded source model has three levels — the first level contains ranks 0 to 3, the second level contains ranks 4 to 15, and the final level contains all the remaining ranks. This implementation proved to be as effective as a similar implementation of Fenwick [16] that uses a structured source model.

Balkenhol et. al. [94, 133] used a cascaded source model in an implementation of a

Table 3.8: The assignment of ranks to the levels of the cascaded source model of Balkenhol et. al. [94] (adapted from reference [98]).

Level number	Ranks	Number of ranks
0	0, 1, >1	3
1	2, >2	2
2	3, 4, >4	3
3	5–8, >8	5
4	9–16, >16	9
5	17–32, >32	17
6	33–64, >64	33
7	65–128, >128	65
8	129–255	127

BWT–based source code. The implementation encodes the ranks of the recency–rank encoder output sequence using an arithmetic code. Balkenhol et. al. assigned ranks to the levels of the cascaded model according to table 3.8. The implementation keeps the levels of the source model from becoming cluttered with ranks that do not occur by using a partial input alphabet. Balkenhol et. al. also considered the use of contexts with order k in some levels of the cascaded source model, where $k > 0$ [16, 98]. This modification improved the performance of the implementation.

Arithmetic codes and the Krichevsky–Trofimov estimator Effros et. al. [10] used an arithmetic code with the Krichevsky–Trofimov (KT) estimator [134] in an implementation of a BWT–based source code. The implementation encodes the symbols of the BWT output sequence directly. The implementation uses the Krichevsky–Trofimov estimator to obtain an estimate of the probability distribution of each consecutive symbol in the BWT output sequence.

Let $r_i(x)$ denote the number of times that symbol x was encountered in the first i symbols of the BWT output sequence, and let $r_0(x) = 0$ for all x. Furthermore, let $\hat{r}_i(x) = r_i(x) + 1/2$. The KT estimate of the probability of a sequence $\mathbf{x}^i$ is recursively defined as

$$P_c(\mathbf{x}^i) = P_c(\mathbf{x}^{i-1}) \frac{\hat{r}_{i-1}(x_i)}{\sum_{x \in \mathcal{A}} \hat{r}_{i-1}(x)}, \quad 1 \leq i \leq n, \tag{3.22}$$

where $P_c(\mathbf{x}^0) \triangleq 1$, and n equals the length of the BWT output sequence.

Effros et. al. [10] investigated the performance of the proposed implementation when used to encode sequences from finite–memory sources. Two cases were considered. In the first case, the implementation has prior knowledge regarding the locations of all transition points between the p.i.i.d. segments of the BWT output sequence. In the second case, the implementation does not have any information regarding the source, and cannot calculate the locations of any transition points between the p.i.i.d. segments of the BWT output sequence. The two cases are discussed in what follows.

In the first case, the implementation applies the arithmetic code independently to each segment of p.i.i.d. symbols in the BWT output sequence [10]. Effros et. al. [10]

proved that this source code is strongly minimax universal for finite–memory sources with finite state spaces. The normalized redundancy of the code converges to zero (w.r.t. the source sequence length) at a rate that is equal to a constant multiple of the optimal convergence rate, which was derived by Rissanen [135]. The magnitude of the constant depends on the size of the symbol alphabet, and converges to unity as the size of the alphabet increases.

In the second case, the implementation source codes the BWT output sequence without any knowledge of the transition points [10]. The implementation encodes the sequence by dividing it into intervals of equal length, and applying the arithmetic code independently to each interval. The implementation has to select an appropriate interval length $w(n)$ in this case, as the performance of the source code depends on it.

The choice of an appropriate interval length depends on the length of the BWT output sequence, as the number of transition points between the p.i.i.d. segments of the BWT output sequence is finite and independent of the sequence length n [10]. Effros. et. al. [10] proposed an equation that may be used to select an appropriate interval length. If the implementation uses the equation to select the interval length, the overall source code is a strong minimax universal source code for finite–memory sources with finite state spaces.

Integer codes An integer code[10] associates variable–length binary codewords with the integers of the infinite alphabet $\mathbb{N} = \{1, 2, 3, \ldots\}$ [16]. A sequence of integers is encoded with an integer code by substituting each integer with its codeword. Each codeword of typical integer codes is self delimiting and uniquely decodable, regardless of the magnitude of the integer that the codeword represents.

Conventional source codes, such as the Huffman codes, are designed to minimize the average length of the codewords that are used to represent source sequences [16]. The design of an optimal conventional code requires knowledge of the distribution of the source symbols. If the symbol distribution is unknown, a universal source code or an adaptive source code may be used. An adaptive source code assigns codewords to symbols according to an estimate of the symbol distribution. The codewords may be changed as the estimate of the distribution changes.

The codewords of an integer code are defined independently from the distribution of the integers, and cannot be adapted to a changing integer distribution [16]. Each integer code may be used to optimally[11] encode integers from a specific distribution. This distribution is a function of the lengths of the codewords, and is referred to as the distribution that the integer code implies.

The similarity between the distribution of the integers that are to be encoded and the distribution that the integer code implies determines the effectiveness of the source code [16]. Suppose that a specific integer code is used to encode integers from a certain distribution. The average redundancy of the codewords that are produced is proportional to the Kullback–Leibler divergence between the actual integer distribution and the implied distribution [11]. An integer code would therefore prove effective if used to encode integers from a distribution that is similar to the implied distribution.

[10] These codes are also known as universal codes or variable–length codes in the literature [16].
[11] Optimality refers to the minimum average codeword length in this case.

Table 3.9: Certain codewords of the α, β and γ codes proposed by Elias [136] (adapted from reference [16]).

Integer	Elias–α	Elias–β	Elias–γ
1	1	1	1
2	01	10	010
3	001	11	011
4	0001	100	00100
5	00001	101	00101
6	000001	110	00110
7	0000001	111	00111
8	00000001	1000	0001000

An appropriate integer code for the recency–rank encoder output sequence would imply a distribution that decreases with respect to increasing integers, as the ranks typically follow a decreasing distribution. A variety of different integer codes were proposed [16, 44], but only some integer codes are appropriate for use in BWT–based source codes. The Elias–γ integer code [136] was used to effectively encode the recency–rank encoder output sequence [118], and is discussed in what follows.

Dorrigiv et. al. [118] used the Elias–γ code [136] in an implementation of a BWT–based source code. Each codeword of the Elias–γ code is a concatenation of two codewords from different codes [16]. The prefix of each codeword of the Elias–γ code is a codeword of the Elias–α integer code, but with its final bit omitted. The suffix of each codeword of the Elias–γ code is a codeword of an Elias–β integer code. Some of the codewords of the Elias codes are provided in table 3.9.

The Elias–β codeword of an integer equals the conventional binary–coded representation of the integer [16, 136]. The most significant bit of the binary–coded integer is the leading bit of the codeword. The leading zeros of each codeword are omitted. The Elias–β codeword represents the value of the integer that is encoded using the Elias–γ code.

The codewords of an Elias–β code are not self–delimiting [16,136]. This property distinguishes it from typical integer codes, which have codewords that are self–delimiting. The suffix of each Elias–γ codeword is delimited by an Elias–α codeword. The Elias–α codeword represents the bit length of the suffix of the Elias–γ codeword. The Elias–α codeword of the integer i has $i-1$ zero–valued bits and a final nonzero–valued bit. The final nonzero–valued bit of the Elias–α codeword is omitted when used as part of an Elias–γ codeword, as it is redundant.

The implementation of Dorrigiv et. al. [118] uses the Elias–γ code [136] to encode the output of the recency–rank encoder. The recency–rank encoder output sequence is encoded in a rank–by–rank fashion. The implementation represents a rank of i with the Elias–γ codeword of the integer $i+1$, as the Elias–γ code has no codeword for the integer zero.

Dorrigiv et. al. [118] proposed an improvement to the source encoder that makes use of the Elias–γ code. This improvement is beneficial to the effectiveness of the source

encoder when applied to sequences containing long runs of the rank zero. The source encoder encodes all ranks that are higher than zero as before. Upon encountering a run of j zero–valued ranks in the recency–rank encoder output sequence (where $j \geq 1$), the source encoder first produces the Elias–γ codeword of the integer one. It subsequently produces the codeword of the integer $j + 1$, thereby encoding the length of the run. This codeword is produced even if the run contains only a single zero–valued rank. The source encoder proceeds by encoding the next nonzero–valued rank of the sequence.

Arithmetic codes and frequency increments A significant problem associated with using an adaptive arithmetic code to encode either the BWT output sequence or the recency–rank encoder output sequence is to accurately estimate the probability distribution of the future symbols or ranks in the sequence [16]. Source models that are used in implementations of the PPM algorithm typically fail to produce accurate estimates of the true symbol or rank distribution.

The reason behind the inaccuracy of the PPM source models, when used to model the BWT output sequence or recency–rank encoder output sequence, lies in the statistical properties of the symbols and ranks of these sequences [16]. In the case of the BWT output sequence, the probability distribution of the symbols changes from segment to segment, and the segment transition points are typically unknown. The recency–rank encoder output sequence contains short bursts of high–valued ranks between long segments of low–valued ranks. The PPM source models do not accurately represent these sequences — another approach is required to obtain accurate estimates of the distribution of the symbols or ranks in these sequences.

Balkenhol et. al. [95] modified the mechanism for updating the symbol frequency counts in the source model of an arithmetic code implementation in an attempt to produce more accurate estimates of the symbol distribution of the BWT output sequence. The problem of appropriately updating the symbol frequency counts is referred to as the generalized frequency update problem. Balkenhol et. al. modified the implementation according to several observations regarding the BWT output sequence. It was observed that the degree of statistical similarity between two symbols in the BWT output sequence typically depends on the distance between them (i.e. the number of symbols that appear between them). Symbols that are located close to each other are more likely to have similar distributions than symbols that are distant from each other.

The previous observation regarding the statistical similarity between nearby symbols in the BWT output sequence may be used to estimate the symbol distribution at certain stages of the sequence with greater accuracy [95]. The observation implies that the estimate of a certain alphabet symbol's probability of occurrence should decrease as the distance to its previous occurrence in the sequence increases. The recency–rank encoder does incorporate this approach, as it assigns lower–valued ranks to identical symbols that occur in the vicinity of one another. An accurate probability estimate of an alphabet symbol also reflects the number of times the symbol appears in the BWT output sequence, however.

Two approaches to updating the symbol frequency counts in the source models of arithmetic encoders are relevant to this summary [95]. The first approach is to increment a symbol frequency count by a constant upon encountering the symbol in

the sequence. The symbol frequency counts are halved as soon as a threshold or a limit is exceeded by any symbol count. This technique implicitly defines a sliding window over the sequence, with symbols inside the window remaining part of the symbol frequency counts. The halving of the symbol frequency counts enables the arithmetic code to adapt to changes in the distribution of the symbols. The first approach is typically followed in the implementation of PPM algorithms.

The second approach updates the symbol frequency counts according to a certain equation [95]. Suppose that the frequency count for a certain alphabet symbol y, after having encoded the first k symbols of the BWT output sequence, is expressed as $f(y|\mathbf{x}^k)$. The symbol frequency counts may be updated according to the equation

$$f(y|\mathbf{x}^k) = \gamma^{-1} f(y|\mathbf{x}^{k-1}) + \delta(y, x_k), \tag{3.23}$$

where $\gamma > 1$ and

$$\delta(p, q) = \begin{cases} 1 & \text{if } p = q, \\ 0 & \text{otherwise.} \end{cases} \tag{3.24}$$

By using equation 3.23, the frequency counts of all but the most recently encoded symbol are decreased during each update. Balkenhol et. al. [95] stated that the symbol frequency counts need not be updated after encoding each symbol in the sequence. The increment may also be set to zero if the encoder is source coding a run of symbols, thereby preventing the model from becoming overly biased towards the run symbol.

Run–length codes As the recency–rank encoder output sequence typically contains many runs of the rank zero, it is reasonable to encode this sequence using a run–length code [16]. A run–length coded sequence is not only an effective representation of a sequence with many runs, but it is also beneficial to the source coding of the ranks that do not occur in runs.

An adaptive arithmetic code increments and rescales the rank frequency counts of the source model according to the ranks that were recently encountered in the sequence [16]. The adaptive code's estimate of the rank distribution is therefore likely to be biased at the end of a long run of a certain rank. If this is the case, the symbols that follow the run may be encoded using an inaccurate source model, thereby reducing the effectiveness of the overall source code [95]. By run–length coding the sequence prior to source coding it with an adaptive arithmetic code, the arithmetic code is made aware of the runs, and may avoid biasing the source model [86].

A general run–length encoder processes the recency–rank encoder output sequence in a rank–by–rank fashion, starting at the front of the rank sequence [86]. It determines whether the current rank of the sequence is part of a run. If the rank is part of a run, it determines the length of the run. A run of ranks is replaced only if the length of the run equals or exceeds a threshold t_h. All ranks that do not occur in a run, or that occur in a run with fewer than t_h ranks are left unchanged.

A run that contains m occurrences of a certain rank y, where $m \geq t_h$, is replaced by a sequence containing [86]

1. an optional escape symbol `SOR` (start of run),

2. one or more instances of the rank y, and

3. a codeword that represents the length of the run.

After replacing a run with the appropriate sequence, the run–length encoder proceeds to the first rank that follows the run in the recency–rank encoder output sequence. Run–length codes differ from one another regarding [86]

- the mechanism that is used to signal the start of a run,
- the manner in which the length of the run is encoded, and
- the value of the threshold t_h.

The start of a run may be signaled by inserting an escape symbol `SOR` in the run–length encoded sequence. It is typically placed in front of each encoded run. The escape symbol should not appear elsewhere in the sequence, as the source decoder would not be able to distinguish between the two instances of the symbol. An alternative approach to signaling the start of a run is to leave the initial ranks of the run uncoded. The number of ranks that are left uncoded should equal or exceed the threshold t_h in order to be recognized as the start of a run.

The effectiveness of a run–length code depends on the codewords that represent the lengths of the runs [86]. An optimal set of codewords may be selected if the distribution of the run lengths is known, but this information is typically not available. The run–length decoder must be able to delimit the codewords of the run lengths in order to successfully decode them.

Wheeler[12] proposed a run–length code that is appropriate for use in implementations of the BWT–based source code [16, 93]. This run–length code became known as the zero run–length (RLE–0) code in the literature [98]. Fenwick [16] reported an improvement of 1% in the effectiveness of a BWT–based source code implementation that first encodes the recency–rank encoder output sequence with the zero run–length code.

The RLE–0 code increases the number of ranks in the rank alphabet by one [16]. The RLE–0 encoder does not replace any run consisting of ranks that are higher than zero by its run length, regardless of the length of the run. Upon encountering a rank m that is higher than one in the recency–rank encoder output sequence, the run–length encoder replaces it with the rank $m + 1$. If the zero–valued rank is encountered in the sequence, the encoder replaces it (and all zero–valued ranks that occur in the same run) with a codeword that represents the length of the run, regardless of the length of the run.

The codeword associated with the length of a run of zero–valued ranks consists of the integers zero and one [16]. The codeword of the run length j is the conventional binary–coded representation of the integer $j + 1$, but with the least significant bit leading the codeword, and the most significant bit omitted. The omission of the most significant bit of the codeword does not influence the decodability of the run–length code, as the rank following the run is always encoded with an integer that is larger than one. The zero run–length codewords that are associated with runs of various lengths are provided in table 3.10 on page 89.

[12]It appears that Wheeler didn't publish his work regarding the run–length code. Fenwick presented the run–length code in reference [93].

Table 3.10: Certain codewords of Wheeler's zero run–length code for the recency–rank encoder output sequence.

Run length	Codeword
1	0
2	1
3	00
4	10
5	01
6	11
7	000
8	100

Abel [86] proposed two new run–length codes that are related to codes proposed by Maniscalco [137, 138]. Maniscalco proposed a set of codewords for the lengths of symbol runs. The codeword of each run length is divided into two parts. The first part is referred to as the exponent of the codeword, and is proportional to the logarithm of the run length. The exponent delimits the second part of the codeword, which is a binary–coded representation of the run length.

Source coding of the postprocessed recency–rank encoder output Balkenhol et. al. [95] implemented a BWT–based source code that postprocesses the recency–rank encoder output sequence prior to source coding it. The postprocessed recency–rank encoder output consists of two sequences — refer to section 3.3.2.3 on page 76 for details regarding the postprocessing. The second postprocessed sequence contains all the ranks of the recency–rank encoder output sequence that are higher than one. The source coding of this sequence was summarized in section 3.3.2.4 on page 81. The first postprocessed sequence equals the recency–rank encoder output sequence, but with all ranks higher than two replaced by the rank two. The source encoder for the first postprocessed sequence is summarized in this section.

The first postprocessed sequence is a ternary sequence [95]. The source encoder for the ternary sequence models it as the output of a Markov model, and uses the frequency counts of the ranks to estimate the rank distribution associated with each state. The ternary sequence is encoded with a universal source code for Markov sources.

Balkenhol et. al. [95] observed that the transition points between the piecewise segments of the BWT output sequence are manifested in the recency–rank encoder output sequence as bursts of ranks greater than one. This observation motivates the use of a Markov model with an order that is greater than zero. Balkenhol et. al. implemented several instances of the same source code for the ternary sequence, but used a Markov source model with a different order in each implementation. The most effective implementation used a Markov source model of order three.

Balkenhol et. al. [95] assigned certain states of the Markov source model to sets. The states of a set share the same rank frequency counts, and share the same estimate of the rank distribution. Upon encountering a rank in any of the states of the set, the

shared frequency count of the rank is incremented. Each set is assigned a threshold — if any of the shared rank frequency counts exceeds the threshold, all of the shared frequency counts of the set are halved. The halving of frequency counts enables the source code to adapt to changes in the first postprocessed sequence.

By assigning certain states of the Markov source model to sets, the accuracy of the model may be improved [95]. The negative impact that runs of the rank zero have on the accuracy of the source model may also be eliminated by separating the all–zero state `000` from the other states of the Markov source model.

Exclusion of the recency–rank encoder Fenwick [16, 88] and Wirth [48] investigated the performance of BWT–based source code implementations that do not use the recency–rank encoder. These implementations source code the BWT output sequence directly.

The motivation behind the omission of the recency–rank encoder concerns the optimal source coding of the BWT output sequence [10]. Each rank of the recency–rank encoder output sequence is a function of the ranks that precede it in the sequence. This fact implies that each rank is a function of several symbols of the BWT output sequence. These symbols may have different probabilities of occurrence. A source encoder for the recency–rank encoder output sequence would have to resolve the intricate relationships between the ranks in order to encode the ranks optimally (i.e. with minimum redundancy). An optimal source encoder for the BWT output sequence is conceptually simpler than the optimal encoder for the recency–rank encoder output sequence, as the dependency between the symbols of the BWT output sequence is somewhat simpler to resolve.

Fenwick [16, 88] investigated an implementation that encodes the symbols of the BWT output sequence directly using an arithmetic code. The implementation uses the multiple–context source model of a PPM–based source code. The implementation was used to source code the files of the Calgary corpus [100], and proved to be less effective than similar implementations that use the recency–rank encoder. Fenwick concluded that the lack of performance was due to the multiple–context source model of the PPM–based source code, which is not an accurate representation of the BWT and its output.

Wirth [48] investigated several implementations of BWT–based source codes that omit the recency–rank encoder. An arithmetic code was used to encode the BWT output sequence in all of the implementations. Several source models were implemented, including the structured multiple–context model of Balkenhol et. al. [95].

Instead of incrementing the frequency count of a symbol by one upon encountering it in the BWT output sequence, Wirth [48] used an increment that increases exponentially as source coding proceeds. By using the exponentially increasing increment instead of the unity increment, a more accurate source model was obtained. Wirth also investigated the use of contexts with order larger than zero, and the impact that these contexts have on the performance of the implementation. The estimation of the transition points between the piecewise segments of the BWT output sequence was considered, and various segmentation algorithms were investigated.

Several of Wirth's implementations were quoted as being effective; however, the

best of these implementations is less effective than some of the implementations that use the recency–rank encoder [16, 48].

Source coding of the IFT and DCT outputs The integers of both the IFT (section 3.3.2.3 on page 73) and the DCT (section 3.3.2.3 on page 74) output sequences are defined over the alphabet $\{0, 1, 2, \ldots n\}$, where n is the length of the source sequence [98]. This alphabet is ordinarily much larger than the alphabets of the original source sequence and the recency–rank encoder output sequence. An implementation of a BWT–based source code requires accurate estimates of the output integer distribution in order to effectively encode these sequences.

Deorowicz [98] proposed two techniques for source coding the IFT and DCT output sequences. The first technique involves the division of the alphabet $\{0, 1, 2, \ldots n\}$ into several groups, so that all integers in the interval $\{2^i, 2^i + 1, \ldots 2^{i+1} - 1\}$, where $i \geq 0$, belong to the same group[13]. The source encoder maintains frequency counts for the integers that occur in each group, as well as frequency counts for the groups themselves. The frequency counts are incremented as consecutive integers of the sequence are encoded.

An integer of the IFT or DCT output sequence is encoded by first encoding the number of the group to which the integer belongs, followed by the integer itself [98]. An arithmetic code is used to encode both the group number and the integer. The arithmetic code uses the frequency counts to estimate the distributions of the groups and the integers. This technique is essentially an implementation of the structured source model, as discussed in section 3.3.1.4 on page 57.

The second technique of Deorowicz [98] replaces each integer of the transformed sequence with a codeword of a binary prefix code. Deorowicz proposed that the Elias–γ code [136] be used as the binary prefix code. The bits of each codeword are encoded independently from one another using a binary arithmetic code. This technique was quoted as being the most effective among the techniques examined by Arnavut [110], who developed the IFT.

The source codes for the IFT output sequence may also be used to encode the output sequence of the DCT [98]. Alternative source codes for the DCT output sequence have been proposed, however. These source codes are not relevant to this thesis.

Construction of explicit context trees Larsson [139] proposed a novel source code that may be used to encode the BWT output sequence directly. The source code was designed under the assumption that the source sequences are produced by a tree source (refer to section 3.1.1.3 on page 40 for a summary of tree sources). The source code requires the construction of a suffix tree of the source sequence. It uses the suffix tree to estimate the structure of the tree source, and uses this estimate as a model of the source. The BWT output sequence is subsequently encoded using an arithmetic code and the source model. The effectiveness of the source code depends on the accuracy of the source model.

Recall that the path from the root node of a suffix tree to each leaf node of the tree corresponds to a suffix of the original source sequence [139]. The source symbol

[13] The integer zero belongs to a group that contains no other integers.

that precedes each suffix may be predicted from the suffix itself. The initial symbols of the suffix are often sufficient to accurately predict the preceding symbol, however. These initial symbols correspond to the nodes of a tree source. As a path from the root node of a suffix tree to each of its internal nodes corresponds to the initial symbols of a suffix, it follows that an estimate of the tree source may be obtained by pruning the suffix tree of the source sequence.

The pruning of the suffix tree is carried out as follows [139]. Following the construction of the complete suffix tree, the encoder iterates over all the internal nodes of the tree, starting at the parent nodes of the leaves and moving towards the root. It counts the number of leaf nodes that ultimately descend from each internal node. This number equals the total number of contexts in the source sequence that share the common prefix corresponding to the path from the root to the internal node. It subsequently obtains the frequency counts of all alphabet symbols that the BWT produces in its traversal of the subtree rooted at the internal node. These frequency counts imply a distribution for the symbols that precede all the contexts that share the common prefix. The algorithm subsequently compares this distribution with the distribution that corresponds to its parent node. If it detects no significant deviation between the distribution and that of its parent, the pruning algorithm merges the node with its parent.

The pruned tree is encoded as part of the source–coded sequence, as the source decoder requires it to decode the sequence [139]. The number of bits that are required to represent the tree is reduced by pruning it. The overly aggressive pruning of the suffix tree may remove certain contexts from the model that correspond to valid contexts of the source. The removal of these contexts may significantly reduce the effectiveness of the source code that uses the model. A tradeoff exists between the shorter representation of the model in the source–coded sequence, and the accuracy of the model.

Prior to encoding the pruned tree, Larsson [139] proposed that the tree undergo another round of pruning. The algorithm that performs the first round of pruning does not take the number of bits that are required to represent the encoded tree into consideration. Some nodes of the pruned tree typically have very little impact on the effectiveness of the source code, but require a significant number of bits to represent. The second round of pruning removes these nodes from the pruned tree. The final tree is source coded by traversing the nodes of the tree, and encoding the number of child nodes that belong to each node. The counts are represented as exponent–mantissa pairs. The exponents are encoded using a first–order arithmetic encoder.

Larsson proposed two alternative approaches to source coding the BWT output sequence [139]. In the first approach, the source encoder records the frequency counts of the symbols that are produced in each context of the source model prior to encoding the symbols. These frequency counts are used to estimate the true distribution of the symbols that occur in each context. An arithmetic code is subsequently used to encode the BWT output sequence according to the distribution of each context of the source model.

In the second approach to source coding the BWT output sequence, the source encoder accumulates frequency counts for the symbols that occur in each context of the source model as each consecutive symbol is encoded [139]. Upon encountering a

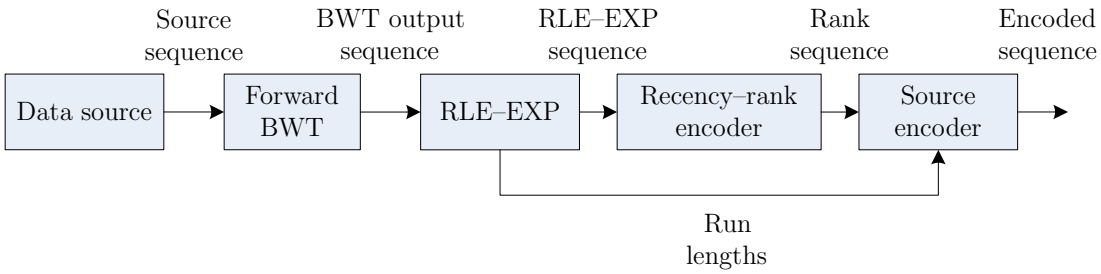


Figure 3.8: The block diagram of the RLE–EXP source code [86].

new symbol in a context, the source encoder uses an escape symbol mechanism to switch to a shorter context.

Larsson [139] claimed that the proposed source code is computationally less complex than the PPM algorithm; however, the proposed code was demonstrated as being ineffective when used to encode short to moderately long sequences. Larsson stated that the performance of the source code may be improved by encoding the pruned tree more effectively, and by improving the pruning algorithm.

3.3.3 Alternative BWT–based source codes

Some BWT–based source codes do not have the structure of the elementary BWT–based source code. These alternative BWT–based source codes have block diagrams that differ from figure 3.5 on page 53. Several alternative BWT–based source codes are discussed in what follows.

3.3.3.1 The RLE–EXP source code

Abel [86] proposed an alternative BWT–based source code that is known as the RLE–EXP source code. The block diagram of this source code is presented in figure 3.8. The significant difference between the RLE–EXP source code and the elementary BWT–based source code is the insertion of a run–length encoder immediately after the forward BWT. The run–length code associated with this encoder is known as the RLE–EXP run–length code.

Abel's motivation for inserting the run–length encoder between the forward BWT and the recency–rank encoder is twofold [86].

1. The recency–rank encoder need not encode as many symbols as would be the case without the run–length encoder, as the run–length encoder removes several run symbols from the BWT output sequence. As the run–length encoder is more efficient than the recency–rank encoder, the overall source code is more efficient.

2. By removing the symbol runs from the BWT output sequence, the source encoder does not increment its frequency counts of the run symbols. The source model is therefore not unnecessarily biased in favour of the run symbol.

The RLE–EXP run–length code does not use any escape symbols to signal the start of a symbol run, as escape symbols disrupt the context structure of the BWT output

sequence [86]. This disruption causes the recency–rank encoder to produce more high–valued ranks, thereby reducing the effectiveness of the source code. The RLE–EXP run–length code produces a threshold run[14] to signal the start of a run.

The codeword of each run length is divided into two parts [86]. The first part of each codeword is referred to as the exponent. The exponent contains $\lceil \log_2(l-1) \rceil$ repetitions of the run symbol, where l is the length of the run. The exponent delimits the second part of the codeword by indicating its length. The second part of the codeword is the conventional binary–coded representation of the run length l.

Abel [86] stated that the binary–coded representation of the run lengths disrupts the context structure of the BWT output sequence. The binary–coded run lengths are therefore removed from the output of the run–length encoder, and forwarded to the source encoder without being encoded by the recency–rank encoder. The sequence of binary–coded run lengths is referred to as the RLE mantissa buffer, or RMB.

Abel [86] investigated several alternatives to the recency–rank encoder, and used some of these alternatives in an RLE–EXP source code implementation. The RLE–EXP source code implementation was found to be most effective if the recency–rank encoder was replaced by the inversion frequencies transform, or IFT (refer to section 3.3.2.3 on page 73 for a summary of the IFT).

Abel [86] proposed a novel source encoder for the RLE–EXP source code implementation. Each integer of the IFT output sequence is replaced with a codeword. The codeword of a certain integer k is divided into two parts. The first part of the codeword is referred to as the exponential part, and consists of an integer that equals $\lceil \log_2(k) \rceil$. The first part of the codeword delimits the second part, which is the conventional binary–coded representation of the integer k. After replacing all integers of the IFT output sequence with their codewords, the codewords are encoded using an arithmetic code.

Suppose that the codeword of the integer k is to be encoded using the arithmetic code [86]. The exponential part of the codeword is encoded using a structured source model with two levels (refer to section 3.3.1.4 on page 57 for a summary of the structured source model). The first level of the structured source model contains the integers zero to four, and the second level contains all integers larger than four.

The binary–coded integers (i.e. the second part of each codeword) are sorted w.r.t. their lengths prior to being encoded using an arithmetic code [86]. The distribution that the arithmetic code selects to encode the conventional binary–coded representation of the integer k depends on the length of the binary number. All binary numbers with a single bit are encoded using the distribution m_1, which is defined over two distinct elements (`0` and `1`). All binary numbers with two bits are encoded using the distribution m_2, which is defined over four elements (`00`, `01`, `10` and `11`). The distribution that is used to encode binary numbers with three bits, m_3, is defined over eight distinct elements.

All binary numbers with four or more bits are divided into two parts prior to being source coded [86]. The first part constitutes the initial three bits of the binary number, and is encoded using the distribution m_3. The second part contains the remaining bits of the binary number — these bits are encoded sequentially using a separate distribution m_0, which is defined over the elements `0` and `1`. The distribution m_3 is

[14] A threshold run has a length equal to the threshold of the run–length code.

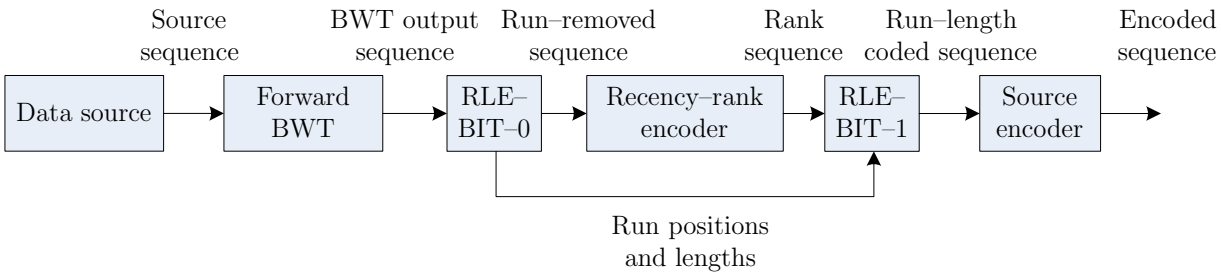


Figure 3.9: The block diagram of the RLE–BIT source code [86].

used to encode the first three bits of each binary number as these bits have a more stationary distribution than the remaining bits.

The source encoder of the RLE–EXP source code also encodes the sequence of run lengths produced by the run–length encoder [86]. Each binary–coded run length is encoded in exactly the same manner as the binary–coded integers of the IFT output sequence.

3.3.3.2 The RLE–BIT source code

Abel [86] proposed a second alternative to the elementary BWT–based source code. The block diagram of this source code, which is referred to as the RLE–BIT source code, is presented in figure 3.9.

The RLE–BIT source code uses a run–length code that resembles the RLE–0 code proposed by Wheeler — refer to section 3.3.2.4 on page 88 for a summary of the RLE–0 code [16, 86, 93]. This run–length code uses an escape symbol to signal the start of a run. The run–length encoder is divided into two blocks. The first block, which is referred to as the RLE–BIT–0 block, is inserted immediately before the recency–rank encoder. The second block, which is referred to as the RLE–BIT–1 block, is inserted immediately after the recency–rank encoder.

The RLE–BIT–0 block of the run–length encoder removes all symbols that occur in each run of the BWT output sequence, except for the first symbol of each run [86]. It also records the start position of each run in the BWT output sequence, as well as the length of each run. The run start positions and run lengths are forwarded to the RLE–BIT–1 block, without being encoded by the recency–rank encoder.

The RLE–BIT–1 block of the run–length encoder increments each rank of the recency–rank encoder output sequence by two [86]. After incrementing each rank by two, the integers zero and one no longer appear in the recency–rank encoder output sequence. These integers may be used to encode the length of each run. The RLE–BIT–1 block replaces each run length with its conventional binary–coded representation, and removes the most significant bit from this codeword. Each codeword is inserted into the appropriate position of the recency–rank encoder output sequence. The start of the run is therefore signaled by the integers zero and one.

Abel [86] proposed a novel source encoder for the RLE–BIT source code. The source model that is used to encode the RLE–BIT–1 output sequence is divided into two parts. The first part of the model is associated with the source coding of the integers

zero, one and two, which are the most frequently occurring integers of the sequence. These integers may be interpreted as a ternary sequence that is similar to the first postprocessed recency–rank encoder output sequence in a BWT–based source code implementation of Balkenhol et. al. [95]. This sequence may be encoded using the same source encoder as Balkenhol et. al. (refer to section 3.3.2.3 on page 76 for a summary of postprocessing, and to section 3.3.2.4 on page 89 for a summary of the source encoder).

The second part of the model is associated with the source coding of all integers larger than two. Abel [86] proposed a structured source model for this purpose. The source model has eight levels, with the integers $\{3, 4, \ldots 257\}$ divided between the levels.

Abel [86] stated that the RLE–BIT source code may not use the IFT as an alternative to the recency–rank encoder, as the IFT does not preserve the length of the BWT output sequence. The WFC transform may be used as an alternative to the recency–rank encoder, however.

3.3.3.3 The hybrid RLE–EXP / RLE–BIT source code

Abel [86] proposed a third BWT–based source code that may be interpreted as a combination of the RLE–EXP and RLE–BIT source codes. The third source code was proposed due to several observations regarding the performance of the RLE–EXP and RLE–BIT source codes.

Abel [86] found that an implementation of the RLE–EXP source code that used the IFT instead of the recency–rank encoder encoded the larger files of the Calgary corpus [100] effectively. The same implementation was not as effective when used to source code shorter files, due to the overhead of the source code. The RLE–BIT source code proved to be more effective when used to source code shorter files.

The third BWT–based source code of Abel [86] uses the RLE–EXP source code to encode files with sizes exceeding 256 kB, and the RLE–BIT source code to encode the shorter files. The RLE–EXP source code implementation uses the IFT as an alternative to the recency–rank encoder, and the RLE–BIT source code implementation uses the AWFC transform. This hybrid RLE–EXP / RLE–BIT source code proved to be very effective when used to source code the files of the Calgary corpus [100].

3.3.3.4 Online source codes

Conventional BWT–based source codes have two drawbacks. The first drawback is the offline nature of these source codes — the source encoder requires access to the entire source sequence prior to transforming and encoding it. The source decoder also requires access to the entire source–coded sequence prior to decoding it. The second drawback concerns the fact that a conventional source encoder is unable to directly infer the context structure of the source from the BWT output sequence.

Yokoo [140] proposed a novel BWT–based source code that addresses the two drawbacks of conventional BWT–based source codes. The encoder of this source code applies the BWT in an online manner, and encodes the source sequence (instead of the BWT output sequence) directly. The BWT is only used to obtain information regarding the context structure of the source sequence.

The BWT–based source code proposed by Yokoo [48,140] is used to encode a source sequence in a symbol–by–symbol fashion, starting at the front of the source sequence. Suppose that the ith symbol of the source sequence is to be encoded. The source encoder initially applies the forward BWT to the first $i-1$ symbols of the source sequence. It proceeds by recording the prefix[15] that corresponds to the preceding context of the ith symbol of the source sequence.

The source encoder locates the recorded prefix in the row–sorted BWT table, and compares the recorded prefix to those prefixes in neighbouring rows of the table [48,140]. The similarity of two prefixes may be measured by the length of the common suffix that they share. The source encoder ranks the symbols that follow the prefixes of the neighbouring rows according to the degree to which their prefixes are similar to the recorded prefix. Those symbols that follow prefixes similar to the recorded prefix are assigned a low rank, while the symbols that follow prefixes that are dissimilar to the recorded prefix are assigned high ranks. The source encoder obtains a rank for each symbol of the alphabet in this manner. The ith symbol of the BWT output sequence is subsequently replaced with its rank.

The final step of the source encoder is to encode each rank [48]. Yokoo [140] proposed several static source codes for this purpose. These codes are simple and have low computational complexity, but are not as effective as some arithmetic codes.

The source code proposed by Yokoo [48,140] uses information regarding the contexts of the source sequence to assign lower ranks to symbols that occur in similar contexts to the current context. The rank distribution is therefore biased towards the lower–valued ranks, which implies that the ranks may be encoded more effectively.

3.3.3.5 Inversion coding

Arnavut [141] proposed a novel BWT–based source code. The source code uses a novel alternative to the recency–rank encoder that is known as the inversion coder. The design of the inversion coder requires knowledge of discrete mathematics and permutations — it is not within the scope of this thesis to examine these subjects in detail. The relevant theory that is required to understand the function of the encoder is introduced.

The inversion coder interprets the BWT output sequence as a multiset permutation [141]. A multiset is essentially an ordinary set that may contain multiple instances of the same element. The inversion coder first collects the frequency counts of the alphabet symbols that are present in the BWT output sequence, and uses these counts to construct the identity permutation of the multiset permutation. The identity permutation contains all the symbols of the BWT output sequence, but in nondescending lexicographical order.

The inversion coder subsequently calculates the canonical sorting permutation that is associated with the multiset permutation [141]. The ith element of the canonical sorting permutation is the index, in the original multiset permutation, of the ith symbol of the identity permutation. The indices of identical symbols occur in ascending order relative to one another in the canonical sorting permutation. The BWT output sequence

[15]The BWT is performed in such a manner that preceding contexts, or prefixes, of the source sequence are sorted lexicographically.

may be recovered using the frequency counts of the alphabet symbols, as well as the canonical sorting permutation.

The goal of the inversion coder is to encode the canonical sorting permutation as effectively as possible [141]. The inversion coder therefore calculates an inversion vector of the canonical sorting permutation. The inversion vector is a simpler representation of the canonical sorting permutation, and may be encoded effectively with a source code that is conceptually simple. The canonical sorting permutation may be recovered from one of its inversion vectors.

Inversion vectors and their generation were investigated in the literature [142]. Arnavut [141] used the left–bigger inversion vector (LBIV) to represent the canonical sorting permutation. The ith element of the LBIV equals the number of elements in the first $i-1$ elements of the canonical sorting permutation that are larger than value i of the canonical sorting permutation.

The canonical sorting permutation may be divided into segments corresponding to distinct alphabet symbols, as the identity permutation is sorted in lexicographical order [141]. As the integer elements in each segment of the canonical sorting permutation appear in ascending order, the integers inside each segment of the LBIV appear in nonascending order.

Many consecutive integers that appear in the canonical sorting permutation of the BWT output sequence differ by one, as the BWT output sequence tends to contain runs of the same symbol [141]. This statement implies that the LBIV segments tend to contain runs of the same integer. This characteristic of the LBIV may be exploited by encoding the absolute difference between consecutive integers of the LBIV, which would ideally equal zero. The sequence of absolute difference values between consecutive integers of an LBIV segment is known as a decorrelated segment sequence.

The source encoder encodes the decorrelated segment sequences using an arithmetic code with a structured source model [141]. It also encodes the frequency counts for each alphabet symbol in the BWT output sequence, as these counts are necessary to recover the canonical sorting permutation, and therefore the BWT output sequence. The source code proved to be effective when used to encode the larger files of the Calgary [100] and Canterbury [112] corpora.

3.3.3.6 Wavelet–tree source codes

Foschini et. al. [143] introduced a novel BWT–based source code that uses a wavelet tree [144] to represent the BWT output sequence. The wavelet tree is source coded using a run–length code and the Elias–γ integer code [136].

A wavelet tree contains a root node, several internal nodes, and leaf nodes [143]. Each parent node has exactly two child nodes. Each leaf node corresponds to a distinct alphabet symbol. Each internal node, as well as the root node, is associated with a symbol sequence. The symbol sequence associated with the root node equals the BWT output sequence. Suppose that a parent node is associated with a sequence, and that the distinct symbols of the sequence constitute a set. This set may be interpreted as a subalphabet. The symbols of the parent node sequence are defined over this subalphabet, as the subalphabet is a subset of the symbol alphabet.

During the construction of the wavelet tree, each parent node divides its subalphabet into two subsets [143]. It extracts all symbols that belong to the first subset from the sequence it is associated with, in a symbol–by–symbol fashion, starting at the front of the sequence. This sequence of extracted symbols is associated with its left child node. The symbols that remain in the sequence belong to the second subset. This sequence is assigned to the right child node of the parent.

The sequence that is associated with each parent node may be represented as a sequence of binary digits [143]. Each symbol that was extracted from the sequence, and assigned to the sequence of the left child node, is assigned a zero–valued bit. Each of the remaining symbols is assigned a nonzero–valued bit. The subalphabet of each parent node should ideally be divided in such a manner that the bit sequence associated with the symbol sequence of the parent contains long runs of zero–valued and nonzero–valued bits.

The wavelet tree is encoded by the source encoder as follows [143]. The source encoder traverses the nodes of the wavelet tree in a breadth–first manner. It encodes the bit sequence associated with each node using a binary run–length code. Each bit sequence may be interpreted as a sequence of alternating runs of zero–valued bits and nonzero–valued bits — the run–length encoder therefore produces the integer length of each of the alternating runs. This sequence of run lengths is encoded using an integer code. The choice of the Elias–γ integer code was empirically motivated.

Foschini et. al. [143] implemented the BWT–based source code and compared its performance to other source code implementations by applying it to the files of the Calgary [100] and Canterbury [112] corpora. It was concluded that the implementation of the BWT–based source code is competitive with the `bzip2` and `gzip` source code implementations in terms of effectiveness. The implementation required less computation time than the `bzip2` implementation to source code the same files.

CHAPTER 4

Enumerative source codes

This chapter presents a summary of the literature that concerns enumerative source codes. An enumerative source code may be constructed by considering all sequences of a certain length that a source can produce. All these sequences are divided into groups, but in such a manner that all equiprobable sequences are assigned to the same group. Both the source encoder and decoder are aware of the groups and the sequences that belong to each group. A source sequence is encoded by producing the number of the group to which the sequence belongs, as well as its index in the group.

One drawback of enumerative source codes is the high degree of computational complexity involved in the enumeration of all source sequences of a certain length. It is not feasible to enumerate all source sequences longer than a few tens of bits, as the number of possible source sequences increases exponentially w.r.t. the source sequence length. Encoders of enumerative source codes typically calculate the codeword of each source sequence, instead of searching for each source sequence and its codeword in a list of all source sequences and codewords. The calculation remains computationally complex, however. Chapter 5 explores several techniques for reducing the computational complexity involved in the calculation of the codewords.

Enumerative source codes emerged in the 1960s [5, 6], and received some attention in the literature during the 1970s [7, 60]. In 1984, Cleary et. al. [8] proved that both exact and approximate enumerative source coding may be performed using an arithmetic source code and an appropriate adaptive model of the source. Enumerative source code implementations that use arithmetic codes are significantly more efficient than direct implementations of enumerative source codes. The specification of an appropriate source model for the exact enumerative source coding of sequences from higher–order sources using an arithmetic code is excessively complex, however. Approximate enumerative source codes were developed to address this problem. These source codes are nearly as effective as the exact enumerative source codes, and are applicable to more complex sources.

Much of the interest in enumerative source codes shifted to arithmetic source codes after the publication of the research by Cleary et. al. [8]. The literature does contain more recent publications regarding enumerative source codes, however [81, 145–152]. Both the earlier and the more recent publications are summarized in this chapter.

Two types of enumerative source codes may be distinguished. Enumerative codes of the first type are referred to as blockwise enumerative codes [5–7, 60, 145, 146, 152]. The encoder of a blockwise enumerative source code partitions a long source sequence into blocks, and independently encodes each block of the sequence. The first enumerative code proposed in the literature was a blockwise enumerative source code [5].

Enumerative source codes of the second type are referred to as symbolwise enu-

merative codes. These source codes are the exact and approximate enumerative source codes that were proposed by Cleary et. al. [8]. Symbolwise enumerative source code implementations use arithmetic source codes. A source sequence is encoded in a symbol–by–symbol manner using this type of source code. Both blockwise and symbolwise enumerative codes are discussed in this chapter.

4.1 Blockwise enumerative source codes

An encoder of a blockwise enumerative source code produces a single codeword for an entire sequence of source symbols. A long source sequence is encoded by dividing it into blocks of consecutive symbols, and separately encoding the symbol sequence of each block.

The first blockwise enumerative source code was developed to encode the indices of nonredundant data samples in a sequence of samples [5, 6]. Schalkwijk [60] encoded sequences of i.i.d. bits with an enumerative source code that is related to the first enumerative source code. The enumerative source code for sequences of i.i.d. bits was generalized to include sources that produce sequences of nonbinary i.i.d. symbols [60], as well as two–state Markov sources that produce bit sequences (i.e. first–order binary Markov sources) [7].

4.1.1 The first enumerative source code

Lynch [5] proposed the first enumerative source code. The first enumerative code was used in a source code implementation that removed redundant data samples from a sequence of samples. This implementation is summarized in what follows.

The implementation of the source code encodes a sequence of data samples [5]. It inserts time words between neighbouring blocks of n consecutive samples of the source sequence. The implementation proceeds by selecting and removing redundant samples that occur between each pair of consecutive time words.

In order to successfully recover the original source sequence, it is necessary to encode the indices of the nonredundant samples between consecutive time words of the original source sequence [5]. Each index is calculated relative to the index of the most recent time word. The implementation may equivalently encode the indices of the redundant samples in the original source sequence.

The indices of the nonredundant samples need not be explicitly encoded by the implementation [5]. It may instead consider all possible combinations of choosing the nonredundant samples from the samples between consecutive time words. A unique integer may be associated with each distinct pattern of redundant and nonredundant samples between consecutive time words. The implementation may encode this integer instead of the indices of the nonredundant samples. A practical implementation requires an efficient technique for mapping the patterns to integers, however.

Suppose that $n - m$ redundant samples are to be removed from a sequence of n samples [5]. The m nonredundant samples may be chosen from the n–sample sequence

in

$$B_{n,m} = \binom{n}{m} \tag{4.1}$$

distinct ways. Equivalently, one may consider an integer sequence of length m that is referred to as the index sequence. The elements of the index sequence equal the distinct indices of the m nonredundant samples of the n–sample sequence. It is assumed that index sequences are sorted in ascending order, and that no index sequence contains any duplicate integers. It follows that a total of $B_{n,m}$ monotonically increasing index sequences exist.

The source encoder proposed by Lynch [5] encodes the monotonically increasing index sequence, thereby producing an integer codeword that corresponds to the index sequence. This integer codeword represents a specific pattern of redundant and nonredundant samples. The index sequence is encoded by using a variation of a method that was proposed by Gordon [153]. Gordon's method uses what is referred to as a path–count matrix to represent all valid index sequences, as well as the values that each integer of the index sequence may assume [5]. The path–count matrix for sequences with n equal to seven and m equal to five is presented in figure 4.1 on page 103.

The path–count matrix is a matrix of nodes [5]. Each of the m columns of the matrix corresponds to one of the integer elements of the index sequence. The leftmost column of the matrix corresponds to the first integer of the index sequence; the second–to–left column represents the second integer of the index sequence, etc. Each of the n rows of the matrix corresponds to one of the samples that occurs between consecutive time words. The bottom row of the matrix corresponds to the first sample between consecutive time words; the second–to–bottom row represents the second sample between consecutive time words, etc. Each node of the matrix corresponds to a specific element of the index sequence being assigned a specific value. The index of the element equals the column number of the node, and the value that is assigned to it equals the row number. The columns are numbered from left to right, and the rows are numbered from bottom to top.

The index sequence may be represented as a path through the path–count matrix, as illustrated in figure 4.1 on page 103 [5]. Each path begins at the node labeled 'start' and terminates at the node labeled 'end' — these two nodes represent two consecutive time words. Each node on the path (with the exception of the 'start' and 'end' nodes) implies that a specific element of the index sequence is assigned a certain sample index. The fact that the index sequence is sorted restricts the number of valid paths through the matrix. Figure 4.1 contains all valid paths through the path–count matrix that contains seven rows and five columns.

In order to develop an efficient source code for the index sequence that corresponds to a path through the path–count matrix, the accumulated number of paths that enter each node of the matrix are counted [5]. The paths are counted by starting at the leftmost column of the path–count matrix, and proceeding in a column–by–column fashion to the rightmost column of the matrix. The number of paths that enter each node of the path–count matrix of figure 4.1 is denoted by a number above the node.

A total of 21 paths enter the node of figure 4.1 that is marked by the label 'end' [5]. The number of paths that enter this node equals the total number of valid paths through the path–count matrix, and therefore the total number of valid index sequences. As a

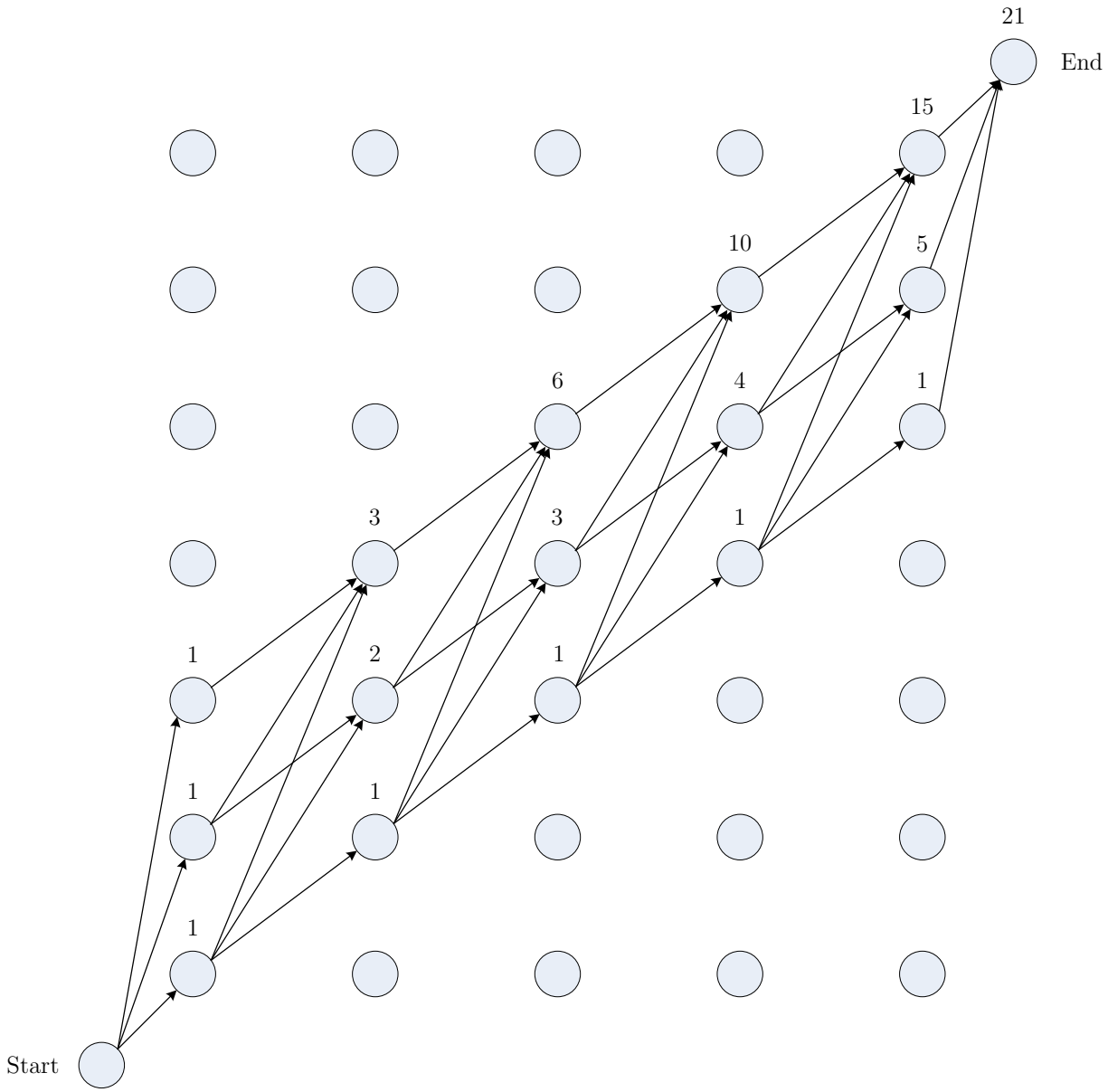


Figure 4.1: The path–count matrix for sequences of seven samples ($n = 7$). Five of the samples of each sequence are nonredundant ($m = 5$). This figure was adapted from reference [5].

total of

$$\begin{aligned} B_{7,5} &= \binom{7}{5} \\ &= 21 \end{aligned} \tag{4.2}$$

valid index sequences exist for the case where $n = 7$ and $m = 5$, the structure of the path–count matrix is verified.

The enumerative source code maps each valid path through the path–count matrix to a distinct integer codeword [5]. The integer codeword is calculated as the sum of m

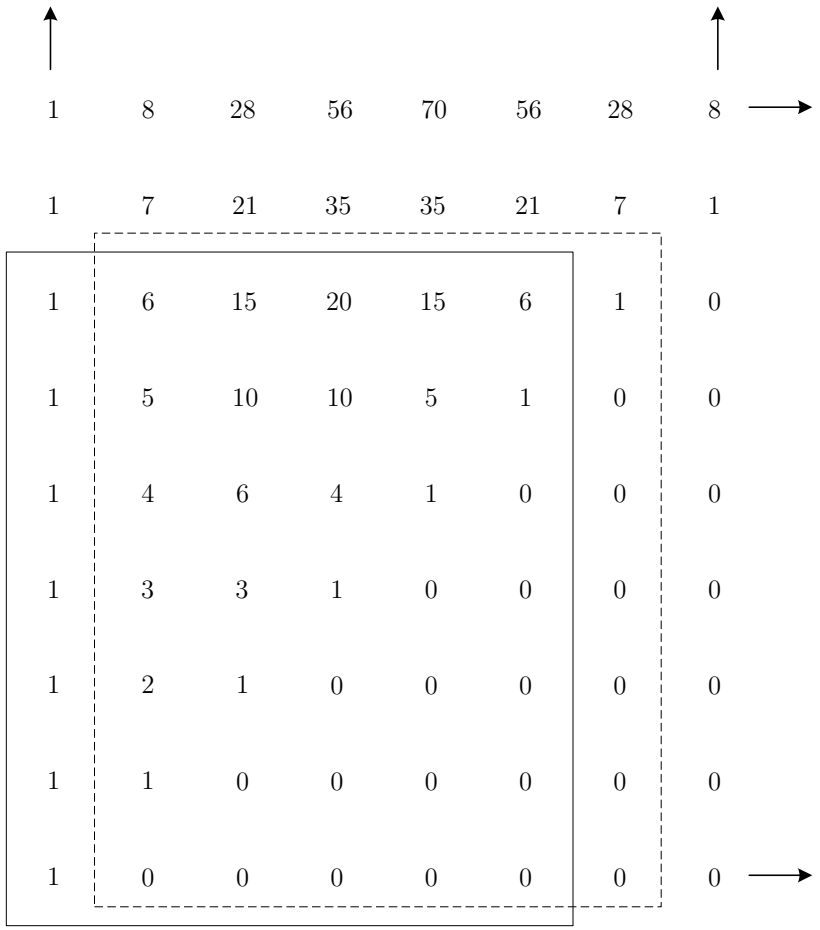


Figure 4.2: Derivation of the coding matrix from the path–count matrix (adapted from reference [5]).

integers. Each of the integers in the sum is associated with one of the nodes on the corresponding path through the path–count matrix. Lynch [5] used a coding matrix to establish the relationship between the nodes on the path through the path–count matrix and the actual integers of the sum. The coding matrix is derived from the path–count matrix as illustrated in figure 4.2 on page 104.

The coding matrix is a matrix of integers [5]. Each integer equals the path count of a specific node of the path–count matrix[1]. The path counts that correspond to the nodes of the path–count matrix are enclosed by a solid–line rectangle in figure 4.2. The coding matrix is obtained by shifting the rectangle of the path–count matrix one column to the right. The second column of the path–count matrix therefore corresponds to the first column of the coding matrix; the third column of the path–count matrix corresponds to the second column of the coding matrix, etc.

The enumerative source encoder calculates the integer codeword of a path through the path–count matrix as follows [5]. It replaces the path count of the node in row i and column j of the path–count matrix with the integer in row i and column j of the

[1]The coding matrix is derived under the assumption that n and m may be arbitrarily large.

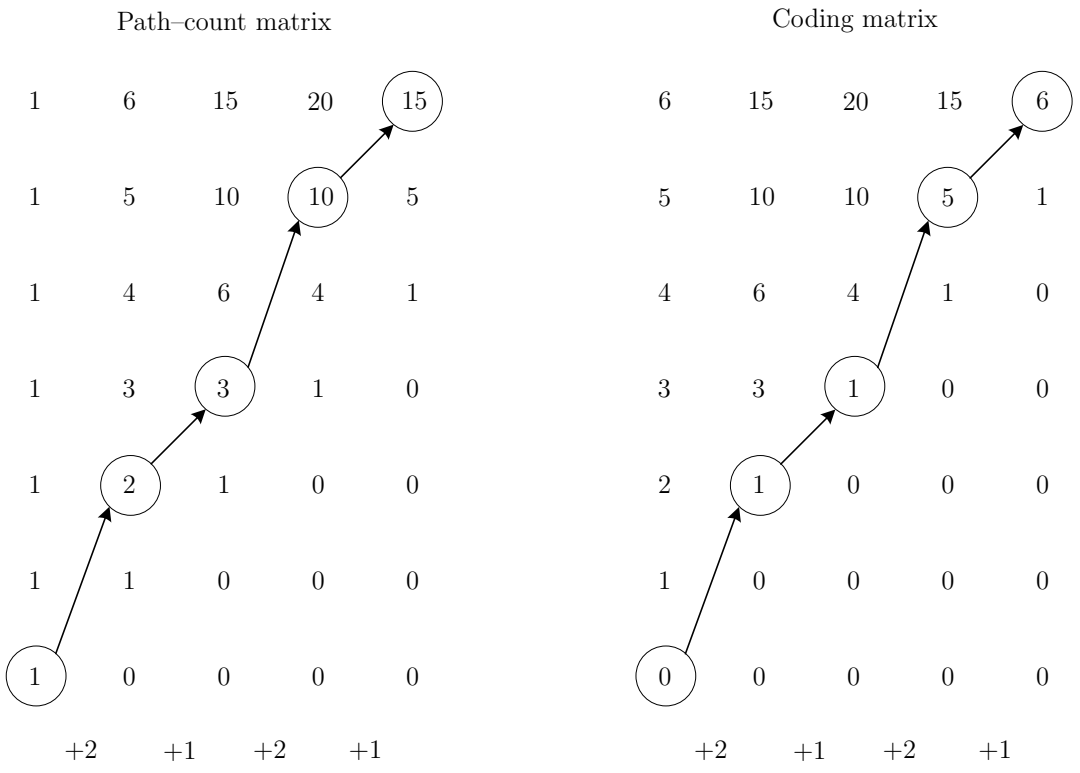


Figure 4.3: A path–count matrix (a) and coding matrix (b). Both matrices contain the path that corresponds to the index sequence $\{1, 3, 4, 6, 7\}$, where $n = 7$ and $m = 5$ (adapted from reference [5]).

coding matrix. It subsequently adds the new path counts of the nodes that lie on the path through the path–count matrix to obtain the codeword. The same codeword may be obtained by drawing the same path in the coding matrix, and adding the integers on the path.

Suppose that the codeword of the index sequence $\{1, 3, 4, 6, 7\}$ is to be calculated under the assumption that $n = 7$ and $m = 5$, and that the index of the first element of any sequence equals one. The index sequence is associated with a path through the path–count matrix as illustrated in figure 4.3(a) — note that the bottom row of the matrix is considered to be the first row of the matrix. In order to calculate the codeword, the same path is drawn in the coding matrix, as illustrated in figure 4.3(b). The codeword of the index sequence is obtained by adding the integers that lie on the path through the coding matrix. The integer codeword for the index sequence $\{1, 3, 4, 6, 7\}$ is calculated as

$$\begin{aligned} c &= 0 + 1 + 1 + 5 + 6 \\ &= 13. \end{aligned} \tag{4.3}$$

Lynch [5] did not prove that the enumerative encoder produces a distinct (and therefore uniquely decodable) integer codeword for each distinct index sequence. In this thesis it is proved that the codewords are uniquely decodable — this proof is provided in section 6.1.1.4 on page 142. Lynch did propose a source decoder that may be used

to decode each integer codeword, however. This source decoder requires knowledge of the number of columns (m) and the number of rows (n) of the coding matrix.

The source decoder first constructs the coding matrix with m columns and n rows [5]. It initializes a running total that is set equal to the integer codeword. After constructing the coding matrix, the source decoder iterates over the columns of the coding matrix, starting at the rightmost column and ending at the leftmost column. It finds the largest integer in each column that does not exceed the running total. This integer is subtracted from the running total prior to the start of the next iteration. After iterating over all the columns of the coding matrix, the source decoder marks each row that contains an integer that was subtracted from the running total. The numbers of the marked rows constitute the index sequence.

The validity of the source decoder is demonstrated by using it to decode the integer codeword 13, which is associated with the index sequence $\{1, 3, 4, 6, 7\}$. The source decoder first constructs the coding matrix with five columns and seven rows, as illustrated in figure 4.3(b) on page 105. It also assigns a value of 13 to the running total $\hat{c}_5$, which is the value of the integer codeword. Starting in the rightmost column of the coding matrix, it finds the largest integer in the column that does not exceed the running total. This integer equals six. The source decoder subtracts six from the running total $\hat{c}_5$, obtaining an updated running total that equals

$$\begin{aligned} \hat{c}_4 &= \hat{c}_5 - 6 \\ &= 7. \end{aligned} \tag{4.4}$$

The source decoder next considers the fourth column of the coding matrix. The largest integer in this column that does not exceed the running total equals five. The updated running total is obtained as

$$\begin{aligned} \hat{c}_3 &= \hat{c}_4 - 5 \\ &= 2. \end{aligned} \tag{4.5}$$

The source decoder proceeds by finding the largest integer in the third column of the coding matrix that does not exceed the running total. This integer, which equals one, is subtracted from the running total to obtain its updated value as

$$\begin{aligned} \hat{c}_2 &= \hat{c}_3 - 1 \\ &= 1. \end{aligned} \tag{4.6}$$

The largest integer of the second column of the coding matrix that does not exceed the running total equals one, which is subtracted from the running total to obtain the value

$$\begin{aligned} \hat{c}_1 &= \hat{c}_2 - 1 \\ &= 0. \end{aligned} \tag{4.7}$$

The source decoder subsequently finds the largest integer of the first column that does not exceed the running total. This integer equals zero, which is subtracted from the running total $\hat{c}_1$.

Having iterated over all columns of the coding matrix, the source decoder marks each row of the matrix that contains an integer that was subtracted from the running total [5]. The first, third, fourth, sixth and seventh rows of the coding matrix in the example contain the relevant integers. This outcome implies that the index sequence of the example contains the integers one, three, four, six and seven. As this is the correct index sequence, the codeword was successfully decoded by the source decoder.

4.1.2 The comments of Davisson

Lynch [5] did not derive any bounds on the effectiveness of the first enumerative source code. Davisson [6] investigated the performance of the first enumerative source code, and provided an analytical definition of both the source encoder and decoder.

Davisson [6] recognized that the integer elements of the coding matrix [5] are equal to the binomial coefficients. Suppose that the coding matrix has m columns numbered from one to m, starting at the leftmost column of the matrix, and n rows numbered from one to n, starting at the bottom row of the matrix. The integer in row i and column k of the coding matrix, $e_{i,k}$, may be expressed as

$$e_{i,k} = \binom{i-1}{k}, \tag{4.8}$$

where

$$\binom{n}{k} \triangleq 0 \ \forall \ n < k \tag{4.9}$$

by definition. The relationship between the elements of the coding matrix and the binomial coefficients reveals that the coding matrix contains Pascal's triangle above its antidiagonal[2].

Davisson [6] provided an analytical definition of the enumerative source encoder. The source encoder calculates the integer codeword of the index sequence $\mathbf{x}^m = \{x_1, x_2, ...x_m\}$ using the equation

$$c_{\mathbf{x}} = \sum_{i=1}^{m} \binom{x_i - 1}{i}, \tag{4.10}$$

where $1 \leq x_i \leq n$.

Davisson [6] also provided an analytical definition of the enumerative source decoder. The running total at the start of the iteration that concerns column j of the coding matrix may be expressed as

$$\hat{c}_j = c_{\mathbf{x}} - \sum_{i=j+1}^{m} \binom{x_i - 1}{i}. \tag{4.11}$$

The integer elements of the index sequence are calculated in sequence from x_m to x_1

[2]The left side of Pascal's triangle is missing from the coding matrix, however.

using the expression

$$x_j = \max(u) : \hat{c}_j - \binom{u-1}{j} \geq 0 \tag{4.12}$$

$$= \min(u) : \hat{c}_j - \binom{u}{j} < 0. \tag{4.13}$$

Davisson [6] proposed a source code implementation for removing redundant data samples from a sample sequence. The implementation of Davisson is similar to the implementation proposed by Lynch [5]. Addendum A on page 255 contains a detailed summary of the implementation that was proposed by Davisson.

The implementation of Davisson [6] concatenates a binary prefix to each integer codeword of the enumerative code. This prefix is the conventional binary–coded representation of the integer m, which is the number of nonredundant samples, between each pair of consecutive time words, that are retained. The binary prefix consists of $\lceil \log_2(n+1) \rceil$ bits. By attaching the prefix to each codeword, the implementation can remove a variable number of redundant samples between different pairs of consecutive time words.

Davisson [6] derived the average code rate of the proposed implementation, which is the average number of bits that the implementation requires to represent a source sample. The average code rate was compared to the minimum rate that is necessary to encode source sequences of which the run lengths of samples follow a geometric distribution. Davisson proved that the code rate of the proposed implementation approaches the entropy of the source asymptotically as the number of samples n between consecutive time words tends to infinity. The derivation of the average code rate is quite condensed, however — a comprehensive derivation is carried out in addendum A.

Davisson [6] remarked that the average code rate of the implementation may converge slowly to the entropy of the source as n increases. Davisson also derived the average code rate of an implementation that uses a run–length code instead of an enumerative code. It was demonstrated that the code rate of this implementation is nearly optimal for a certain source. Davisson remarked that it is questionable whether an enumerative code produces codewords that are significantly shorter on average than the codewords of the run–length code for this source. As the run–length code is simple to implement, it may be the superior choice in certain cases.

4.1.3 Variable–to–fixed length enumerative source codes

Schalkwijk [60] proposed a variable–to–fixed length enumerative source code that is related to the enumerative source code of Lynch [5][3]. Schalkwijk used the same enumerative source code as the code proposed by Lynch to encode source sequences of bits. Instead of encoding the indices of nonredundant samples in a sequence, the enumerative source code is used to encode the indices of nonzero–valued bits in a bit sequence.

[3]It appears possible that Schalkwijk [60] was unaware of the work by Lynch [5], as it was not cited in his publication.

The source decoder reconstructs the original source sequence using the indices of the nonzero–valued bits, as well as the length of the original sequence.

Schalkwijk [60] proved that the enumerative source code may be used to rank distinct bit sequences of equal weight and length. The proof was carried out by mathematical induction on the length of the source sequence. The integer codewords assigned to n–bit sequences $\mathbf{x}^n$ of equal weight m satisfies the expression $0 \leq c_{\mathbf{x}} \leq \binom{n}{m} - 1$.

Schalkwijk [60] developed a variable–to–fixed length source code for sources that produce i.i.d. bits. Each bit of the source sequence $\mathbf{X}^n = \{X_1, X_2, \ldots X_n\}$ has a probability distribution that may be expressed as

$$\Pr(X_i = x_i) = \begin{cases} q & \text{if } x_i = 0, \\ p & \text{if } x_i = 1, \end{cases} \tag{4.14}$$

where $p+q = 1$, and $0 \leq p, q \leq 1$. The mean number of zero–valued and nonzero–valued bits in a source sequence is equal to nq and np, respectively.

The encoder of the variable–to–fixed length source code processes the source sequence in a bit–by–bit manner, starting at the front of the sequence [60]. It maintains running totals of the number of zero–valued bits and nonzero–valued bits that it encounters in the sequence. Let the running totals of the zero–valued and nonzero–valued bits, immediately prior to processing bit i of the sequence, be denoted by $y_0(i)$ and $y_1(i)$ respectively. Upon $y_0(i)$ equaling nq or $y_1(i)$ equaling np, for any value of i, the source encoder produces a codeword for the source bits that are reflected by the running totals, and resets both running totals to zero.

Suppose that the running total of the zero–valued bits equals nq prior to the source encoder processing bit i of the source sequence (i.e. $y_0(i) = nq$) [60]. The source encoder first copies the previous $y_0(i) + y_1(i)$ bits from the source sequence, which are the bits reflected by the running totals. It subsequently appends a total of $np - y_1(i)$ nonzero–valued bits to the sequence of copied bits, thereby obtaining a sequence of n bits that contains nq zero–valued bits and np nonzero–valued bits. This sequence is encoded using the enumerative source code for bit sequences of equal length and weight.

Suppose instead that the running total of the nonzero–valued bits equals np prior to the running total of the zero–valued bits being equal to nq [60]. The source encoder again copies the previous $y_0(i) + y_1(i)$ bits from the source sequence, but appends a total of $nq - y_0(i)$ zero–valued bits to the copied sequence in this case. This n–bit sequence contains nq zero–valued bits and np nonzero–valued bits. It is encoded using the enumerative source code for bit sequences of equal length and weight.

The source encoder doesn't need to encode the number of zero–valued bits or nonzero–valued bits of each sequence that it encodes [60]. Each sequence contains a total of nq zero–valued and np nonzero–valued bits. The source encoder only produces the integer codeword of the sequence, using the encoder of the enumerative source code. It encodes this integer using the conventional binary–coded representation of an integer. Each codeword requires $\left\lceil \log_2 \binom{n}{np} \right\rceil$ bits to represent.

The source decoder of the variable–to–fixed length source code divides the encoded sequence into blocks containing $\left\lceil \log_2 \binom{n}{np} \right\rceil$ bits each [60]. The sequence of each block is decoded using the enumerative source code for bit sequences, thereby obtaining an n–bit sequence for each block. As these n–bit sequences contain redundant bits

(i.e. the bits that were appended to the original source sequence), the source decoder independently processes the bits of each sequence in a bit–by–bit manner, starting at the first bit of each sequence. It maintains running totals of the zero–valued bits and nonzero–valued bits that it encounters in each sequence. As soon as the running total of the zero–valued bits equals nq, or the running total of the nonzero–valued bits equals np, the source decoder removes the remaining bits from the sequence. It finally concatenates all the processed sequences, thereby obtaining the original source sequence.

The effectiveness of the variable–to–fixed length source code depends on the average number of bits that are appended to each variable–length sequence. It is reasonable to expect that the fraction of appended bits in each n–bit sequence becomes smaller as longer sequences are considered, due to the law of large numbers. The effectiveness of the variable–to–fixed length source code should therefore improve when implemented with larger values of n. This is indeed the case, as proved by Schalkwijk [60].

Schalkwijk [60] derived expressions for the effectiveness of the variable–to–fixed length enumerative source code. The effectiveness of the source code is expressed in terms of the average number of source bits that are represented by each bit that the source encoder produces. The source encoder represents a total of

$$r_a^{-1} = \frac{n}{\left\lceil \log_2 \binom{n}{np} \right\rceil} \tag{4.15}$$

source bits and appended bits per codeword bit. The appended bits do not convey any information, and should be distinguished from the source bits in order to evaluate the effectiveness of the source code, however.

In order to derive an appropriate bound on the rate of the variable–to–fixed length source code, Schalkwijk [60] associated each source sequence with a random walk through a coding array. This coding array is different from the coding matrix used by the source code implementation of Lynch [5], however. The coding array is a two–dimensional array of nodes with $nq + 1$ columns and $np + 1$ rows.

The rows of the coding array are numbered from zero to np, starting at the bottom row of the array [60]. The columns of the coding array are numbered from zero to nq, starting at the leftmost column of the array. All nodes in the same row of the array represent sequences with the same number of nonzero–valued bits. The number of nonzero–valued bits in each sequence of a row is equal to the row number. Nodes in the same column of the array represent sequences with the same number of zero–valued bits. The number of zero–valued bits in each sequence of a column is equal to the column number.

A source sequence may be represented with the coding array by processing the sequence in a bit–by–bit fashion, and moving a marker from one node of the array to another depending on the value of each bit [60]. The marker is initially placed at the bottom–left node of the coding array. As each consecutive bit of the source sequence is processed, the marker moves either to the node above it (if the source bit is nonzero) or to the node on its right (if the source bit is zero). Upon reaching the final column or the final row, the sequence is padded to n bits using nonzero–valued bits or zero–valued bits, and the marker is moved to the top–right node of the coding array.

Schalkwijk [60] derived the probabilities associated with the marker reaching the nodes in the final column and final row of the coding array. These probabilities were used to derive a probability distribution for the number of source bits that appear in each n–bit sequence that is encoded. The mean of this distribution, k_{ave}, is expressed as

$$k_{\text{ave}} = n\left[1 - \binom{n}{np} p^{np} q^{nq}\right]. \tag{4.16}$$

Stirling's approximation [44] of the factorial of an integer n,

$$n! \approx (2\pi n)^{1/2} n^n e^{-n}, \tag{4.17}$$

may be substituted into equation 4.16 to obtain the approximation

$$k_{\text{ave}} \approx n[1 - (2\pi pqn)^{-1/2}]. \tag{4.18}$$

The approximation of the mean value k_{ave} (equation 4.18) becomes more accurate as n increases. The average fraction of source bits per n–bit sequence is approximated as

$$\frac{k_{\text{ave}}}{n} \approx 1 - (2\pi pqn)^{-1/2}. \tag{4.19}$$

The limit of the average fraction of source bits per n–bit sequence, as n tends to infinity, equals

$$\begin{aligned} \lim_{n\to\infty} \frac{k_{\text{ave}}}{n} &= 1 - \lim_{n\to\infty} (2\pi pqn)^{-1/2} \\ &= 1. \end{aligned} \tag{4.20}$$

The number of source bits per encoded bit, as n tends to infinity, may therefore be calculated as

$$\begin{aligned} r_a^{-1} &= \lim_{n\to\infty} \frac{k_{\text{ave}}}{\left\lceil \log_2 \binom{n}{np} \right\rceil} \\ &= \lim_{n\to\infty} \frac{n}{\left\lceil \log_2 \binom{n}{np} \right\rceil} \\ &= [h(p)]^{-1}, \end{aligned} \tag{4.21}$$

where $h(\cdot)$ is the binary entropy function. The last step of equation 4.21 is obtained by approximating the binomial coefficient using Stirling's approximation of the factorial of an integer. Equation 4.21 proves that the source code is asymptotically optimal in terms of its effectiveness.

Schalkwijk [60] generalized the enumerative source code for sequences of bits to include q–ary symbol sequences, where $q \geq 2$. The codeword for a length–n sequence $\mathbf{x}^n$ of q–ary symbols may be calculated using the expression

$$c_{\mathbf{x}} = \sum_{k=1}^{n} \sum_{d=0}^{x_k - 1} \left\{ (n-k)! / \left[(w_k^d - 1)! \prod_{i=0, i\neq d}^{q-1} (w_k^i)! \right] \right\}, \tag{4.22}$$

where $(-1)! = \infty$ and w_k^d is the number of symbols in the sequence $\mathbf{x}_k^n$ that equal d, with $0 \leq d \leq q-1$. An implementation of this algorithm may use a q–dimensional array of multinomial coefficients. Schalkwijk stated that this source code may be verified by mathematical induction on the length of the source sequence. Schalkwijk did not define the decoder of the enumerative code for q–ary symbol sequences analytically, however.

Schalkwijk [60] claimed that the computational complexity of the variable–to–fixed length source code is comparable to that of the Elias source code [154], assuming that both are implemented using relatively short block lengths. The Elias source code is essentially a fixed–to–variable length arithmetic code. Arithmetic codes had to be implemented using short block lengths at the time reference [60] was published. This was due to the fact that the probability interval of a source sequence, as produced by an arithmetic code, shrinks exponentially as the sequence becomes longer. Digital computer implementations of the Elias source code had limited accuracy at the time, and could not accurately represent very short probability intervals.

The problem of accurately representing short probability intervals with a digital computer has since been resolved [16]. Modern implementations of arithmetic codes may be applied to sequences that are arbitrarily long. An arithmetic source code may also be used to encode sequences with arbitrary symbol distributions. These advantages are part of the reason for the success of arithmetic codes.

Schalkwijk [60] mentioned a fixed–to–variable length source code for sequences of i.i.d. bits. The encoder of this source code first encodes the weight of the source sequence, after which the source sequence is encoded using the enumerative source code. This source code was not investigated in reference [60], but it is regarded in section 6.2.1 on page 166.

4.1.4 Generalization of enumerative source codes

Cover [7] proposed a general enumerative source code for bit sequences. This general enumerative source code may be used to obtain the index of any sequence $\mathbf{x}^n = \{x_1, x_2, \ldots x_n\}$ in any ordered subset of all bit sequences of length n.

The enumerative source code proposed by Cover [7] concerns any ordered subset S_b of all bit sequences of length n,

$$S_b \subseteq \{0,1\}^n. \tag{4.23}$$

The sequences of the subset S_b are sorted in ascending order under the assumption that the first bit of each sequence (x_1) is the most significant bit. Cover stated that the index of any sequence $\mathbf{x}^n \in S_b$ may be calculated using the expression

$$i_{S_b}(\mathbf{x}^n) = \sum_{j=1}^{n} x_j n_{S_b}(x_1, x_2, \ldots x_{j-1}, 0), \tag{4.24}$$

where $n_{S_b}(x_1, x_2, \ldots x_{j-1}, 0)$ denotes the number of sequences in S_b with the prefix $\{x_1, x_2, \ldots x_{j-1}, 0\}$.

Cover [7] did not provide a rigorous mathematical proof of the validity of equation 4.24, but justified it using a simple argument. Equation 4.24 may instead be verified as follows. Consider a full binary tree with $n+1$ levels, including the root node. Assume

that the left child of each internal tree node is labeled with the value zero, and that the right child of each internal tree node is labeled with the value one. Let each path from the root node to a leaf node correspond to a bit sequence that equals the concatenation of the node labels on the path.

Suppose that the index of the n–bit sequence $\mathbf{x}^n$, in a certain subset S_b, is to be calculated. This index may be calculated by traversing all nodes in the tree that correspond to the path associated with the sequence. At each internal node on the path, the path may either branch to the left child node or the right child node.

If the path branches to the right child node, all sequences associated with the descendents of the left child node are smaller than the sequence $\mathbf{x}^n$. If the path branches to the left child node, none of the sequences associated with the descendents of the right child node are smaller than the sequence $\mathbf{x}^n$. By adding the number of descendents (i.e. the number of leaf nodes) of the left child node at each internal node where the path branches to the right child node, the total number of bit sequences smaller than $\mathbf{x}^n$ may be calculated.

In order to verify equation 4.24, observe that each nonzero term of the sum may be associated with an internal node of the tree where the path branches to the right child node (i.e. x_j equals one). Each of these terms equals the number of descendents of the internal node's left child, but which belong to the subset S_b.

The elements of the subset S_b determine whether the function $n_{S_b}(\cdot)$ can be readily evaluated. Cover [7] considered various subsets, and derived an expression for the function $n_{S_b}(\cdot)$ that is associated with each subset. Some of these expressions are provided in what follows.

1. If S_b is the entire set of n–bit sequences, $n_{S_b}(x_1, x_2, \ldots x_k) = 2^{n-k}$. Any sequence of this set is assigned an integer codeword by interpreting it as the conventional binary–coded representation of an integer, and calculating its decimal value.

2. If S_b is the set of all n–bit sequences with equal weight w,

$$n_{S_b}(x_1, x_2, \ldots x_{k-1}, 0) = \binom{n-k}{m(w,k)}, \tag{4.25}$$

where $m(w,k) = w - \sum_{i=1}^{k-1} x_i$. This expression is equivalent to the analytical definition of the enumerative source encoder for bit sequences that was proposed by Schalkwijk [60].

3. If S_b is the set of all n–bit sequences with weights between (and including) $w-r$ and $w+r$ (in other words, $|\sum_{i=1}^{n} x_i - w| \leq r$),

$$n_{S_b}(x_1, x_2 \ldots x_{k-1}, 0) = \sum_{i=w-r}^{w+r} \binom{n-k}{m(i,k)}. \tag{4.26}$$

The usefulness of this expression becomes apparent when considering a source that produces i.i.d. bits, with $\Pr(X_i = 1) = p$. According to the law of large numbers, the normalized weight of any sequence of length n from this source satisfies

$$\frac{1}{n} \sum_{i=1}^{n} x_i \to p \tag{4.27}$$

as n tends to infinity. An enumerative source code may be constructed for sequences from this source by setting w equal to np, and selecting a sufficiently large value for r to ensure that the probability of the source producing a sequence $\mathbf{x}^n$ that is not present in S_b is negligible. Length–n sequences from this source may then be encoded using the enumerative source encoder defined by equations 4.24 and 4.26.

Cover [7] extended the general enumerative source code to include ordered subsets of integer sequences, in addition to subsets of bit sequences. The subset S_b that consists of all $n!$ permutations of the sequence $\mathbf{x}^n = \{1, 2, 3, \ldots n\}$ was investigated. The expression

$$i_{S_b}(\mathbf{x}^n) = \sum_{i=1}^{n} r_i (n-i)! \tag{4.28}$$

holds for this subset, where

$$r_i = |\{x_j : i+1 \leq j \leq n \wedge x_j < x_i\}|. \tag{4.29}$$

Cover [7] also extended the general enumerative source code to include ordered subsets of discrete monotone functions. Consider an integer–valued function $F : \{0, 1, \ldots m\} \rightarrow \{0, 1, \ldots n\}$. If $F(i) \leq F(i+1)$ for all i the function F is referred to as a monotone nondecreasing function. If $F(i) < F(i+1)$ for all i the function F is referred to as a strictly monotone increasing function. If $F(m) = n$, the function F is referred to as a distribution function. A function F is considered smaller than a function G if $F(y) < G(y)$ where $y = \min(k) : F(k) \neq G(k)$.

Cover [7] derived enumerative source codes for the functions in the following ordered subsets of discrete monotone functions:

1. $S_b = \{F : F$ monotone nondecreasing function$\}$,
2. $S_b = \{F : F$ monotone nondecreasing distribution function$\}$,
3. $S_b = \{F : F$ strictly monotone increasing function$\}$, and
4. $S_b = \{F : F$ strictly monotone increasing distribution function$\}$.

Cover stated that discrete convex functions may also be enumerated in a similar manner, but did not derive an enumerative source code for these functions in reference [7].

Cover [7] proposed an enumerative source code for sequences from the two–state, first–order, binary Markov source of figure B.1 on page 260. A detailed summary of this source code is provided in addendum B on page 259. This section contains only a short description of the source code.

The bit sequences from the first–order Markov source may be enumerated by counting the occurrences of all distinct pairs of consecutive bits (i.e. {00, 01, 10, 11}) in each source sequence [7]. The frequency counts of the distinct bit pairs of a source sequence are collectively referred to as the count profile of the sequence.

All sequences of the first–order Markov source that have the same count profile are equiprobable, and constitute an ordered subset of sequences. The enumerative source

encoder for bit sequences from the first–order Markov source calculates and encodes the index of a sequence in its subset [7]. The enumerative source encoder also encodes the count profile of each sequence, as the source decoder requires the count profile in order to select the correct subset.

Cover [7] stated that the code rate of the proposed source code approaches the entropy rate of the source asymptotically. A comprehensive derivation of the rate of the code is not provided in reference [7], however. The expression for the rate of the code is derived in addendum B on page 259.

4.1.5 Fixed–to–fixed length enumerative source codes

An n–bit to k–bit fixed–length source code assigns a k–bit codeword to each n–bit source sequence, where $k < n$. This source code is lossy, as the number of distinct source sequences exceeds the number of distinct codewords[4]. Some codewords are therefore assigned to more than one distinct source sequence, and may be incorrectly decoded by the source decoder.

A block–error probability P_b is associated with a fixed–to–fixed length source code and a specific source. The block–error probability is defined as the probability that the source decoder will incorrectly decode the codeword of a random sequence from the source. An optimal fixed–to–fixed length source code (i.e. a code that has the minimum block–error probability P_b when used to encode sequences from a certain source) assigns each of the 2^k most likely n–bit source sequences a distinct k–bit codeword.

Theoretical bounds on the code rate and block–error probability of certain fixed–to–fixed length source codes have been derived [155]. The derivation of the bounds that are discussed in this thesis was carried out under the assumptions that each source code is applied to sequences $\mathbf{X}^n = \{X_1, X_2, \ldots X_n\}$ of i.i.d. bits, and that each source bit has the distribution of equation 4.14 on page 109. The distribution is restricted to the case where $0 < p < 0.5$. The bounds concern fixed–to–fixed length source codes that assign a distinct codeword to each source sequence in the set $\mathcal{A}_\delta^n$, which contains all source sequences with weight $w \leq \lfloor \delta n \rfloor$, where $0 < \delta \leq 0.5$. This set is also referred to as a Hamming ball.

The first bound concerns the code rate (i.e. the number of codeword bits per source bit) [155]. This bound is valid provided that $0 < p < \delta \leq 0.5$. The bound on the code rate is derived as

$$\begin{aligned} R_a &= \frac{k}{n} \\ &= \frac{\lceil \log_2 |\mathcal{A}_\delta^n| \rceil}{n} \\ &\leq h(\delta) + \frac{1}{n}, \end{aligned} \tag{4.30}$$

where $h(\cdot)$ is the binary entropy function.

The second bound concerns the block–error probability P_b associated with the source code and the source [155]. This bound is valid for the case where $0 < p < \delta \leq 0.5$.

[4] This statement is valid if each distinct n–bit sequence has a nonzero probability of being produced by the source.

The bound is derived as

$$\begin{aligned} P_b &= \Pr(\mathbf{X}^n \notin \mathcal{A}_\delta^n) \\ &\leq 2^{-nD(\delta||p)}, \end{aligned} \tag{4.31}$$

where $D(\delta||p)$ is the binary divergence function (i.e. the Kullback–Leibler divergence), which is defined as

$$D(\delta||p) = \delta \log_2 \left(\frac{\delta}{p}\right) + (1-\delta) \log_2 \left(\frac{1-\delta}{1-p}\right). \tag{4.32}$$

These bounds imply that it is possible to encode sequences of i.i.d. bits using a fixed–length source code with a code rate that does not exceed $h(p) + \epsilon$ codeword bits per source bit, where ϵ is a positive constant that tends to zero as the sequence length n tends to infinity. The block–error probability of this source code approaches zero as the sequence length n tends to infinity.

Willems [155] investigated a fixed–to–fixed length enumerative source code for a source that produces sequences of i.i.d. bits $\mathbf{X}^n = \{X_1, X_2, \ldots X_n\}$, where each bit has the same distribution as defined for the derivation of the bounds. The enumerative source encoder obtains the index of any source sequence with weight $0 \leq w \leq \lfloor \delta n \rfloor$ in the set $\mathcal{A}_\delta^n$, where $0 \leq \delta \leq 1$. This index is encoded as a fixed–length bit sequence with a length of $\lceil \log_2 |\mathcal{A}_\delta^n| \rceil$ bits. The sequences in the set $\mathcal{A}_\delta^n$ may be interpreted as binary–coded integers that are sorted in ascending order according to their values (this is equivalent to a lexicographical order). The first bit of each sequence is considered the most significant bit.

The implementation of the fixed–to–fixed length enumerative source code of Willems [155] represents source sequences in the set $\mathcal{A}_\delta^n$ using a matrix of nodes (refer to figure 4.4). This matrix is similar to the coding matrix used by Lynch [5], as well as the coding array of Schalkwijk [60].

The node matrix has $n+1$ columns and $\lfloor \delta n \rfloor + 1$ rows [155]. A source sequence is represented with the node matrix by processing the sequence in a bit–by–bit manner, and moving a marker from one node to another as each bit is processed. The marker is initially placed at the bottom–left node of the node matrix. Upon encountering a zero–valued bit in the source sequence, the marker is moved to the node in the column to its right. If a nonzero–valued bit is encountered in the source sequence, the marker is moved to the node in the row above it. An n–bit sequence in the set $\mathcal{A}_\delta^n$ is therefore represented as a path from the bottom–left node of the matrix to one of the end nodes, which are coloured black in figure 4.4.

Each node of the node matrix is assigned a count [155]. Each count equals the total number of paths that leave the node and terminate at an end node. These counts are referred to as path counts. The end nodes are assigned path counts of one. Each node that borders the end nodes is assigned a path count that equals the number of end nodes that may be reached by passing through it. Each of the remaining nodes of the node matrix is assigned a path count that is the sum of the path count of the node above it and the path count of the node to its right[5]. The path count of the bottom–left node equals the number of distinct paths that reach an end node. This path count equals the number of distinct n–bit source sequences in the set $\mathcal{A}_\delta^n$.

[5]If the node is in the top row of the matrix, it is assigned the path count of the node to its right.

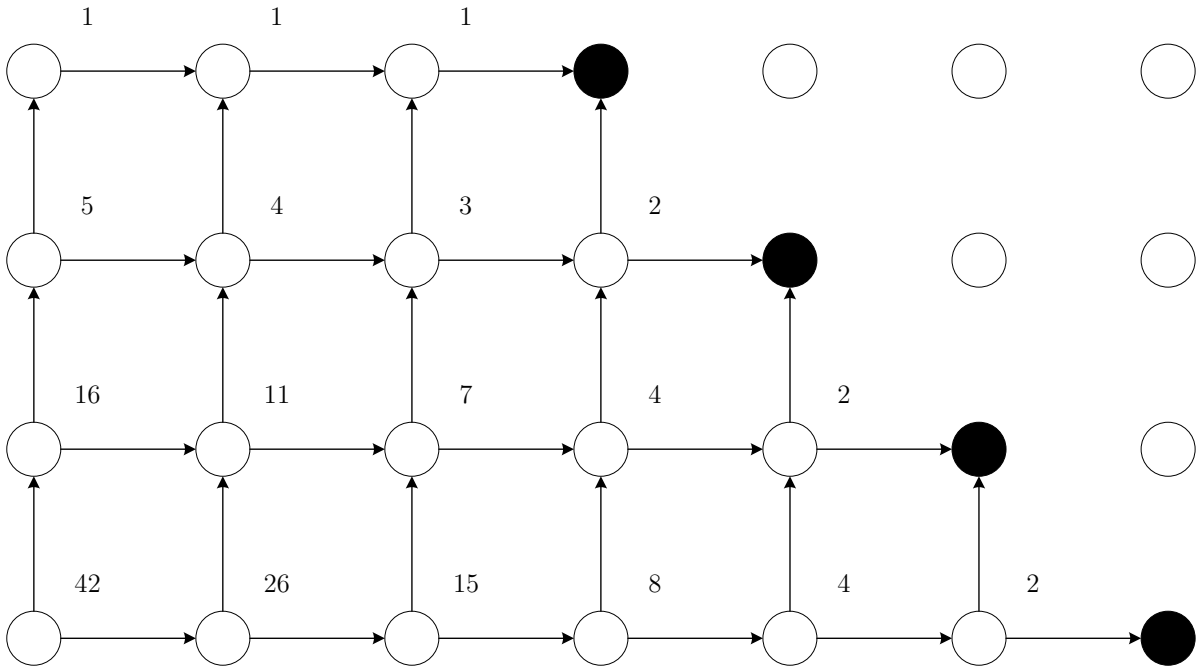


Figure 4.4: Representation of six–bit source sequences using a node matrix, where $\lfloor \delta n \rfloor = 3$ (adapted from reference [155]).

Consider a path through the node matrix that corresponds to a certain sequence $\mathbf{x}^n$ in the set $\mathcal{A}_\delta^n$ [155]. Let each of the path's nodes from which the marker moves to the node in the row above it (i.e. each node that corresponds to a nonzero–valued bit in the sequence) be referred to as a nonzero–bit node. As the source sequence is processed from the MSB to the LSB, all sequences that correspond to paths that branch to the node on the right–hand side of a nonzero–bit node are smaller than the sequence $\mathbf{x}^n$. The number of paths that branch to the node on the right–hand side of each nonzero–bit node equals the path count of the node to its right.

The integer codeword of a sequence $\mathbf{x}^n$ in the set $\mathcal{A}_\delta^n$ is calculated by considering all nonzero–bit nodes on the node matrix path that corresponds to the source sequence [155]. All the path counts of the nodes on the right–hand side of the nonzero–bit nodes are added together. This sum equals the total number of sequences in the set $\mathcal{A}_\delta^n$ that are smaller than the source sequence. The sum of the path counts is therefore equal to the integer codeword of the sequence $\mathbf{x}^n$.

The source decoder of the enumerative source code maintains a running total that is initially set equal to the integer codeword [155]. The source decoder iterates over the rows of the node matrix, starting at the bottom row of the matrix. It subtracts the path count of a certain node in each row from the running total prior to continuing with the next iteration. Each path count that is subtracted from the running total equals the largest path count among the nodes in the row that does not exceed the running total.

Let the node with the largest path count that does not exceed the running total in each row be referred to as a branching node [155]. The node immediately to the

left of each branching node is a nonzero–bit node. As the path of the source sequence branches to the node above each nonzero–bit node, the source decoder may obtain the indices of all nonzero–valued bits in the sequence. As the remaining bits of the sequence are zero–valued bits, the source sequence may be successfully recovered from the codeword.

4.1.6 Hierarchical enumerative source codes

Oktem et. al. [145] proposed a hierarchical enumerative source code for sequences of bits. This blockwise source code may be used to effectively encode bits that are statistically independent, but not necessarily identically distributed.

The source encoder of the enumerative code initially divides the source sequence into blocks of equal length [145]. It repeatedly subdivides these blocks until the sequence is divided into short blocks of equal length. The repeated division of the sequence implies a hierarchy of blocks. The shortest blocks belong at the bottom level of the hierarchy, while the entire sequence belongs at the top level of the hierarchy.

The source encoder independently encodes each of the blocks at the bottom level of the hierarchy using a fixed–to–variable length enumerative source code for sequences of i.i.d. bits [145]. The codeword of each block consists of a prefix and a suffix. The prefix equals the weight of the block sequence. The suffix equals the index of the block sequence in an ordered set of all sequences of the same length and weight. The suffix is encoded using the conventional binary–coded representation of integers.

The block hierarchy is motivated by considering the rate of adaptation of the fixed–to–variable length enumerative source code [145]. If the source encoder of the fixed–to–variable length source code is used to independently encode longer block sequences, it cannot adjust to rapid, drastic changes in the distribution of the source bits. If it is used to independently encode shorter block sequences, it may adapt to more rapid changes in the distribution.

The drawback of encoding shorter block sequences with the fixed–to–variable length source code concerns the prefix of each codeword [145]. The prefix of each codeword is the binary–coded representation of the weight of a block sequence, and is necessary to successfully decode the codeword. The weights of several shorter blocks require more bits to represent in total than the sum of the weights of the shorter blocks[6]. A tradeoff exists between the rate of adaptation of the source encoder, and the effective source coding of the sequence weight. The hierarchy of blocks may be used obtain the advantages of encoding both shorter blocks and longer blocks (i.e. rapid adaptation and little overhead).

The source encoder of the hierarchical source code does not encode the weights of the block sequences at the bottom level of the hierarchy directly [145]. It initially encodes the total weight of the sequence (i.e. the top–level block). It subsequently iterates over the levels of the hierarchy, starting at the second–from–top level and ending at the bottom level. At each level of the hierarchy, it produces a number of codewords. Each codeword represents the individual weights of all block sequences that form part of the same block in the next higher level of the hierarchy.

[6]This statement is valid as the logarithm function is a concave function.

Let all blocks of a certain level of the hierarchy that form part of the same block in the next higher level be referred to as a block segment [145]. The source encoder uses the total weight of the block sequences in each block segment as it produces a single codeword for the individual weights of the block sequences in that block segment. It does not encode the total weight of the block sequences in a block segment, however. The reason for not encoding the total weight of the block sequences in a block segment may be understood by considering the source decoder.

The source decoder decodes the block weights in a level–by–level manner, starting at the top level of the hierarchy [145]. Suppose that the source decoder is to recover the block weights of a block segment in a certain level of the hierarchy. The weight of each block sequence in the next higher level of the hierarchy is known to the source decoder before it starts to decode the individual block weights of the block segment. It is therefore able to decode the individual block weights successfully. Upon reaching the bottom level of the hierarchy, the source decoder recovers the weights of the individual block sequences, and decodes each of the block codewords to recover the source sequence.

The source encoder encodes the individual weights of the block sequences in a block segment as follows [145]. It interprets the total weight of the block sequences as a sequence of unity–weight elements. The sequence contains a total of w unity–weight elements, assuming that the total weight of the block sequences equals w bits. Certain unity–weight elements in the sequence are separated by division elements — the weight elements between each pair of consecutive division elements belong to a certain block of the segment. The sequence contains a total of $k-1$ division elements, assuming each block segment consists of k blocks.

Each sequence of unity–weight elements and division elements may be represented as a bit sequence by associating unity–weight elements with zero–valued bits, and the division elements with nonzero–valued bits [145]. An enumerative code is used to encode each of these bit sequences. The integer codeword of each sequence equals its index in an ordered set of all bit sequences of length $w+k-1$, and with a weight of $k-1$ bits. Each integer is encoded using its conventional binary–coded representation. Each codeword requires a total of

$$l(w,k) = \left\lceil \log_2 \binom{w+k-1}{k-1} \right\rceil \tag{4.33}$$

bits to represent.

Oktem et. al. [145] observed that the individual weights of the block sequences in a block segment may be ineffectively encoded in some scenarios. As an example, consider the case where the total weight of the block sequences in a segment equals the total length of the block sequences in the segment. All block sequences of this segment consist solely of nonzero–valued bits. The source encoder need not produce any codeword in this case. The source encoder will, however, produce a codeword with a nonzero length (refer to equation 4.33).

Oktem et. al. [145] improved the hierarchical enumerative source code by encoding the block weights of a block segment differently depending on whether the block segment has a majority of zero–valued or nonzero–valued bits. If the block segment contains a majority of nonzero–valued bits, the source encoder encodes the number of

zero–valued bits in the block sequences of the segment. If the block segment contains a majority of zero–valued bits, the weights of the block sequences are encoded as before. The source encoder need not indicate whether it encodes the number of zero–valued or nonzero–valued bits of the blocks in a segment, as the source decoder may obtain this information from the blocks on the higher levels of the hierarchy.

Oktem et. al. [145] implemented the hierarchical enumerative source code, and compared the performance of the implementation to the performance of two alternative source code implementations. An adaptive arithmetic source code implementation, as well as the Unix software utility `compress` [70] (which is essentially an implementation of the LZW source code[7]) were considered. The implementations were used to source code the output of three abstract information sources, two of which were nonstationary.

The proposed implementation proved to be significantly more effective than the other implementations when used to encode source sequences from the two nonstationary sources [145]. It was slightly less effective than the adaptive arithmetic code when used to encode sequences from the stationary source. The implementations were not compared in terms of their computational complexity, however.

Oktem et. al. [146] generalized the hierarchical enumerative source code in order to improve its effectiveness when used to encode bits that are statistically dependent on one another. The general source code may be used to effectively encode bits from nonstationary first–order binary Markov sources (refer to figure B.1 on page 260 for the state–transition diagram of a first–order binary Markov source).

The general hierarchical enumerative source code may be interpreted as an improvement of the enumerative source code for first–order binary Markov sources, which was proposed by Cover [7]. The enumerative code proposed by Cover is summarized in section 4.1.4 on page 112. The source encoder of Cover's enumerative source code obtains the count profile of a source sequence prior to encoding it. It subsequently calculates the integer index of the source sequence in an ordered set that contains all bit sequences with the same count profile. The source encoder encodes the count profile of the sequence, as well as the integer index.

Oktem et. al. [146] observed that the weight of a source sequence may be expressed in terms of its count profile. Let the frequency count of the bit pair ab in the source sequence be denoted by m_{ab}, where $ab \in \{0,1\}^2$. The weight of the source sequence may be expressed as

$$w = m_{01} + m_{11}, \tag{4.34}$$

assuming that the first bit of the source sequence has a zero value. It follows that

$$m_{00} + m_{10} = n - w - 1, \tag{4.35}$$

where n is the length of the sequence.

The encoder of the general hierarchical enumerative source code encodes the weights associated with the block sequences of each block segment on each level of the hierarchy [146]. The weights are encoded in an identical fashion to the ordinary hierarchical enumerative source code. In addition, the source encoder encodes the frequency count m_{11} of each block sequence at the bottom of the hierarchy using the conventional

[7]Refer to section 2.6.5 on page 28 for a summary of Lempel–Ziv source codes.

binary–coded representation of integers. A total of $\lceil \log_2(w_i) \rceil$ bits are required to represent the frequency count m_{11} of a block sequence with a weight of w_i bits, where $w_i > 0$.

By encoding both the weight and the frequency count m_{11} of each block sequence, the source encoder enables the source decoder to calculate the remaining bit–pair frequency counts of each block sequence at the bottom of the hierarchy as follows [146]. It first calculates the frequency count m_{01} using equation 4.34, the sequence weight and the value of m_{11}. The source decoder next obtains the frequency count m_{10}, as this frequency count is equal to m_{01} (assuming the first and last bits of the sequence are both zero–valued) [7]. The frequency count m_{00} is subsequently calculated using equation 4.35.

Oktem et. al. [146] demonstrated the effectiveness of the general hierarchical enumerative source code by implementing it, and using it to encode bit sequences from three abstract, first–order binary Markov sources. One of the three sources was a stationary first–order Markov source, while the remaining sources were piecewise–stationary first–order Markov sources. The code rate of the implementation was compared to the code rate of an implementation that uses an adaptive arithmetic code, as well as the code rate of the Unix software utility `compress` [70].

The general hierarchical enumerative source code proved to be more effective than the source codes it was compared to [146]. The computational complexity of the implementation was not compared to the complexity of the other implementations, however. Oktem et. al. did state that the binomial coefficients required by both the enumerative encoder and decoder should be stored in a look–up table to improve the efficiency of the implementation.

4.1.7 Binary combinatorial codes

Dai et. al. [149] proposed a fixed–to–variable length enumerative source code that is known as a binary combinatorial code. The source encoder of this source code calculates the integer codeword of the source sequence, which equals its index in an ordered set of all sequences with the same length and weight as the source sequence. It encodes the weight of the source sequence, as well as its integer index. The integer index is encoded using the conventional binary–coded representation of an integer. The source encoder encodes the weight of the source sequence differently than conventional enumerative source codes.

Dai et. al. [149] stated that the encoder of a conventional enumerative source code has to perform n–bit integer addition and storage to rank an n–bit source sequence. This requirement limits the block length of blockwise enumerative source codes, as the majority of modern digital computer systems have word lengths of 32 or 64 bits. The source coding of longer source sequences is more effective, as the length of the conventional binary codeword of the source sequence weight w is a logarithmic function of the sequence length.

The encoder of the source code proposed by Dai et. al. [149] uniformly divides a long source sequence into short blocks of m bits, and independently encodes the sequence of each block. The weight of each block sequence is encoded using a Huffman code, as the conventional binary–coded representation of the integer weights of short blocks is

ineffective. A static Huffman code is constructed for the weights of the block sequences.

Dai et. al. [149] assumed that the source produces i.i.d. bits, which implies that the weight of a source sequence has a binomial distribution. As the Poisson distribution may be interpreted as an approximation of the binomial distribution, the Huffman code is designed for weights with a Poisson distribution. The parameter of the Poisson distribution is selected as

$$\lambda = mp, \tag{4.36}$$

where p denotes the probability of a source bit having a nonzero value, and m is the length of each block sequence.

The proposed source code was implemented and used to encode discrete cosine transform coefficients from a video codec, as well as a binary image of VLSI layout data [149]. The implementation proved to be more effective and more efficient than implementations of a Huffman code and an arithmetic code. The authors did not specify how the source code may be generalized in order to effectively encode bits that are statistically dependent on one another, however.

4.1.8 The efficient enumerative source codes of Ryabko

Ryabko [152] proposed an enumerative source code implementation with a computational complexity that increases sublinearly w.r.t. the length of the source sequence. The efficient implementation was derived by rewriting the expression for the integer codeword of a source sequence.

Suppose that an ordered subset S_b of all length–n symbol sequences is specified, with each symbol belonging to the alphabet $\mathcal{A}$ (i.e. $S_b \subseteq \mathcal{A}^n$). The index[8] of a sequence $\mathbf{x}^n$ in the ordered subset S_b may be calculated as [7]

$$i_{S_b}(\mathbf{x}^n) = \sum_{i=1}^{n} \sum_{\{y:y<x_i\}} n_{S_b}(x_1, x_2, \ldots x_{i-1}, y), \tag{4.37}$$

where $n_{S_b}(x_1, x_2, \ldots x_{i-1}, y)$ denotes the number of sequences in S_b with the prefix $\{x_1, x_2, \ldots x_{i-1}, y\}$.

Ryabko [152] defined several intermediate terms, and expressed the index of equation 4.37 using the intermediate terms. The intermediate terms are defined as

$$P(x_1) = \frac{n_{S_b}(x_1)}{|S_b|} \tag{4.38}$$

and

$$P(x_k|x_1, x_2, \ldots x_{k-1}) = \frac{n_{S_b}(x_1, x_2, \ldots x_k)}{n_{S_b}(x_1, x_2, \ldots x_{k-1})}, \tag{4.39}$$

where $k = \{2, 3, \ldots n\}$, as well as

$$Q(x_k|x_1, x_2, \ldots x_{k-1}) = \sum_{\{y:y<x_k\}} P(y|x_1, x_2, \ldots x_{k-1}), \tag{4.40}$$

[8] Refer to section 4.1.4 on page 112 for the origin of this expression.

where $k = \{1, 2, \ldots n\}$.

The index of a source sequence $\mathbf{x}^n$ may be expressed using the intermediate terms as [152]

$$\begin{aligned} i_{S_b}(\mathbf{x}^n) &= |S|\big(Q(x_1) + Q(x_2|x_1)P(x_1) + Q(x_3|x_1, x_2)P(x_1)P(x_2|x_1) + \ldots\big) \\ &= |S| \sum_{i=1}^{n} \left[Q(x_i|x_1, x_2, \ldots x_{i-1}) \prod_{j=1}^{i-1} P(x_j|x_1, x_2, \ldots x_{j-1}) \right] \\ &= |S| \sum_{i=0}^{n/2-1} \left[\prod_{j=1}^{2i} P(x_j|x_1, \ldots x_{j-1}) \right] \Big[Q(x_{2i+1}|x_1, \ldots x_{2i}) \\ &\quad + Q(x_{2i+2}|x_1, \ldots x_{2i+1}) P(x_{2i+1}|x_1, \ldots x_{2i}) \Big]. \end{aligned} \tag{4.41}$$

The implementation proposed by Ryabko [152] represents each intermediate term using a total of $2\log(n) + O(1)$ digits. It uses the Schonhager–Strassen [156] method of multiplying and dividing numbers to efficiently evaluate equation 4.41.

Ryabko [152] stated that the computational complexity of the encoder and decoder is proportional to $O(\log^3(n).\log\log(n))$ as n tends to infinity, where n is the length of the source sequence. The computational complexity is expressed in terms of the number of operations that are performed on single–bit words.

4.1.9 Q–ary enumerative source codes

Schalkwijk [60] proposed an enumerative source code for sequences of q–ary symbols, where $q \geq 2$. The source encoder of this enumerative code calculates the index of a sequence of q–ary symbols in an ordered set that contains sequences with the same length as the source sequence, and with the same number of symbols of each type. The calculation of the index involves the computation of multinomial coefficients if $q > 2$, as is evident from equation 4.22 on page 111.

Schalkwijk [60] did not comment on the computational complexity of the source encoder for sequences of q–ary symbols, where $q > 2$. Tanaka et. al. [150] speculated that this enumerative code may be ineffective or overly complex to implement. Tanaka et. al. subsequently derived a new enumerative source code for sequences of q–ary symbols.

The novel enumerative source code proposed by Tanaka et. al. [150] produces a codeword for each length–n sequence of symbols. Each codeword consists of a prefix that represents the Lee–weight of the q–ary source sequence, and a suffix that represents the index of the source sequence in an ordered set of all sequences with Lee–weight and length equal to that of the source sequence.

The Lee–weight of a sequence is defined using a function $f : \mathcal{A} \rightarrow \{0, 1, \ldots |\mathcal{A}| - 1\}$, where $\mathcal{A}$ denotes the alphabet of the source symbols [150]. The function f maps each distinct alphabet symbol to a distinct integer in the specified range. The Lee–weight of a sequence is defined as the sum of the integers assigned to the symbols of the sequence.

The Lee–weight of the sequence $\mathbf{x}^n$ is expressed as

$$w_L(\mathbf{x}^n) = \sum_{i=1}^{n} f(x_i). \tag{4.42}$$

In order to derive an enumerative source code that uses the Lee–weight, Tanaka et. al. [150] considered a specific combinatorial problem. The problem involves the calculation of the number of sequences of length l, where $n \leq l \leq qn$, that can be constructed by concatenating n patterns of the set P, where $P = \{\mathbf{u}_i : 1 \leq i \leq q\}$, and $\mathbf{u}_i$ is the i–bit sequence $\{0, 0, 0, \ldots 0, 1\}$. The patterns of the set P may be used repeatedly during the construction of each sequence. The number of distinct sequences that may be constructed equals

$$N(n, l, q) = \sum_{j=0}^{\lfloor (l-n)/q \rfloor} (-1)^j \binom{n}{j} \binom{l-1-jq}{n-1}. \tag{4.43}$$

The integer $N_{n,k}$ may be interpreted as the total number of length–n q–ary symbol sequences with Lee–weights equal to k, where $N_{n,k} \triangleq N(n, n+k, q)$ [150]. These integers may be used to define a general version of Pascal's triangle. Row n of the general triangle contains $n(q-1)+1$ integers, where $n \geq 0$. Integer k of row n of the general version of Pascal's triangle is equal to the integer $N_{n,k}$, where $0 \leq k \leq n(q-1)$. The general version of Pascal's triangle is equal to the conventional triangle if q is set equal to two.

Tanaka et. al. [150] defined branch values between integers in consecutive rows of the general version of Pascal's triangle. The branch value between integers $N_{n,k}$ and $N_{n+1,k+j}$ is defined as

$$b_{n,k}^{(j)} = \sum_{h=0}^{j-1} N_{n,k+j-h}, \tag{4.44}$$

where $N_{n,k+j-h} \triangleq 0$ if $k+j-h < 0$ or $k+j-h > n(q-1)$.

Tanaka et. al. [150] proved that any length–n sequence of q–ary symbols with a Lee–weight of k may be expressed as a distinct path in the general version of Pascal's triangle. The transitions between the integers of the triangle depend on the symbols of the sequence. The sum of the branch values on the path associated with a sequence equals the index of the sequence in an ordered set of all n–symbol sequences with the same Lee–weight as the source sequence.

The encoder of the enumerative source code of Tanaka et. al. [150] encodes the Lee–weight of a source sequence as a q–ary number with $\lceil \log_q((q-1)n+1) \rceil$ digits. The index of the source sequence is encoded as a q–ary number with $\lceil \log_q(N_{n,w_L(\mathbf{x}^n)}) \rceil$ digits.

Tanaka et. al. [150] investigated the effectiveness of the proposed source code when used to encode finite–length sequences, as well as sequences with lengths that tend to infinity. The sequences consisted of i.i.d. symbols. It was proved that the proposed source code is weighted universal, but not minimax universal. The proposed source code has an average codeword length that is smaller than the average codeword length of the q–ary enumerative source code proposed by Schalkwijk [60], assuming that the source codes are used to encode short sequences. The enumerative source code of Schalkwijk

proved to be more effective than the proposed source code when used to encode longer source sequences.

4.1.10 The efficient enumerative source codes of Hertz et. al.

Hertz et. al. [151] proposed a technique for reducing the computational complexity of two enumerative source codes for sequences of q–ary symbols, where $q \geq 2$. The original enumerative codes were proposed by Schalkwijk [60] and Tanaka et. al. [150]. Implementations of both enumerative source codes rely on the addition of certain coefficients in order to calculate the index of the source sequence in an ordered set. In the case of Schalkwijk's enumerative code the coefficients are binomial or multinomial coefficients. The coefficients of the source code proposed by Tanaka et. al. may be computed using equation 4.43.

Hertz et. al. [151] proved that the number of coefficients that are added in order to obtain the codeword of a source sequence using the enumerative codes of Schalkwijk [60] and Tanaka et. al. [150] is equal to the Lee–weight of the source sequence[9]. The average Lee–weight of the source sequences may be reduced by mapping the more likely alphabet symbols to smaller integers (refer to equation 4.42), thereby reducing the computational complexity of the implementation. By specifying an appropriate permutation of the alphabet symbols, the source encoder may minimize the average Lee–weight of the source sequences, provided that the source symbol distribution is known.

The reduced–complexity encoder of Schalkwijk's [60] enumerative source code for q–ary symbols encodes a source sequence as follows [151]. The source encoder produces the same codeword prefix as before. It subsequently determines the appropriate permutation of the alphabet symbols — the symbols of this permutation are sorted in the order of nonincreasing frequency counts. This permutation is considered the new symbol alphabet, with the first symbol of the permutation being the lexicographically smallest symbol. The encoder uses the permuted symbol alphabet as it calculates the codeword suffix. By using the permuted symbol alphabet during the calculation of the codeword suffix, the computational complexity of the source encoder is reduced.

The source decoder that is paired with the reduced–complexity encoder decodes the codeword prefix as before [151]. It subsequently sorts the alphabet symbols in the order of nonincreasing symbol frequency counts, thereby obtaining the symbol permutation that was used by the source encoder. The source decoder proceeds by decoding the suffix of the codeword using the permuted symbol alphabet, after which it obtains the original source sequence by reversing the permutation of the alphabet symbols.

Hertz et. al. [151] stated that the computational complexity associated with the encoder and decoder of the enumerative source code proposed by Tanaka et. al. [150] may be reduced by using the same permutation of the alphabet symbols as used by the reduced–complexity encoder of the enumerative source code of Schalkwijk [60]. As the encoder of the source code proposed by Tanaka et. al. produces a codeword prefix that represents the Lee–weight of the source sequence, and not the frequency counts of

[9]It is assumed that the function f of equation 4.42 maps lexicographically smaller symbols to smaller integers by default.

the distinct alphabet symbols, the source encoder has to explicitly encode the alphabet symbol permutation in order to guarantee that the codewords are uniquely decodable.

The reduced–complexity source encoder of the enumerative source code proposed by Tanaka et. al. [150] calculates both the prefix and the suffix of each codeword using the permuted symbol alphabet [151]. As the source decoder obtains the symbol permutation directly from the source–coded sequence, it decodes each codeword using the permuted symbol alphabet. The original source sequence is obtained by reversing the permutation of the alphabet symbols.

Hertz et. al. [151] considered the effective source coding of the alphabet symbol permutation, as is required by the reduced–complexity encoder of the enumerative source code proposed by Tanaka et. al. [150]. The permutation may be encoded more effectively if the source sequence does not contain all distinct symbols of the alphabet. Hertz et. al. subsequently proved that the reduced–complexity encoder of the enumerative source code proposed by Tanaka et. al. [150] produces codewords that are shorter on average than the codewords of the original source encoder. This observation holds true if each source sequence is significantly longer than the number of distinct alphabet symbols.

Hertz et. al. [151] considered the implementation of the enumerative source code proposed by Tanaka et. al. [150]. It was concluded that the general version of Pascal's triangle may become prohibitively large to represent using a lookup table if long source sequences are to be encoded. Hertz et. al. proposed the use of chained lists to reduce the amount of memory that is required to represent the triangle. The implementation may use a recursive method for calculating the codeword of a source sequence if the memory requirement of the triangle remains excessive. This method may also be used in an implementation of the enumerative source code proposed by Schalkwijk [60].

4.2 Symbolwise enumerative source codes

The encoder of a symbolwise enumerative source code does not calculate the codeword of an entire source sequence in a single instance. It encodes a source sequence in a symbol–by–symbol manner, starting at the front of the sequence. The symbols of the sequence are encoded using an arithmetic encoder with an appropriate adaptive source model [8]. The length of the codeword that a symbolwise enumerative source encoder produces when used to encode a source sequence is identical to the length of the codeword that would be produced were a blockwise enumerative source encoder used.

Although Rissanen et. al. [157] and Rissanen [158] were the first to prove that arithmetic source coding and enumerative source coding are equivalent, Cleary et. al. [8] proved that enumerative source coding may be performed using an arithmetic source code. Enumerative source coding may be performed by deriving an appropriate adaptive source model that an arithmetic encoder uses as it encodes each source symbol. The model is derived using the frequency counts of the symbols in the source sequence. The symbol distribution of the model is a function of the symbols that were encoded previously, as well as the total frequency counts of the distinct alphabet symbols that appear in the source sequence.

Cleary et. al. [8] derived an expression for the symbol distribution of a source model

that may be used with an arithmetic encoder in order to perform enumerative source coding. The conditional distribution of the symbol X_i in the n–symbol source sequence $\mathbf{X}^n$ is expressed as

$$\Pr(X_i = y|\mathbf{x}_1^{i-1}) = \frac{C(\mathbf{x}_1^{i-1}y)}{C(\mathbf{x}_1^{i-1})}, \tag{4.45}$$

where $C(\mathbf{x}_1^j)$ equals the number of source sequences in the ordered set S_b with prefix equal to $\mathbf{x}_1^j$, and $1 \leq i \leq n$. It is assumed that $C(\mathbf{x}_1^0) = |S_b|$.

To illustrate the derivation of an appropriate symbol distribution, Cleary et. al. [8] considered a binary memoryless source, and defined the ordered set S_b as containing all sequences with the same length and weight as the source sequence. The distribution of bit X_i, conditioned on the sequence $\mathbf{x}_1^{i-1}$, was derived by observing that the number of source sequences in the set S_b with prefix $\mathbf{x}_1^{i-1}$ equals the number of valid ways in which the bits of the sequence $\mathbf{x}_i^n$ may be arranged. Cover [7] proved that

$$C(\mathbf{x}_1^{i-1}) = \frac{(n-i+1)!}{n_0(\mathbf{x}_i^n)!n_1(\mathbf{x}_i^n)!}, \tag{4.46}$$

where $n_0(\mathbf{x}^n)$ and $n_1(\mathbf{x}^n)$ equal the frequency counts of the zero–valued and nonzero–valued bits in the sequence $\mathbf{x}^n$, respectively. Substitution of this expression into equation 4.45 provides an expression for the conditional symbol distribution

$$\Pr(X_i = y|\mathbf{x}_1^{i-1}) = \frac{n_y(\mathbf{x}_i^n)}{n-i+1}, \tag{4.47}$$

where $y \in \{0, 1\}$.

Wirth [48] provided an expression for the symbol distribution of an enumerative source model under the assumption that the source produces i.i.d. q–ary symbols, where $q \geq 2$. The ordered set S_b was defined as containing all sequences with the same length and the same number of each distinct alphabet symbol as the source sequence. The expression

$$C(\mathbf{x}_1^{i-1}) = \frac{(n-i+1)!}{\prod_{y\in\mathcal{A}} n_y(\mathbf{x}_i^n)!} \tag{4.48}$$

may be evaluated to obtain the number of sequences with prefix $\mathbf{x}_1^{i-1}$ in the set S_b. By substituting this expression into equation 4.45, the conditional symbol distribution

$$\Pr(X_i = y|\mathbf{x}_1^{i-1}) = \frac{n_y(\mathbf{x}_i^n)}{n-i+1}, \tag{4.49}$$

where $y \in \mathcal{A}$, is obtained.

Cleary et. al. [8] found that the symbol distribution of the enumerative source model is significantly more complex to derive if the source symbols are statistically dependent on one another. The distribution of the enumerative source model that may be used to encode bits from a first–order binary Markov source[10] was derived in order to motivate this statement. An appropriate distribution may be derived by considering

[10] This source model is equivalent to the first–order binary Markov source as considered by Cover [7], and illustrated in figure B.1 on page 260.

the frequency counts of the distinct bit pairs in the source sequence, instead of the individual bits.

Assume that $n_{yy} \triangleq n_{yy}(\mathbf{x}_i^n)$ where $y \in \{0, 1\}$, and that $y' = 1 - y$. Cleary et. al. [8] proved that the conditional probability distribution associated with bit X_i may be expressed as

$$\Pr(X_i = y|\mathbf{x}_1^{i-1}) = \frac{n_{yy}}{n_{yy} + n_{y'y}} \tag{4.50}$$

and

$$\Pr(X_i = y'|\mathbf{x}_1^{i-1}) = \begin{cases} \frac{n_{y'y}}{n_{yy}+n_{y'y}} & \text{if } n_{yy} > 0, \\ 1 & \text{if } n_{yy} = 0, \end{cases} \tag{4.51}$$

where $x_{i-1} = y$ and $i \geq 2$. The expression

$$\Pr(X_1 = y) = \frac{n_{yy'}(n_{yy} + n_{y'y})}{n_{yy'}(n_{yy} + n_{y'y}) + n_{y'y}(n_{y'y'} + n_{yy'})}, \tag{4.52}$$

where $n_{yy} \triangleq n_{yy}(\mathbf{x}^n)$, is valid for the first bit of the sequence. A comparison between equation 4.47 and equations 4.50 to 4.52 reveals that the expression for the enumerative source model distribution is significantly more complex in the case of a first–order Markov source, compared to a source that produces i.i.d. bits. The derivation of enumerative source models for more sophisticated sources becomes exceedingly complex. This trend appears to limit the practicality of source codes that rely on exact enumeration.

Cleary et. al. [8] made use of conditioning classes to manage the complexity involved in deriving symbol distributions of source models. A conditioning class may be interpreted as a distinct preceding context of one or more symbols in the source sequence. The conditional probability distribution of a source symbol X_i is solely a function of the frequency counts of the symbols that follow the same conditioning class as the symbol X_i. Only two conditioning classes are defined for source bits from a first–order binary Markov source. One of these conditioning classes corresponds to a zero–valued preceding bit, and the other to a nonzero–valued preceding bit.

The symbolwise enumerative source code discussed up to this point uses conditional symbol distributions for exact enumeration. Cleary et. al. [8] proposed two symbolwise enumerative source codes that perform approximate enumeration. The source encoder of an approximate enumerative source code encodes each symbol of the source sequence using an adaptive source model with a symbol distribution that approximates the symbol distribution for exact enumeration. The motivation behind the use of an approximate symbol distribution is to avoid the complexity inherent in the derivation of the exact symbol distribution.

The first approximate enumerative source code approximates the number of ways in which the distinct symbols that follow each conditioning class may be arranged [8]. Let the total number of times that symbol y appears after a certain conditioning class z in the source sequence $\mathbf{x}^n$ be denoted by $n_{zy}(\mathbf{x}^n)$, and define

$$n_z(\mathbf{x}^n) = \sum_{y \in \mathcal{A}} n_{zy}(\mathbf{x}^n). \tag{4.53}$$

The symbols that follow a certain conditioning class z in the source sequence may be arranged in a maximum of

$$\frac{n_z(\mathbf{x}^n)!}{\prod_{y\in\mathcal{A}} n_{zy}(\mathbf{x}^n)!} \tag{4.54}$$

possible ways. The first approximate enumerative source code approximates the number of sequences with prefix $\mathbf{x}_1^{i-1}$ in the set S_b as the product of several integers, where each integer equals the number of possible ways of arranging the symbols that follow a certain conditioning class. The approximation is expressed as

$$C(\mathbf{x}_1^{i-1}) = \prod_{z\in\mathcal{Z}} \frac{n_z(\mathbf{x}_i^n)!}{\prod_{y\in\mathcal{A}} n_{zy}(\mathbf{x}_i^n)!}. \tag{4.55}$$

Equation 4.55 is an approximation of the true number of sequences in the set S_b that share the prefix $\mathbf{x}_1^{i-1}$, as some of the combinations that are included in the count correspond to invalid sequences (i.e. sequences in which invalid transitions occur between certain conditioning classes) [8]. This approximation is typically an overestimation of the actual number of sequences that share the specific prefix (and which are elements of the set S_b). The probability distribution of the symbol X_i, conditioned on the prefix $\mathbf{x}_1^{i-1}$ of the source sequence and the number of symbols that occur in each conditioning class, may be obtained by substituting equation 4.55 into equation 4.45.

The second approximate enumerative source code proposed by Cleary et. al. [8] uses what is referred to as the fixed frequency model. The fixed frequency model is derived using the frequency counts of the symbols that appear in each conditioning class of the source sequence. The conditional probability distribution of the symbol X_i, which occurs in conditioning class z, is defined as

$$\Pr(X_i = y|\mathbf{x}_1^{i-1}) = \frac{n_{zy}(\mathbf{x}^n)}{n_z(\mathbf{x}^n)}. \tag{4.56}$$

This approximation is less accurate than the first approximation, as it completely disregards any symbols that may have appeared previously in the source sequence.

Cleary et. al. [8] distinguished between parameterized enumerative source codes and adaptive source codes. The source encoder of a parameterized enumerative source code encodes the model parameters (i.e. the frequency counts of the symbols that occur in each distinct conditioning class) as part of the source–coded sequence. It may only encode the symbol counts upon having access to the entire interval of the source sequence over which the symbol frequency counts are to be collected, however. Adaptive source codes estimate the model parameters in an adaptive fashion using those source symbols encoded previously. These codes may be applied in situations where the distribution of the source sequence is unknown apriori.

Cleary et. al. [8] proposed methods A and B for estimating the symbol distribution according to the symbol frequency counts of each conditioning class in an adaptive manner. These methods were discussed in section 2.5.3 on page 19. Cleary et. al. found that adaptive source codes that use methods A and B were more effective than both approximate enumerative source codes when used to encode certain practical data.

CHAPTER 5

Efficient computation of binomial coefficients

The enumerative source code implementations proposed in chapter 6 of this thesis have to compute large binomial coefficients in order to encode source sequences and decode their codewords. The average magnitude of the binomial coefficients that have to be computed is directly proportional to the length of the source sequences. Universal enumerative source codes are ideally used to encode longer source sequences, as the normalized average redundancy of this type of code is inversely proportional to the length of the source sequences. The efficient computation of large binomial coefficients is a prerequisite of universal enumerative source code implementations that are practical.

Three approaches to computing large binomial coefficients are summarized in this chapter. The first and second approaches are impractical, and are included in the summary to illustrate the degree of complexity involved in the computation of large binomial coefficients. The third approach involves the decomposition of large numbers into their prime factors, and is more practical than the alternative approaches. This approach was used in all the enumerative source code implementations proposed in chapter 6.

5.1 Approach 1: Direct computation

Consider the computation of the binomial coefficient

$$\begin{aligned} B_{m,k} &= \binom{m}{k} \\ &= \frac{m!}{k!(m-k)!}, \end{aligned} \tag{5.1}$$

where m and k are nonnegative integers, and $k \leq m$. The multiplicative terms common to the numerator and denominator of equation 5.1 may be cancelled prior to the computation of the binomial coefficient. The number of calculations that have to be performed in order to obtain the value of the binomial coefficient may therefore be reduced by first computing the integers

$$d_x = \max(k, m-k) \tag{5.2}$$

and

$$d_n = \min(k, m-k), \tag{5.3}$$

and subsequently computing the binomial coefficient using the expression

$$B_{m,k} = \frac{\prod_{i=1}^{m-d_x}(d_x + i)}{d_n!}. \tag{5.4}$$

The factorial $d_n!$ has to be computed in order to evaluate equation 5.4. The factorial is computed by multiplying a running product with a succession of integers. The jth step of the computation may be expressed as

$$p_j = j(p_{j-1}), \tag{5.5}$$

where

$$p_j = j! \tag{5.6}$$

is the running product after the jth step of the computation, with $1 \leq j \leq d_n$ and $p_0 \triangleq 1$.

A lower bound on the number of bit operations involved in the computation of a factorial with the running–product approach is subsequently derived. This derivation is carried out under the assumption that the number of bit operations involved in the multiplication of two binary numbers is lower bounded by the length of the product (expressed in bits). This lower bound is reasonable, as a processor has to assign the correct values to the bits of the registers that represent the product, in addition to any other bit operations.

The running product p_j of equation 5.6 equals the factorial $j!$. A lower bound on the number of bits needed to represent a factorial is therefore required to derive a lower bound on the number of bit operations involved in the computation of a factorial. The number of bits that are required to represent the factorial $j!$ may be lower bounded using Stirling's approximation of the factorial, which is expressed as [44]

$$j! = \sqrt{2\pi j}\left(\frac{j}{e}\right)^j e^{\lambda_j}, \tag{5.7}$$

where

$$\frac{1}{12j+1} < \lambda_j < \frac{1}{12j} \tag{5.8}$$

and $j > 0$. The lower bound on the number of bits that are required to represent the factorial $j!$ is derived as

$$\begin{aligned} \lceil \log_2(j!+1) \rceil &\geq \lceil \log_2(j!) \rceil \\ &> \frac{1}{2}\log_2(2\pi j) + j\log_2\left(\frac{j}{e}\right) + \frac{1}{12j+1}\log_2(e), \end{aligned} \tag{5.9}$$

where $j > 0$. The lower bound of equation 5.9 is used to derive a lower bound on the total number of bit operations involved in the computation of the factorial $j!$ with the running–product approach. This lower bound is expressed as

$$\begin{aligned} C_{\mathrm{D}}(j) &\geq \sum_{i=1}^{j} \lceil \log_2(i!+1) \rceil \\ &> \sum_{i=1}^{j} \left[\frac{1}{2}\log_2(2\pi i) + i\log_2\left(\frac{i}{e}\right) + \frac{1}{12i+1}\log_2(e)\right], \end{aligned} \tag{5.10}$$

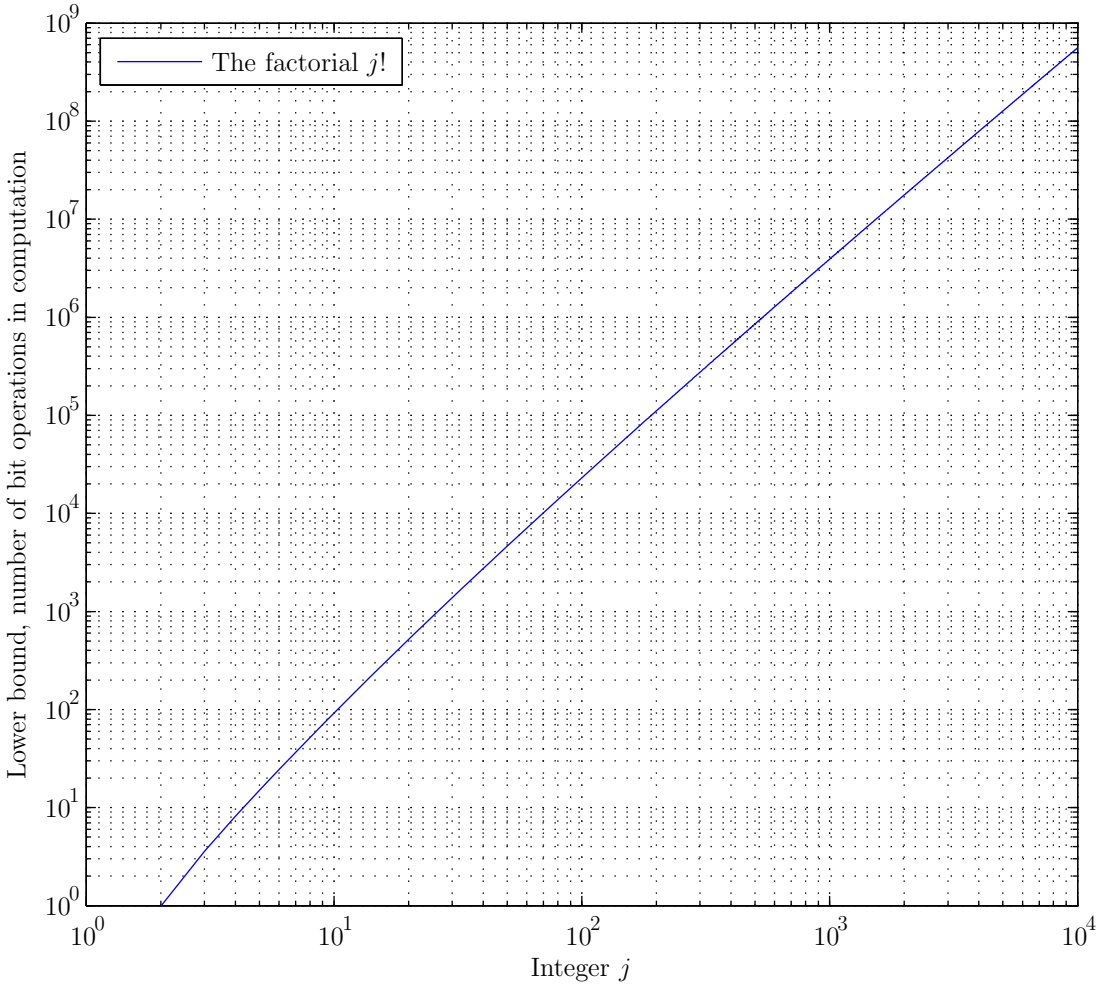


Figure 5.1: A lower bound on the number of bit operations involved in the computation of the factorial $j!$ with the running–product approach, as a function of the integer j.

where $j > 0$.

The lower bound on the number of bit operations involved in the computation of the factorial $j!$ with the running–product approach is plotted in figure 5.1 as a function of the integer j. The figure reveals that the number of bit operations increases rapidly with respect to the integer j. It requires more than 10^8 bit operations to compute the factorial 5000!. Considering the fact that an enumerative source code implementation might be required to compute this factorial in order to encode a single source sequence of 10^4 bits, it is evident that a more efficient approach to computing binomial coefficients is required.

A lower bound on the number of bit operations required to compute the binomial coefficient $B_{m,k}$ may be derived by using equation 5.10 as a lower bound on the number of bit operations involved in the computation of the factorial $d_n!$ of equation 5.4. The lower bound on the number of bit operations that are required to compute the binomial coefficient $B_{m,k}$, with $m = 10^4$, is plotted in figure 5.2 on page 133 as a function of the integer k. The figure reveals that an excessive number of bit operations are required to

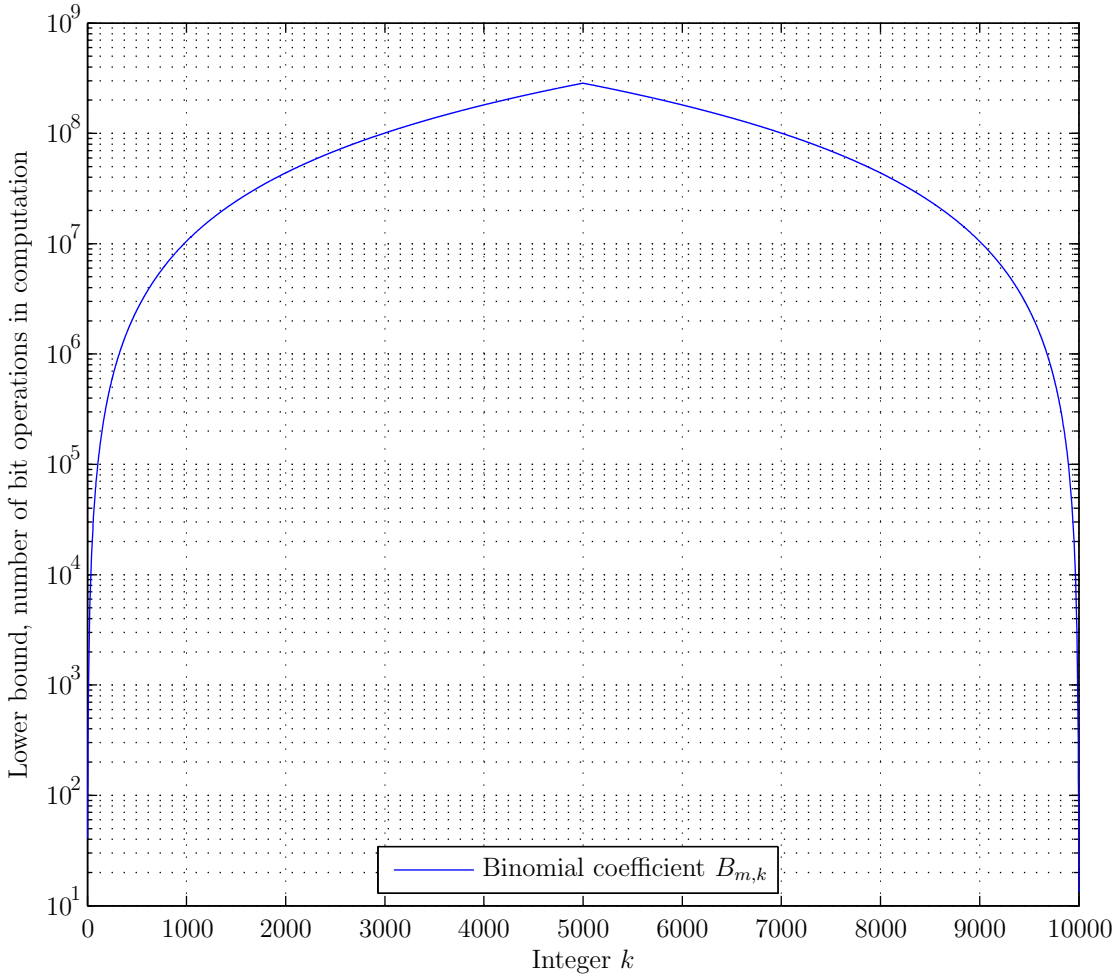


Figure 5.2: A lower bound on the number of bit operations required to compute the binomial coefficient $B_{m,k}$, as a function of the integer k. The case where $m = 10^4$ is considered.

compute the majority of binomial coefficients $B_{m,k}$, where $m = 10^4$. It requires more than 10^6 bit operations to compute any binomial coefficient $B_{m,k}$ with $m = 10^4$ and $350 \leq k \leq 9650$. As an enumerative source code implementation typically computes multiple binomial coefficients to encode a single source sequence, it is clear that the direct approach to calculating binomial coefficients is not suitable for enumerative source code implementations.

5.2 Approach 2: Computation using a lookup table

Instead of directly computing each binomial coefficient, an enumerative source code implementation may use a precomputed lookup table to reduce the number of calculations that are required to encode each source sequence. The lookup table may either contain the factorials of several integers, or it may contain several binomial coefficients. The use of both types of lookup table is investigated in this section.

5.2.1 Lookup table of factorials

Consider an enumerative source code implementation that obtains the factorial of an integer from a precomputed lookup table, instead of directly computing the factorial. In order to compute a certain binomial coefficient $B_{m,k}$, it first obtains the factorials $m!$, $k!$ and $(m-k)!$ from the lookup table. It subsequently computes the coefficient using equation 5.1.

In order to determine whether an enumerative source code implementation that uses a lookup table of factorials is feasible, a lower bound on the number of bits required to represent the lookup table is derived. The derivation does not take into account any overhead associated with delimiting the factorials of the table. The derivation uses the lower bound of equation 5.9 for the number of bits required to represent the factorial of an integer.

In order to successfully encode any source sequence with a length of n bits, the enumerative source code implementation requires a lookup table of the factorials $\{0!, 1!, 2!, \ldots n!\}$. The size of this lookup table is lower bounded according to the expression

$$\begin{aligned} C_{\text{LF}}(n) &\geq \sum_{i=0}^{n} \lceil \log_2(i!+1) \rceil \\ &> \sum_{i=1}^{n} \left[\frac{1}{2}\log_2(2\pi i) + i\log_2\left(\frac{i}{e}\right) + \frac{1}{12i+1}\log_2(e) \right], \end{aligned} \tag{5.11}$$

where $n > 0$.

The lower bound of equation 5.11 is plotted in figure 5.3 on page 135 as a function of the source sequence length n. The bound on the size of the lookup table is expressed in megabytes. The figure reveals that the size of the lookup table increases rapidly with respect to the source sequence length n. The enumerative source code implementation requires a lookup table that is larger than one megabyte in order to encode source sequences longer than 1419 bits.

Table 5.1 on page 136 contains the lengths of the longest source sequences, as a function of the lookup table size, that the enumerative source code implementation can successfully encode. Due to the rapid increase in the size of the lookup table, the implementation requires a lookup table larger than 66.3 megabytes to successfully encode all source sequences up to 10^4 bits long. The use of a lookup table that contains factorials is clearly an impractical approach to the computation of binomial coefficients.

5.2.2 Lookup table of binomial coefficients

Instead of obtaining factorials from a lookup table, and using the factorials to compute binomial coefficients, an enumerative source code implementation may instead obtain the binomial coefficients directly from a precomputed lookup table. This implementation encodes a source sequence by adding certain binomial coefficients that it obtains from the lookup table. This enumerative source code implementation performs significantly fewer calculations to encode a source sequence than an implementation that computes the binomial coefficients.

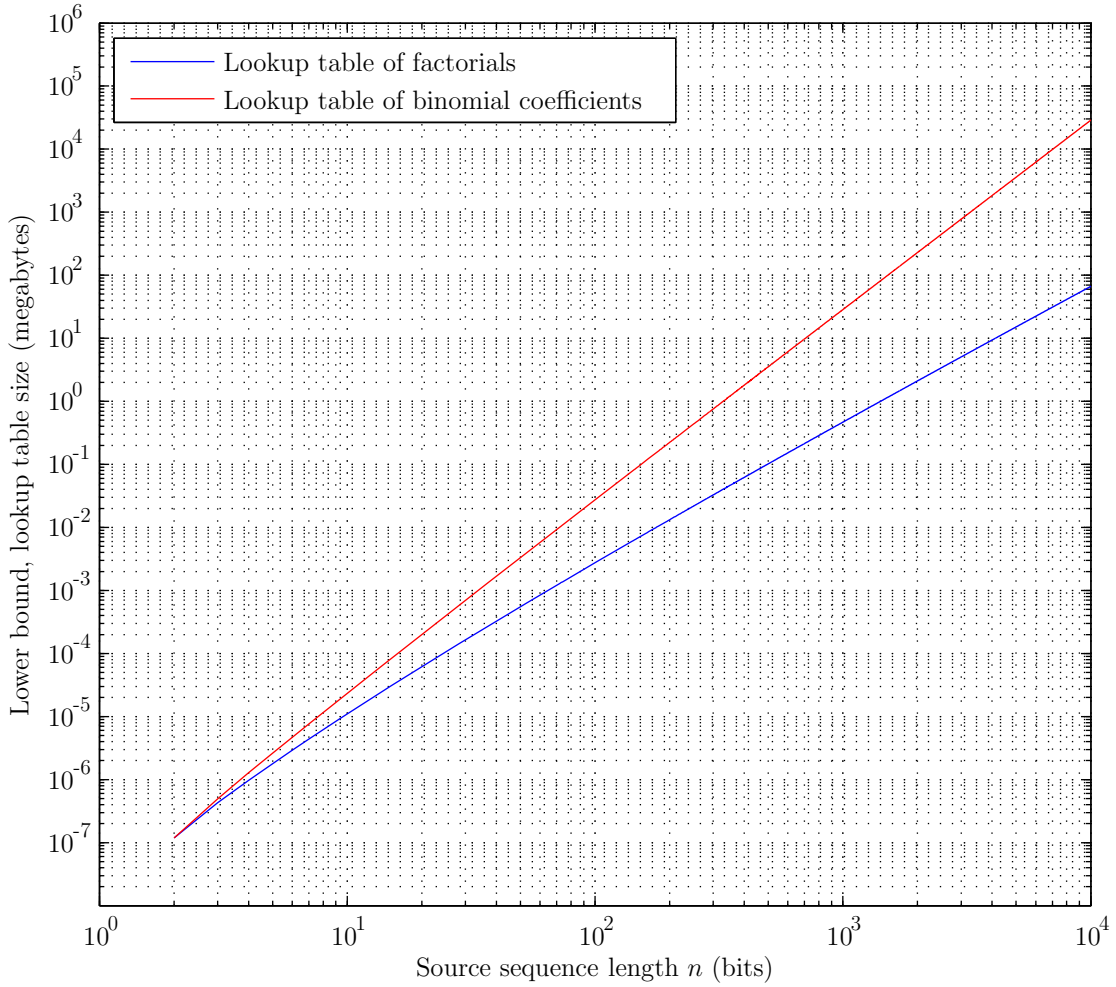


Figure 5.3: Lower bounds on the sizes of the lookup tables of factorials and binomial coefficients, as a function of the source sequence length.

In order to successfully encode any n–bit source sequence, the enumerative source code implementation requires the binomial coefficients $B_{m,k}$, where $0 \leq k \leq m \leq n$. A lower bound on the number of bits that are required to represent the binomial coefficient $B_{m,k}$ may be derived by using Stirling's approximation of the factorial $j!$, as provided in equations 5.7 and 5.8 [44]. The lower bound is derived as

$$\begin{aligned} \lceil \log_2(B_{m,k}+1) \rceil &\geq \left\lceil \log_2\left(\frac{m!}{k!(m-k)!}\right)\right\rceil \\ &> \frac{1}{2}\log_2\left(\frac{m}{2\pi k(m-k)}\right) + m\log_2(m) - k\log_2(k) \\ &\quad - (m-k)\log_2(m-k) + \left(\frac{1}{12m+1} - \frac{1}{12k} - \frac{1}{12(m-k)}\right)\log_2(e), \end{aligned} \tag{5.12}$$

where $0 < k < m$ and $m > 1$.

Table 5.1: The maximum length of source sequences that can be successfully encoded using factorial lookup tables of various sizes.

Lookup table size	Source sequence length
1 kB	63 bits
10 kB	174 bits
100 kB	490 bits
1 MB	1 419 bits
10 MB	4 126 bits
66.32 MB	10 000 bits

A lower bound on the size of the lookup table may be obtained by substituting the lower bound of equation 5.12 into the expression

$$\begin{aligned} C_{\mathrm{LB}}(n) &\geq \sum_{m=0}^{n}\sum_{k=0}^{m} \lceil \log_2(B_{m,k}+1) \rceil \\ &> \sum_{m=2}^{n}\sum_{k=1}^{m-1} \lceil \log_2(B_{m,k}+1) \rceil \,, \end{aligned} \tag{5.13}$$

where $n > 1$. This expression does not take into account any overhead associated with delimiting the binomial coefficients of the table.

The lower bound on the size of the lookup table that the enumerative source code implementation requires to successfully encode all n–bit source sequences is plotted as a function of the integer n in figure 5.3 on page 135. The figure reveals that the lookup table of binomial coefficients that is required to encode all n–bit sequences is consistently larger than the lookup table of factorials that is required to encode sequences of the same length. Lookup tables of binomial coefficients are considered impractical, as the smaller lookup tables of factorials are considered impractical.

5.3 Approach 3: Prime factor decomposition

The third approach to computing large binomial coefficients was used in all practical implementations of the enumerative source codes proposed in chapter 6 of this thesis. This approach may be used to encode source sequences of more than 10^4 bits with an acceptably low degree of computational complexity[1], and without lookup tables that are excessively large. The same approach may be used to realize a computationally–efficient enumerative source decoder.

The third approach relies on the prime factor decomposition of integers larger than zero. Let the total number of primes smaller than or equal to the integer j be denoted

[1]The computational complexity is deemed sufficiently low for simulating the enumerative source codes on a laptop PC, and in a reasonable amount of time. The simulations involve the source coding of source sequences, the decoding of the codewords, and the measurement of the average codeword length.

by $\theta(j)$, where $j > 0$. The integer j may be decomposed into its prime factors as

$$\begin{aligned} j &= p_1^{c_1(j)} p_2^{c_2(j)} \dots p_{\theta(j)}^{c_{\theta(j)}(j)} \\ &= \prod_{i=1}^{\theta(j)} p_i^{c_i(j)}, \end{aligned} \tag{5.14}$$

where p_i is the ith prime number, and $c_i(j)$ is the exponent of the prime number p_i in the prime factor decomposition of the integer j.

The factorial $j!$ may be expressed as the product of its prime factors, but the number of primes in the product increases very rapidly w.r.t. the integer j. An alternative expression for the factorial $j!$ may be derived by observing that the factorial $j!$ equals the product of the integers from unity to j, and that the exponents of identical prime numbers in the prime factor decompositions of these integers may be added together. The factorial $j!$ may therefore be expressed as

$$\begin{aligned} j! &= p_1^{\sum_{q=1}^{j} c_1(q)} p_2^{\sum_{q=1}^{j} c_2(q)} \dots p_{\theta(j)}^{\sum_{q=1}^{j} c_{\theta(j)}(q)} \\ &= \prod_{i=1}^{\theta(j)} p_i^{\sum_{q=1}^{j} c_i(q)}. \end{aligned} \tag{5.15}$$

Equation 5.15 is used to derive an alternative expression for the binomial coefficient $B_{m,k}$, where $0 \leq k \leq m \leq n$. This expression is derived as

$$\begin{aligned} B_{m,k} &= \frac{m!}{k!(m-k)!} \\ &= p_1^{a_1} p_2^{a_2} \dots p_{\theta(m)}^{a_{\theta(m)}} \\ &= \prod_{i=1}^{\theta(m)} p_i^{a_i}, \end{aligned} \tag{5.16}$$

where

$$\begin{aligned} a_i &\triangleq a_i(m,k) \\ &= \sum_{q=1}^{m} c_i(q) - \sum_{q=1}^{k} c_i(q) - \sum_{q=1}^{m-k} c_i(q). \end{aligned} \tag{5.17}$$

The evaluation of the alternative expression for the binomial coefficient $B_{m,k}$ involves several integer additions and subtractions. These integers correspond to the prime factor exponents of the integers from unity to m.

An enumerative source code implementation that needs to decompose several integers into their prime factors in order to encode each source sequence is computationally inefficient. In order to avoid the computational burden of decomposing an integer into its prime factors, an enumerative source code implementation may use two precomputed lookup tables. The first table contains all of the $\theta(n)$ prime numbers smaller than or equal to the integer n, where n is the length of the longest source sequence

that the implementation is required to encode. The second lookup table contains the integers

$$v_{m,i} = \sum_{q=1}^{m} c_i(q) \tag{5.18}$$

for all $m \in \{1, 2, \ldots n\}$ and all $i \in \{1, 2, \ldots \theta(n)\}$. The integer $v_{m,i}$ equals the exponent of the prime number p_i in the expression for the factorial $m!$ (refer to equation 5.15).

In order to compute a binomial coefficient $B_{m,k}$, an enumerative source code implementation obtains the integers $v_{m,i}$, $v_{k,i}$ and $v_{m-k,i}$, for all $i \in \{1, 2, \ldots \theta(m)\}$, from the second lookup table. It computes the exponent of each prime number in the expression for the binomial coefficient (equation 5.16) as

$$a_i(m, k) = v_{m,i} - v_{k,i} - v_{m-k,i}. \tag{5.19}$$

It subsequently obtains the prime numbers of the set $\{p_1, p_2, \ldots p_{\theta(m)}\}$ from the first lookup table, and evaluates equation 5.16 to obtain the binomial coefficient $B_{m,k}$.

A mathematical analysis of the computational complexity of the third approach to computing large binomial coefficients is beyond the scope of this thesis. The computational complexity of the third approach was instead determined by measuring the average time that a practical enumerative source code implementation requires to encode several source sequences and decode their codewords (when using the third approach). These results are presented in section 6.1.2.6 on page 165.

The sizes of the first and second lookup tables of the third approach to computing large binomial coefficients were determined by generating the two tables. An enumerative source code implementation was able to successfully encode any source sequence of up to 10^4 bits (and decode its codeword) with these tables. A total of 5 948 bytes were required to represent the first table, while the second table occupied a total of 125 488 bytes. The total size of both tables equaled 131 436 bytes, which is not an excessive number of bytes.

Figure 5.3 on page 135 reveals that the implementation which uses a lookup table of factorials is able to encode source sequences of up to approximately 550 bits if it uses a lookup table of the same size as the tables of the practical source code implementation (i.e. the third approach). The figure of 550 bits is 18 times smaller than the maximum length of the source sequences that may be encoded using the practical implementation.

CHAPTER 6

Mathematical analysis and practical performance

This chapter contains a mathematical analysis of several enumerative source codes, some of which are novel. It contains mathematical definitions of the source encoder and decoder of each code. Proofs regarding the unique decodability of the source codes are also presented in this chapter.

The effectiveness of each source code is investigated both theoretically and empirically. The summary of each enumerative source code includes a theoretical bound on the effectiveness of the code. The source codes were implemented and used to encode sequences from abstract information sources in order to compare their performance.

6.1 Single–field enumerative source codes

Each codeword of a single–field enumerative source code contains only a single binary–coded integer, unlike the codewords of more sophisticated enumerative source codes. These enumerative source codes are introduced in order to gain insight into the performance and characteristics of the more sophisticated enumerative source codes.

Some of the single–field enumerative source codes serve as building blocks for multi–field enumerative source codes. The single–field enumerative source codes may be used to effectively encode bit sequences from simple abstract sources, however.

6.1.1 Constant weight, fixed–to–fixed length code

The constant weight, fixed–to–fixed length code is a single–field enumerative source code that is used to encode sequences from a specific information source. This abstract information source is referred to as the fixed–weight binary source, and is introduced in what follows.

6.1.1.1 The fixed–weight binary source

The fixed–weight binary source is characterized by two integer parameters. These parameters are denoted by n and w, where $n > 0$ and $0 \leq w \leq n$. The source produces a sequence of n bits in a single instance, and each sequence contains exactly w nonzero–valued bits.

The fixed–weight binary source with parameters n and w can produce a total of

$$\binom{n}{w} = \frac{n!}{w!(n-w)!} \tag{6.1}$$

distinct n–bit sequences with w nonzero–valued bits. All of the distinct source sequences are equiprobable, and all source sequences are statistically independent from one another. The per–sequence entropy of the source equals

$$H(\mathbf{X}^n) = \log_2 \binom{n}{w} \tag{6.2}$$

bits.

6.1.1.2 Derivation of the source code

Several terms are introduced prior to the derivation of the constant weight, fixed–to–fixed length source code. These definitions concern the relationship between bit sequences and binary numbers, and are used throughout the remainder of this thesis.

A source sequence of bits is considered equivalent to a binary number. Each bit of an n–bit source sequence $\mathbf{x}^n = \{x_1, x_2, \ldots x_n\}$ has a certain level of significance, as is the case with the bits of a binary number. For the purpose of this thesis, the trailing bit of a sequence (x_n) is considered the least significant bit, and the leading bit (x_1) is considered the most significant bit.

Each bit in a source sequence has an integer degree that is associated with it. The degree of a bit may be interpreted as its level of significance. The smallest degree equals unity, and is associated with the least significant bit of a sequence. The largest degree equals n, where n is the length of the source sequence. This degree is associated with the most significant bit of a sequence. The degree of bit x_i of an n–bit source sequence $\mathbf{x}^n$ equals $n - i + 1$.

The degrees of the w nonzero–valued bits of the source sequence $\mathbf{x}^n$ are assigned to the vector $\mathbf{d}_\mathbf{x}$, where $\mathbf{d}_\mathbf{x} = \{d_{1,\mathbf{x}}, d_{2,\mathbf{x}}, \ldots d_{w,\mathbf{x}}\}$. The elements of the vector are arranged in ascending order (i.e. $d_{i,\mathbf{x}} > d_{k,\mathbf{x}}$ for all $i > k$).

The constant weight, fixed–to–fixed length source code is derived by considering an n–bit source sequence $\mathbf{x}^n$ of weight w, as produced by the fixed–weight binary source with parameters n and w. It is assumed that $n > 1$ and $1 < w \leq n$ in order to complete the derivation. The most significant nonzero–valued bit in the source sequence $\mathbf{x}^n$ has a degree equal to $d_{w,\mathbf{x}}$. All source sequences with no nonzero–valued bits of degree $d_{w,\mathbf{x}}$ or larger are numerically smaller than the source sequence $\mathbf{x}^n$, regardless of the remaining bits of the sequence. There exists a total of

$$z_w = \binom{d_{w,\mathbf{x}} - 1}{w} \tag{6.3}$$

distinct source sequences that satisfy this requirement.

Certain n-bit source sequences of weight w have a nonzero–valued bit with a degree of $d_{w,\mathbf{x}}$, but no nonzero–valued bits with a larger degree. Some of these sequences may be numerically smaller than the source sequence $\mathbf{x}^n$. The total number of these

numerically smaller sequences may be established by considering the bits of $\mathbf{x}^n$ that have degrees smaller than $d_{w,\mathbf{x}}$. Observe that no sequence having any nonzero–valued bits with degree j, where $d_{w-1,\mathbf{x}} < j < d_{w,\mathbf{x}}$, as well as a nonzero–valued bit with degree $d_{w,\mathbf{x}}$, is numerically smaller than the source sequence $\mathbf{x}^n$. It is therefore sufficient to consider only those source sequence bits with degrees smaller than or equal to $d_{w-1,\mathbf{x}}$ in order to determine how many of the remaining sequences are numerically smaller than $\mathbf{x}^n$.

Let the function $Q(\mathbf{x}^n, d_{w,\mathbf{x}})$ represent the number of n–bit source sequences of weight w that are found to be numerically smaller than the sequence $\mathbf{x}^n$ by only considering those source sequence bits with a degree smaller than or equal to $d_{w,\mathbf{x}}$. The recursive expression

$$Q(\mathbf{x}^n, d_{w,\mathbf{x}}) = z_w + Q(\mathbf{x}^n, d_{w-1,\mathbf{x}}), \tag{6.4}$$

where $Q(\mathbf{x}^n, d_{0,\mathbf{x}}) \triangleq 0$, may be derived from the previous statements. This expression may be simplified as

$$\begin{aligned} Q(\mathbf{x}^n, d_{w,\mathbf{x}}) &= \sum_{i=1}^{w} z_i \\ &= \sum_{i=1}^{w} \binom{d_{i,\mathbf{x}} - 1}{i}, \end{aligned} \tag{6.5}$$

where $\binom{a}{b} \triangleq 0$ for all $a < b$. The sum of equation 6.5 equals the total number of distinct n–bit source sequences, each having a weight of w bits, that are numerically smaller than $\mathbf{x}^n$. This equation is equivalent to equation 4.10 on page 107, which is an expression for the integer codeword of an index sequence, as produced by the enumerative source code of Lynch [5, 6].

The binary number that corresponds to the integer $Q(\mathbf{x}^n, d_{w,\mathbf{x}})$ is padded with zero–valued bits on the side of its most significant bit to obtain a k–bit binary number. The k–bit binary number is the codeword of the source sequence $\mathbf{x}^n$. If the original, unpadded binary number is longer than k bits, the source sequence that corresponds to the binary number cannot be assigned a unique codeword. This source sequence is assigned a codeword of k zero–valued bits. This codeword is incorrectly decoded by the source decoder.

6.1.1.3 Definition of the source code

The constant weight, fixed–to–fixed length source code is formally defined in terms of mathematical functions as follows. Let $f : \mathbb{B}^n \mapsto Z$ where $\mathbb{B} = \{0, 1\}$ and $Z = \{z \in \mathbb{N}_0 : 0 \leq z \leq \binom{n}{w} - 1\}$. The function f maps an n–bit source sequence with weight w to its index in the numerically–ordered set of all n–bit sequences with weight w. The function f may be decomposed as

$$f = f_2 \circ f_1. \tag{6.6}$$

The trivial function $f_1 : \mathbb{B}^n \mapsto D$, where $D = \{\mathbf{d} \in \mathbb{N}^w : 1 \leq d_i \leq n \wedge d_i > d_j \, \forall \, i > j\}$, or $\mathbf{d}_\mathbf{x} = f_1(\mathbf{x})$, is defined over all source sequences $\mathbf{x}^n$. The elements of the vector

$\mathbf{d} \equiv \mathbf{d}_{\mathbf{x}}$ equal the degrees of the w nonzero–valued bits of $\mathbf{x}^n$ in ascending order. The function $f_2 : D \mapsto Z$, or $z = f_2(\mathbf{d})$, maps the degree vector associated with the source sequence to the integer index of the source sequence in the numerically–ordered set of all n–bit source sequences with weight w. It is defined as

$$f_2(\mathbf{d}) = \sum_{i=1}^{w} \binom{d_i - 1}{i}, \tag{6.7}$$

where $\binom{a}{b} \triangleq 0$ for all $a < b$. Let the function $\beta_m : \mathbb{N}_0 \mapsto \mathbb{B}^m$ be defined as

$$\beta_m(y) = \begin{cases} B_m(y) & \text{if } 0 \leq y \leq 2^m - 1, \\ \mathbf{0}^m & \text{if } y > 2^m - 1, \end{cases} \tag{6.8}$$

where $\mathbf{0}^m$ denotes the m–bit sequence that consists of zero–valued bits. The function $B_m(y)$ maps the integers in its domain of $0 \leq y \leq 2^m - 1$ to their respective binary numbers, which are zero–padded to m bits on the side of the most significant bit as necessary. The source encoder of the constant weight, fixed–to–fixed length source code is defined using the function $g_1 : \mathbb{B}^n \mapsto \mathbb{B}^k$. The codeword of the sequence $\mathbf{x}^n$ is expressed as

$$\begin{aligned} \mathbf{c}_{\mathbf{x}} &= g_1(\mathbf{x}) \\ &= \beta_k\big(f(\mathbf{x})\big). \end{aligned} \tag{6.9}$$

6.1.1.4 Proof of unique decodability

This section contains a proof regarding the unique decodability of the constant weight, fixed–to–fixed length (n–bit to k–bit) source code, as used to encode sequences from a fixed–weight binary source with parameters n and w. It is proved that this source code is uniquely decodable if the condition $k \geq \lceil \log_2 \binom{n}{w} \rceil$ is satisfied.

If the parameter w of the fixed–weight binary source equals zero or n, the source produces only one distinct sequence, and the proof regarding the unique decodability of the source code is trivial. The proof that is presented in this section is carried out under the assumption that $0 < w < n$, where $n > 1$.

In order to prove that the source code is uniquely decodable, it is first proved that the function $f = f_2 \circ f_1$ is invertible. The first step of this proof is to establish that the function f_2 is invertible. The invertibility of the function f_2 is established by proving that the function f_2 is both injective and surjective.

Consider two elements x and y in the domain of a function h. The function h is injective if $h(x) \neq h(y)$ for all x and y with $x \neq y$. A surjective function h has the property that, for every element y in the codomain of h, there exists some element x in its domain that satisfies the equation $h(x) = y$.

Lemma 6.1.1. *The function f_2 is injective.*

Proof. In order to prove that the function f_2 is injective, it is sufficient to prove that

$$\sum_{v=1}^{w} \binom{j_v - 1}{v} \neq \sum_{k=1}^{w} \binom{h_k - 1}{k} \tag{6.10}$$

under the assumption that there exists one or more elements j_i with $j_i \neq h_i$, and where $\mathbf{j}, \mathbf{h} \in D$. Let

$$r = \max(i) : j_i \neq h_i \tag{6.11}$$

and without loss of generality assume that $j_r > h_r$. The remainder of the proof is divided into two sections. The first section addresses the case where $h_r \leq r$, and the second section addresses the case where $h_r > r$.

In order to prove that equation 6.10 is valid under the assumption that $h_r \leq r$, it is sufficient to consider the case where $h_r = r$, as $h_r \geq r$ by definition. If $h_r = r$, the equation

$$\sum_{k=1}^{r} \binom{h_k - 1}{k} = 0 \tag{6.12}$$

holds, as $h_i = i$ for $1 \leq i \leq r$. Due to the assumption that $j_r > h_r$, the inequality

$$\binom{j_r - 1}{r} > 0 \tag{6.13}$$

holds. It follows that

$$\sum_{v=1}^{r} \binom{j_v - 1}{v} > \sum_{v=1}^{r} \binom{h_v - 1}{v}, \tag{6.14}$$

as each binomial coefficient is nonnegative. This inequality is used to prove that

$$\begin{aligned}
\sum_{v=1}^{w} \binom{j_v - 1}{v} &= \sum_{v=1}^{r} \binom{j_v - 1}{v} + \sum_{v=r+1}^{w} \binom{j_v - 1}{v} \\
&= \sum_{v=1}^{r} \binom{j_v - 1}{v} + \sum_{v=r+1}^{w} \binom{h_v - 1}{v} \\
&> \sum_{v=1}^{r} \binom{h_v - 1}{v} + \sum_{v=r+1}^{w} \binom{h_v - 1}{v} \\
&> \sum_{v=1}^{w} \binom{h_v - 1}{v}.
\end{aligned} \tag{6.15}$$

The proof concerning the case where $h_r \leq r$ is therefore complete.

The remainder of the proof addresses the case where $h_r > r$. The equation

$$\sum_{k=1}^{r} \binom{h_k - 1}{k} = \sum_{v=0}^{r-1} \binom{h_{r-v} - 1}{r - v} \tag{6.16}$$

is obtained by setting v equal to $r - k$. The inequality

$$h_{r-v} \leq h_r - v, \tag{6.17}$$

where $v \geq 0$, is substituted into equation 6.16 to obtain the inequality

$$\begin{aligned}\sum_{k=1}^{r}\binom{h_k-1}{k} &\leq \sum_{v=0}^{r-1}\binom{h_r-v-1}{r-v} \\ &\leq \sum_{v=0}^{r}\binom{h_r-v-1}{r-v},\end{aligned} \tag{6.18}$$

where the last step follows from the fact that all binomial coefficients are greater than or equal to zero. This inequality may be rewritten as

$$\sum_{k=1}^{r}\binom{h_k-1}{k} < \sum_{v=0}^{r}\binom{h_r-v-1}{r-v} \tag{6.19}$$

due to the fact that $\binom{h_r-r-1}{0} = 1$ under the assumption that $h_r \geq r+1$. The identity

$$\sum_{i=0}^{r}\binom{n-i}{r-i} = \binom{n+1}{r} \tag{6.20}$$

is substituted into equation 6.19 to obtain the inequality

$$\begin{aligned}\sum_{k=1}^{r}\binom{h_k-1}{k} &< \binom{h_r}{r} \\ &< \binom{j_r-1}{r},\end{aligned} \tag{6.21}$$

where the last step follows from the fact that $j_r > h_r$. The left–hand side of equation 6.10 may therefore be simplified as

$$\begin{aligned}\sum_{v=1}^{w}\binom{j_v-1}{v} &= \sum_{v=1}^{r}\binom{j_v-1}{v} + \sum_{v=r+1}^{w}\binom{j_v-1}{v} \\ &= \sum_{v=1}^{r}\binom{j_v-1}{v} + \sum_{v=r+1}^{w}\binom{h_v-1}{v} \\ &> \sum_{v=1}^{r}\binom{h_v-1}{v} + \sum_{v=r+1}^{w}\binom{h_v-1}{v} \\ &> \sum_{v=1}^{w}\binom{h_v-1}{v},\end{aligned} \tag{6.22}$$

thereby completing the proof. □

Lemma 6.1.2. *The function f_2 is surjective.*

Proof. The largest and smallest integers in the range of function f_2 are determined in order to complete the proof. The smallest integer in the range of function f_2 is

associated with the degree vector $\mathbf{d} = \{1, 2, \ldots w\}$. The smallest integer equals

$$\begin{aligned} f_2(\mathbf{d}) &= \sum_{i=1}^{w} \binom{d_i - 1}{i} \\ &= \sum_{i=1}^{w} \binom{i-1}{i} \\ &= 0, \end{aligned} \tag{6.23}$$

as $\binom{n}{k} \triangleq 0$ for all $n < k$.

The largest integer in the range of function f_2 is associated with the degree vector $\mathbf{d} = \{n - w + 1, n - w + 2, \ldots n\}$. The largest integer equals

$$\begin{aligned} f_2(\mathbf{d}) &= \sum_{i=1}^{w} \binom{d_i - 1}{i} \\ &= \sum_{j=0}^{w-1} \binom{n-j-1}{w-j} \\ &= \binom{n}{w} - 1, \end{aligned} \tag{6.24}$$

where the first step involves the substitution $j = w - i$, and the last step follows from equation 6.20 and the fact that

$$\binom{n-w-1}{0} = 1 \tag{6.25}$$

for all $w < n$ (which corresponds to the nontrivial proof).

The largest and smallest elements in the range of function f_2, as well as the fact that the range of the function contains integer elements, imply that the function has a maximum of $\binom{n}{w}$ distinct elements in its range. The domain of function f_2 contains exactly $\binom{n}{w}$ distinct elements. The fact that function f_2 is injective implies that the function contains exactly $\binom{n}{w}$ elements in its range. The range of function f_2 is therefore equal to its codomain.

The fact that the range and codomain of function f_2 are equal implies that there exists, for each element z in the codomain of function f_2, some element $\mathbf{d}$ in the domain of function f_2 so that $f_2(\mathbf{d}) = z$. The function f_2 is therefore surjective. □

Theorem 6.1.1. *The function f_2 is invertible.*

Proof. Function f_2 is invertible as it is both injective and surjective (refer to lemmas 6.1.1 and 6.1.2). □

Theorem 6.1.2. *The constant weight, fixed–to–fixed length (n–bit to k–bit) source code, when used to encode sequences from a fixed–weight binary source with parameters n and w, is uniquely decodable if $k \geq \lceil \log_2 \binom{n}{w} \rceil$.*

Proof. The source code is proved as being uniquely decodable by proving that function g_1, as defined in equation 6.9, is invertible if $k \geq \left\lceil \log_2 \binom{n}{w} \right\rceil$. It is therefore assumed that $k \geq \left\lceil \log_2 \binom{n}{w} \right\rceil$, from which it follows that

$$\begin{aligned} 2^k &\geq 2^{\left\lceil \log_2 \binom{n}{w} \right\rceil} \\ &\geq 2^{\log_2 \binom{n}{w}} \\ &\geq \binom{n}{w} \end{aligned} \tag{6.26}$$

and that

$$\binom{n}{w} - 1 \leq 2^k - 1. \tag{6.27}$$

The left–hand side of equation 6.27 equals the largest integer that function f associates with any element in its restricted domain[1], which consists of all source sequences from a fixed–weight binary source with parameters n and w. As the maximum value of function f is smaller than or equal to $2^k - 1$, function β_k in equation 6.9 may be replaced by function B_k from equation 6.8 to obtain the equation

$$\begin{aligned} \mathbf{c_x} &= g_1(\mathbf{x}) \\ &= B_k\big(f(\mathbf{x})\big). \end{aligned} \tag{6.28}$$

Function f_1 is invertible provided that the integer n is known. It follows that function f is invertible, as both functions f_1 and f_2 are invertible (refer to theorem 6.1.1 for a proof regarding the invertibility of function f_2). The inverse of function f is expressed as

$$f^{-1} = f_1^{-1} \circ f_2^{-1}. \tag{6.29}$$

It follows that function g_1 is invertible, as both functions f and B_k are invertible. The inverse of function g_1 is derived as

$$\begin{aligned} \mathbf{x} &= g_1^{-1}(\mathbf{c_x}) \\ &= f^{-1}\big(B_k^{-1}(\mathbf{c_x})\big). \end{aligned} \tag{6.30}$$

□

6.1.1.5 Theoretical performance

The theoretical performance of the constant weight, fixed–to–fixed length (n–bit to k–bit) source code is investigated in this section. It is assumed that the source code is used to encode sequences from a fixed–weight binary source with parameters n and w. The performance of the source code is characterized in terms of its block–error probability, as well as the redundancy of its codewords.

The block–error probability of a source code is defined by considering a random source sequence $\mathbf{X}^n$. Suppose that the source code is used to encode this sequence,

[1] This statement follows from equation 6.6 and the maximum value of function f_2, as determined in lemma 6.1.2.

thereby obtaining its codeword $\mathbf{C_x}$. The source decoder is subsequently applied to the codeword $\mathbf{C_x}$. Let the output of the source decoder be denoted by $\hat{\mathbf{X}}^n$. A block error is said to occur if $\hat{\mathbf{X}}^n \neq \mathbf{X}^n$. The block–error probability is defined as

$$P_b \triangleq \Pr(\hat{\mathbf{X}}^n \neq \mathbf{X}^n). \tag{6.31}$$

The block–error probability of the constant weight, fixed–to–fixed length source code is derived in what follows. Theorem 6.1.2 states that the source code is uniquely decodable if the codeword length k satisfies the inequality $k \geq \left\lceil \log_2 \binom{n}{w} \right\rceil$. The block–error probability P_b of the source code is therefore equal to zero if $k \geq \left\lceil \log_2 \binom{n}{w} \right\rceil$.

If the codeword length k satisfies the inequality $k \leq \left\lceil \log_2 \binom{n}{w} \right\rceil - 1$, the block–error probability of the source code is larger than zero. The number of source sequences that are correctly recovered by the source decoder equals 2^k, as each codeword is associated with at least one source sequence. The remaining $\binom{n}{w} - 2^k$ source sequences are not correctly decoded from the codewords assigned to them. The block–error probability of the source code equals the fraction of source sequences that are incorrectly recovered, as all source sequences are equiprobable. The block–error probability of the source code may therefore be expressed as

$$P_b = \begin{cases} 0 & \text{if} \quad k \geq \left\lceil \log_2 \binom{n}{w} \right\rceil, \\ 1 - 2^k \binom{n}{w}^{-1} & \text{if} \quad k \leq \left\lceil \log_2 \binom{n}{w} \right\rceil - 1. \end{cases} \tag{6.32}$$

The redundancy of the constant weight, fixed–to–fixed length source code with a block–error probability of zero is considered in what follows. In order to derive a bound on the redundancy of the code, the codeword length is selected as

$$k = \left\lceil \log_2 \binom{n}{w} \right\rceil \tag{6.33}$$

bits, as this is the minimum codeword length that guarantees a uniquely decodable code.

The average per–codeword redundancy of the constant weight, fixed–to–fixed length source code is defined as the difference between the average codeword length and the per–sequence entropy of the source. As the codewords of the constant weight, fixed–to–fixed length source code are of equal length k, the average per–codeword redundancy of the source code, measured in bits, may be bounded as

$$\begin{aligned} R(\mathbf{X}^n) &= k - H(\mathbf{X}^n) \\ &= \left\lceil \log_2 \binom{n}{w} \right\rceil - \log_2 \binom{n}{w} \\ &< 1, \end{aligned} \tag{6.34}$$

where the expression for the per–sequence entropy was substituted from equation 6.2.

6.1.1.6 Practical results

This section presents the block–error probability and redundancy associated with a practical implementation of the constant weight, fixed–to–fixed length code. The results were obtained by using the implementation to encode sequences from the fixed–weight binary source in two configurations. The performance of the implementation

was compared to the performance of a Huffman code, an arithmetic code and an ideal variable–length code in each case.

The source codes that were compared to the constant weight, fixed–to–fixed length source code are summarized in the following section. This section is followed by a discussion of each configuration of the source code implementation, as well as the presentation of the source coding results. The results are discussed, and the source codes are compared to one another.

Alternative source codes An n–bit extended Huffman code was considered for the source coding of sequences from the fixed–weight binary source. The Huffman code was not implemented in software, as the exact length of each codeword of the extended Huffman code may be derived mathematically. This derivation is subsequently carried out.

The extended Huffman code associates its codewords with a total of $\binom{n}{w}$ leaf nodes of a full binary tree. The codewords occupy (at most) two consecutive levels of the binary tree, due to the fact that the source sequences are equiprobable. Assume that the tree levels are numbered according to their depth (relative to the root node), and that the root node is located on level zero of the tree. The codewords of the Huffman code are located on the levels numbered

$$l_1 = \left\lceil \log_2 \binom{n}{w} \right\rceil \tag{6.35}$$

and

$$l_2 = l_1 - 1. \tag{6.36}$$

Let the number of codewords located on level number l_1 of the tree be denoted by m_1, and the number of codewords located on level number l_2 of the tree be denoted by m_2. The number of codewords on each of these levels may be derived using the equation

$$m_1 + m_2 = \binom{n}{w}, \tag{6.37}$$

as well as Kraft's inequality [16], which is expressed as

$$m_1 2^{-l_1} + m_2 2^{-l_2} \leq 1. \tag{6.38}$$

Kraft's inequality holds with equality for the Huffman code under consideration, as the binary tree containing the codewords is full. The inequality may be manipulated to obtain the equation

$$\begin{aligned} m_1 + 2m_2 &= 2^{l_1} \\ &= 2^{\left\lceil \log_2 \binom{n}{w} \right\rceil}. \end{aligned} \tag{6.39}$$

By substituting equation 6.37 into equation 6.39, the number of codewords of length l_2 is derived as

$$m_2 = 2^{\left\lceil \log_2 \binom{n}{w} \right\rceil} - \binom{n}{w}. \tag{6.40}$$

Table 6.1: Parameters and quantities associated with the first configuration of the constant weight, fixed–to–fixed length code.

Parameter / Quantity	Symbol	Value	Unit
Source sequence length	n	500	bits
Source sequence weight	w	125	bits
Codeword length	k	1 – 500	bits
Trials per codeword length	—	10 000	trials
Per–sequence source entropy	$H(\mathbf{X}^n)$	401.04	bits

By substituting equation 6.40 into equation 6.37, the number of codewords of length l_1 is derived as

$$m_1 = 2\binom{n}{w} - 2^{\lceil \log_2 \binom{n}{w} \rceil}. \tag{6.41}$$

The arithmetic source code that was compared to the constant weight, fixed–to–fixed length code was not implemented. A bound on the average length of the codewords that an arithmetic code produces was used in the comparison. This bound is expressed as [4]

$$H(\mathbf{X}^n) \leq L(\mathbf{X}^n) < H(\mathbf{X}^n) + 2, \tag{6.42}$$

where $L(\mathbf{X}^n)$ is the average length of the codewords produced by the encoder. Equation 2.6 on page 15, which expresses the length of each codeword in terms of the probability of occurrence of the corresponding source sequence, was also used in the comparison.

The ideal variable–length source code is a hypothetical source code that was compared to the constant weight, fixed–to–fixed length code. An ideal variable–length source code produces a codeword of length $-\log_2(\Pr(\mathbf{x}^n))$ bits when used to encode a source sequence $\mathbf{x}^n$ that occurs with a probability of $\Pr(\mathbf{x}^n)$. The ideal variable–length source code has an average codeword length that equals the per–sequence entropy of the source.

Configuration 1: Variable codeword length The first configuration involved the selection of fixed values for n (the length of each source sequence) and w (the weight of each source sequence). The length k of the codewords that the constant weight, fixed–to–fixed length code produced was varied in order to investigate the block–error probability associated with each source code[2]. The parameters of the first configuration are presented in table 6.1.

The source coding results associated with the constant–weight source code in the first configuration are presented in figure 6.1 on page 150. This figure contains a plot of the block–error probabilities of the source codes as a function of the codeword length of the constant weight, fixed–to–fixed length source code. The constant weight, fixed–to–fixed length source code is referred to as the proposed code in the discussion of the results.

[2]A block–error probability was associated with each variable–length lossless source code by interpreting all codewords with lengths exceeding the value of k as containing an error. This method of comparing fixed–length and variable–length source codes was used by Caire et. al. [85].

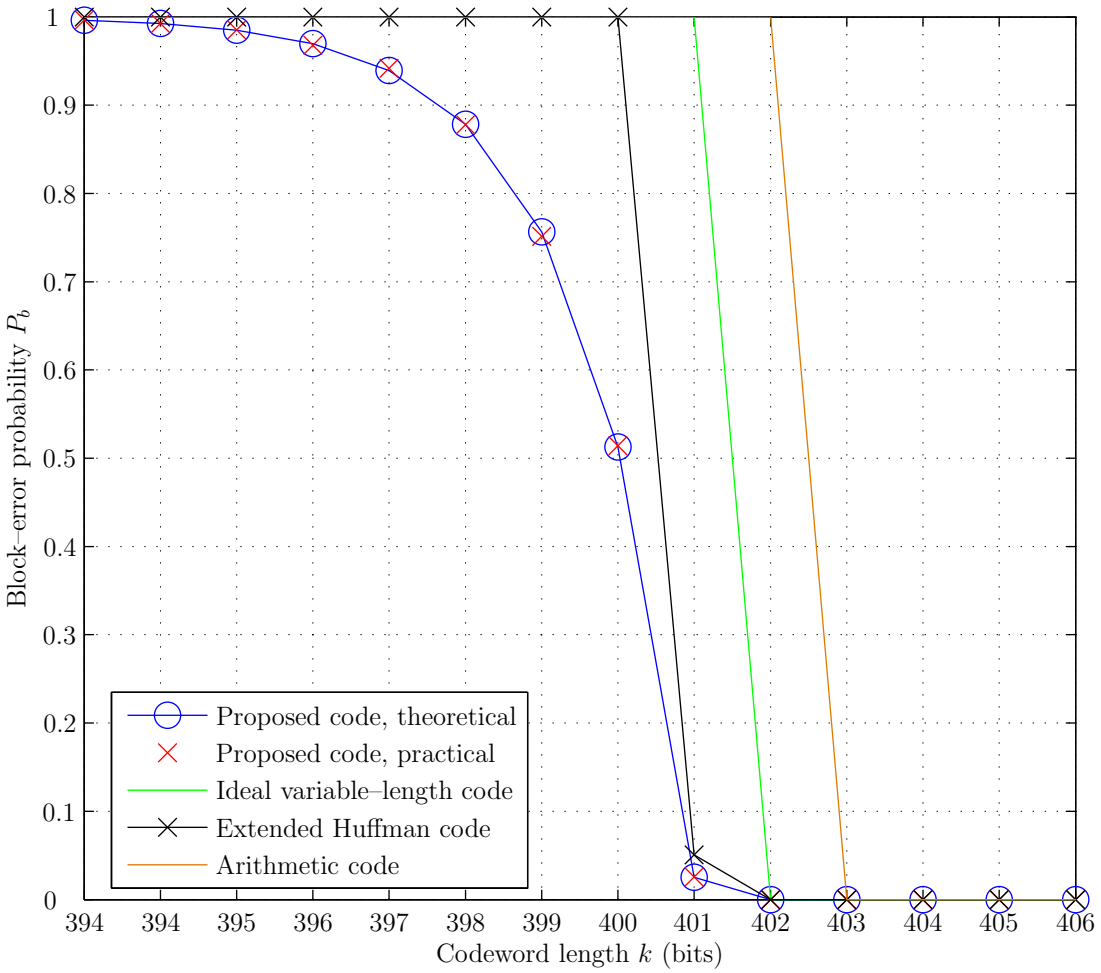


Figure 6.1: Block–error probabilities of various source codes as a function of the codeword length of the constant weight, fixed–to–fixed length code in the first configuration.

Figure 6.1 reveals that the theoretical block–error probability curve of the proposed code (equation 6.32) and the block–error probability curve of the practical implementation overlap. This observation implies that the proposed source code was correctly implemented.

The block–error probabilities of the source codes are nonincreasing functions of the codeword length k. Shannon proved that no fixed–to–fixed length source code can encode and decode source sequences with a block–error probability that is arbitrarily close to zero if the rate of the code is lower than the entropy of the source, regardless of the length of the source sequences [44]. The per–sequence entropy of the fixed–weight source, with parameters as provided in table 6.1, was calculated as 401.04 bits using equation 6.2. Figure 6.1 confirms that the block–error probabilities of all the source codes are significantly larger than zero upon considering codewords shorter than 401 bits[3].

[3]Shannon's theorem, as presented in this section, refers to the code rate of the source code. The curves of figure 6.1 are plotted as functions of the codeword length and not the code rate. The

A comparison of the curves of figure 6.1 reveals that the variable–length source codes (the Huffman code, arithmetic code and ideal variable–length code) do not perform as well as the proposed source code. This observation is consistent with the observations of Caire et. al. [85] regarding the suboptimal performance of variable–length source codes that are applied in fixed–length source coding scenarios. The variable–length source codes are designed to produce codewords with a minimum average length. The minimization of the average codeword length is not equivalent to the goal of minimizing the block–error probability, which applies to the design of fixed–length source codes.

Figure 6.1 reveals that the extended Huffman source code has a lower block–error probability than the ideal variable–length source code, assuming a codeword length of 401 bits. This observation may be justified by considering the lengths of the codewords of the ideal variable–length code. A codeword of the ideal variable–length source code has a length of $-\log_2(\Pr(\mathbf{x}^n))$ bits, where $\Pr(\mathbf{x}^n)$ denotes the corresponding source sequence's probability of occurrence. As all source sequences are equiprobable, the codewords all have the same length, which is equal to the per–sequence entropy of the source. This fact implies that all codewords of the ideal variable–length source code will be associated with block errors, as the per–sequence source entropy exceeds 401 bits. The Huffman code cannot represent codewords with a fractional number of bits, however. A fraction of

$$\frac{m_2}{m_1 + m_2} \approx 0.9495 \tag{6.43}$$

of the codewords of the extended Huffman code have a length of 401 bits, and the remaining codewords have a length of 402 bits. The block–error probability of the Huffman source code is therefore lower than the block–error probability of the ideal variable–length source code if k equals 401 bits.

The final observation regarding figure 6.1 concerns the arithmetic code. The curve associated with the arithmetic code was computed using equation 2.6 on page 15. According to this equation, the length of each codeword of the arithmetic code equals

$$\begin{aligned} l_A &= \lceil -\log_2(\Pr(\mathbf{x}^n)) \rceil + 1 \\ &= 403 \end{aligned} \tag{6.44}$$

bits. It follows that the arithmetic code is outperformed by the extended Huffman code in terms of block–error probability, as its codewords are longer than the codewords of the extended Huffman code.

Configuration 2: Variable source sequence weight The second configuration of the constant weight, fixed–to–fixed length code involved source sequences of a fixed length n. The weight of the source sequences from the fixed–weight binary source was varied in order to investigate the average codeword length and redundancy of each source code. The length of the codewords of the constant weight, fixed–to–fixed length source code was specified to guarantee its unique decodability. The appropriate codeword length for each value of w was obtained by evaluating equation 6.33 with each value of w. The parameters of the second configuration of the proposed source code are presented in table 6.2 on page 152.

conclusions drawn with respect to Shannon's theorem still hold, however.

Table 6.2: Parameters and quantities associated with the second configuration of the constant weight, fixed–to–fixed length code.

Parameter / Quantity	Symbol	Value	Unit
Source sequence length	n	500	bits
Source sequence weight	w	0 – 500	bits
Codeword length	k	Eq. 6.33	bits
Trial runs per source sequence weight	—	10 000	trials
Per–sequence source entropy	$H(\mathbf{X}^n)$	Eq. 6.2	bits

The length of the codewords produced by the implementation of the proposed source code is plotted as a function of the source sequence weight in figure 6.2 on page 153. This figure includes a plot of the per–sequence source entropy as a function of the source sequence weight. The curve of the per–sequence entropy of the fixed–weight binary source resembles the curve of the binary entropy function. The relationship between the two curves may be established by substituting Stirling's approximation of the factorial $n!$ [44] into the equation for the per–sequence entropy, thereby obtaining the approximation

$$\begin{aligned} H(\mathbf{X}^n) &= \log_2 \binom{n}{w} \\ &\approx nh\left(\frac{w}{n}\right) \end{aligned} \tag{6.45}$$

to the per–sequence entropy of the fixed–weight source with parameters n and w, where $h(\cdot)$ denotes the binary entropy function. The accuracy of the approximation improves w.r.t. the source sequence length n.

Figure 6.2 suggests that the codeword length of the proposed source code is approximately equal to the per–sequence entropy of the fixed–weight binary source. The difference between the codeword length of the proposed source code and the per–sequence entropy of the source (i.e. the redundancy) is investigated in what follows.

The average per–codeword redundancy $R(\mathbf{X}^n)$ of the constant weight, fixed–to–fixed length source code is plotted in figure 6.3 on page 154. The average redundancy of the practical implementation of the proposed source code is upper bounded by the blue curve, which represents the bound of equation 6.34. The statement that the average per–codeword redundancy of the proposed source code does not exceed the per–sequence source entropy by more than one bit is therefore verified.

A comparison of the redundancy of the proposed source code and the redundancy of the extended Huffman code reveals that the Huffman code is significantly less redundant than the proposed code. This observation may be justified by considering the fact that the proposed source code is a fixed–length source code, and that the length of all its codewords is greater than or equal to the per–sequence entropy of the source. The variable–length Huffman code is not constrained in this fashion — several of the codewords of the Huffman code typically contain fewer bits than the per–sequence entropy of the source. The average codeword length of the Huffman code can therefore

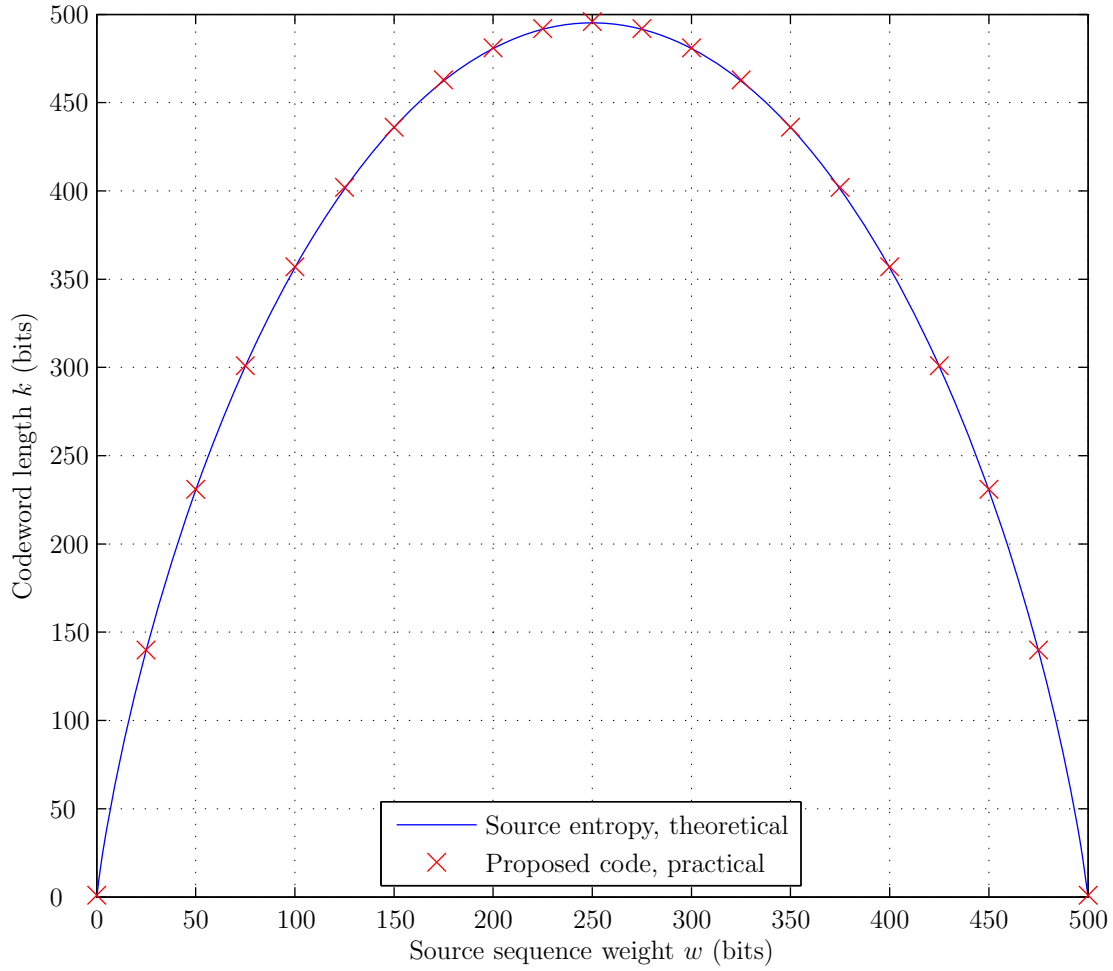


Figure 6.2: Per–sequence source entropy and codeword length of the constant weight, fixed–to–fixed length code in the second configuration, as a function of the source sequence weight.

approach the per–sequence source entropy more closely than the codeword length of the proposed source code, yet still exceed the source entropy.

The curve of the average per–codeword redundancy of the arithmetic code was computed using equation 6.42. This curve is a bound on the performance of the arithmetic code, and should not serve as an absolute measure of its performance. It merely proves that the average per–codeword redundancy of the arithmetic code does not exceed two bits.

6.1.2 Variable weight, fixed–to–fixed length code

The variable weight, fixed–to–fixed length (n–bit to k–bit) source code is a single–field enumerative source code that may be considered a generalization of the constant weight, fixed–to–fixed length source code. It may be used to encode n–bit source sequences of any weight w. The binary memoryless source is an example of an abstract source that produces sequences which may be effectively encoded using the variable weight, fixed–

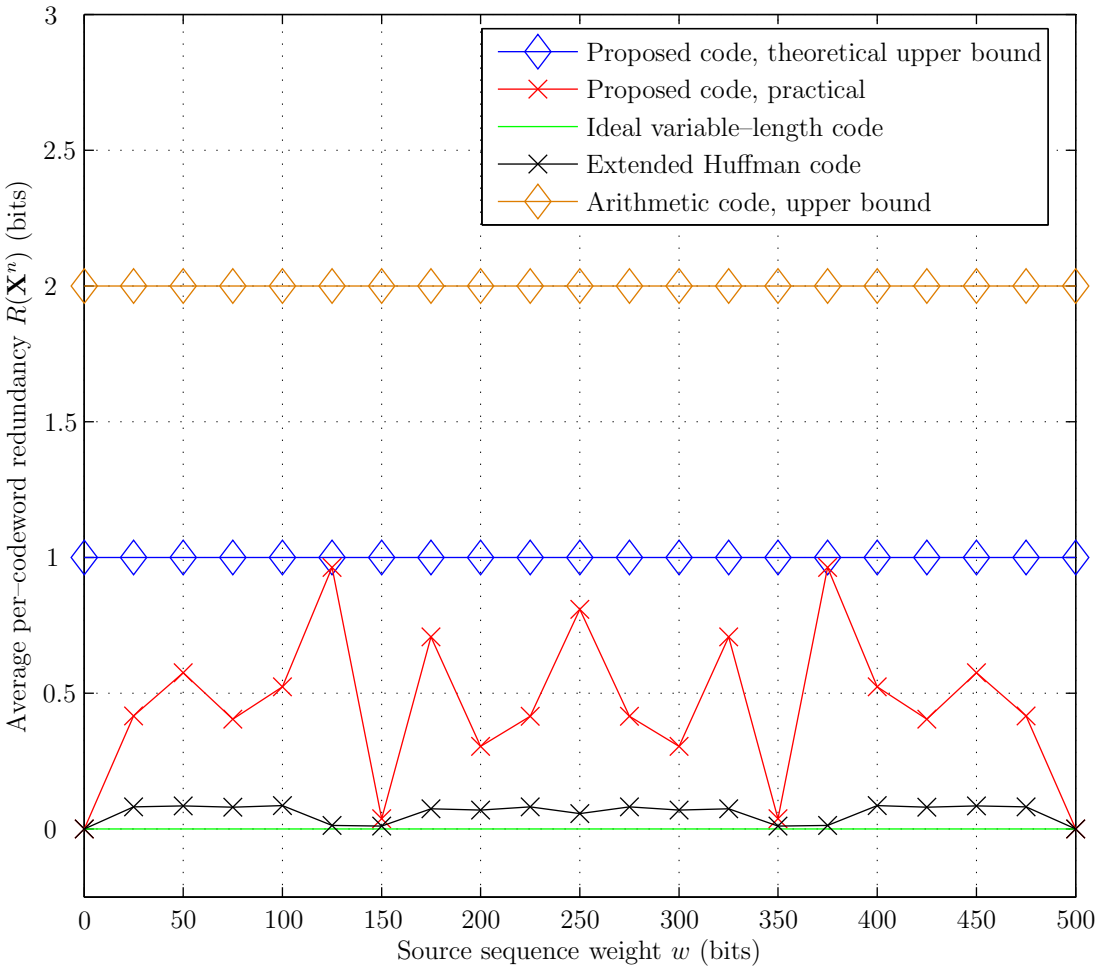


Figure 6.3: Average per–codeword redundancies of certain variable–length codes, as well as the constant weight, fixed–to–fixed length source code in the second configuration.

to–fixed length source code.

6.1.2.1 Derivation of the source code

Consider a set S_b that consists of all n–bit sequences, and assume that this set is ordered according to two rules. The first rule states that all sequences with weight i precede all sequences with weight j in the ordered set, for all values of i and j with $i < j$. The second rule states that equal–weight sequences in the set are ordered relative to one another by interpreting these sequences as binary numbers, and ordering the numbers numerically in ascending order. The second rule produces a sequence order that is identical to the order of the equal–weight sequences of the constant weight, fixed–to–fixed length source code.

The index of a source sequence $\mathbf{x}^n$ in the ordered set S_b equals the number of sequences that precede it in the set. Consider an n–bit source sequence $\mathbf{x}^n$ with a weight of w bits. The number of sequences with weight w that precede the sequence $\mathbf{x}^n$ in the ordered set S_b may be calculated using equation 6.5. The remaining sequences

that precede the sequence $\mathbf{x}^n$ in the ordered set consist of all distinct n–bit sequences with weights $0 \leq w' < w$. As a total of $\binom{n}{w'}$ distinct n–bit sequences of weight w' exist, the total number of distinct n–bit sequences with weights $0 \leq w' < w$ may be derived as

$$t = \sum_{w'=0}^{w-1} \binom{n}{w'}. \tag{6.46}$$

The index of the source sequence $\mathbf{x}^n$ in the ordered set S_b is obtained by adding the total number of distinct n–bit sequences with weights $0 \leq w' < w$ and the number of n–bit sequences with weight w that are numerically smaller than the sequence $\mathbf{x}^n$, where w is the weight of the source sequence.

The integer index of the source sequence $\mathbf{x}^n$ in the ordered set S_b is encoded using the conventional binary–coded representation of an integer. The binary number is padded to a length of k bits with zero–valued bits on the side of its most significant bit (if necessary). If the binary–coded representation of a certain index is longer than k bits, the source sequence that corresponds to the index cannot be assigned a unique codeword. This source sequence is assigned a codeword that consists of k zero–valued bits. This codeword is incorrectly decoded by the source decoder.

6.1.2.2 Definition of the source code

The mathematical definition of the variable weight, fixed–to–fixed length source code uses several of the functions that were introduced in section 6.1.1.3. Let the source encoder of the variable weight, fixed–to–fixed length source code correspond to the function $g_2 : \mathbb{B}^n \mapsto \mathbb{B}^k$. The codeword produced by the source encoder, when used to encode the source sequence $\mathbf{x}^n$, is expressed as

$$\begin{aligned} \mathbf{c_x} &= g_2(\mathbf{x}) \\ &= \beta_k\left(f(\mathbf{x}) + \sum_{w'=0}^{w-1} \binom{n}{w'}\right), \end{aligned} \tag{6.47}$$

where w is the weight of the source sequence $\mathbf{x}^n$, and k is the length of the codeword.

6.1.2.3 Proof of unique decodability

This section contains a proof of the fact that the variable weight, fixed–to–fixed length source code is uniquely decodable when used to encode source sequences from a certain set. It is shown that function g_2 of equation 6.47 is invertible if its domain is restricted to the sequences of the set.

Consider the function $f_3 : \mathbb{B}^n \mapsto I_n$, which is defined as

$$f_3(\mathbf{x}) = f(\mathbf{x}) + \sum_{w'=0}^{w-1} \binom{n}{w'}, \tag{6.48}$$

where w is the weight of the source sequence $\mathbf{x}^n$, and $I_n = \{j \in \mathbb{N}_0 : 0 \leq j \leq 2^n - 1\}$.

Lemma 6.1.3. *The codomain of function f_3 may be divided into $n+1$ disjoint subsets. All sequences of equal weight w in the domain of the function are injectively and surjectively mapped to the elements of the $(w+1)$th subset, where $w \in \{0, 1, \ldots n\}$.*

Proof. Let the set S_i, where $i \in \{0, 1, \ldots n\}$, be defined as

$$S_i = \left\{ \sum_{j=0}^{i-1} \binom{n}{j}, \sum_{j=0}^{i-1} \binom{n}{j} + 1, \ldots \sum_{j=0}^{i-1} \binom{n}{j} + \binom{n}{i} - 1 \right\}. \tag{6.49}$$

Set S_i is the $(i+1)$th set of the $n+1$ disjoint subsets into which the codomain of function f_3 is divided.

In order to prove that the codomain subsets are disjoint, it is sufficient to prove that

$$S_i \cap S_j = \emptyset \tag{6.50}$$

for all i, j with $i \neq j$. Equation 6.50 is derived as follows. The largest integer of the set S_i equals

$$m_x = \sum_{j=0}^{i-1} \binom{n}{j} + \binom{n}{i} - 1. \tag{6.51}$$

The smallest integer of the set S_{i+k}, where $k > 0$, equals

$$m_n = \sum_{j=0}^{i+k-1} \binom{n}{j}. \tag{6.52}$$

It follows that

$$\begin{aligned} m_x &= \sum_{j=0}^{i-1} \binom{n}{j} + \binom{n}{i} - 1 \\ &= \sum_{j=0}^{i} \binom{n}{j} - 1 \\ &< \sum_{j=0}^{i} \binom{n}{j} + \sum_{j=i+1}^{i+k-1} \binom{n}{j} \\ &< m_n. \end{aligned} \tag{6.53}$$

Due to the fact that $m_x < m_n$, and that m_x and m_n are respectively the largest and smallest integers of the sets S_i and S_{i+k}, equation 6.50 holds for all $0 \leq i < n$ and all

$$\begin{aligned} j &= i + k \\ &> i. \end{aligned} \tag{6.54}$$

As $S_i \cap S_j = S_j \cap S_i$, equation 6.50 holds for all $0 < i \leq n$ and all $j < i$. It follows that equation 6.50 holds for all integers i, j with $i \neq j$. The proof regarding the disjoint nature of the sets is therefore complete.

It is subsequently proved that the union of all sets S_i, where $i \in \{0, 1, \ldots n\}$, consists of all the elements in the codomain of function f_3. In order to complete the proof, the equation

$$\bigcup_{i=0}^{n} S_i = I_n \tag{6.55}$$

is derived from several observations. Let the largest integer of the set S_i be denoted by m_x, and the smallest integer of the set S_{i+1} by m_n. It follows that

$$\begin{aligned} m_n - m_x &= \sum_{j=0}^{i} \binom{n}{j} - \sum_{j=0}^{i-1} \binom{n}{j} - \binom{n}{i} + 1 \\ &= 1. \end{aligned} \tag{6.56}$$

This equation implies that

$$S_i \cup S_{i+1} = \left\{ v \in \mathbb{N}_0 : \sum_{j=0}^{i-1} \binom{n}{j} \leq v \leq \sum_{j=0}^{i+1} \binom{n}{j} - 1 \right\}. \tag{6.57}$$

Equation 6.57 is used to derive the expression

$$\begin{aligned} \bigcup_{i=0}^{n} S_i &= \left\{ v \in \mathbb{N}_0 : 0 \leq v \leq \sum_{j=0}^{n} \binom{n}{j} - 1 \right\} \\ &= \{ v \in \mathbb{N}_0 : 0 \leq v \leq 2^n - 1 \}, \end{aligned} \tag{6.58}$$

thereby proving that equation 6.55 holds.

The final statement of lemma 6.1.3 that remains to be proved is that all bit sequences of a certain weight w in the domain of function f_3 are injectively and surjectively mapped to the elements of S_w, where $w \in \{0, 1, \ldots n\}$. It was established that function f is invertible in the proof of theorem 6.1.2. Function f therefore maps all n–bit sequences of equal weight w both injectively and surjectively to the $\binom{n}{w}$ distinct integer elements in the set $\{0, 1, \ldots \binom{n}{w} - 1\}$. As the sum

$$t = \sum_{w'=0}^{w-1} \binom{n}{w'} \tag{6.59}$$

remains constant when evaluated in terms of all n–bit sequences with equal weight w, it follows that function f_3 maps all n–bit sequences with equal weight w both injectively and surjectively to the elements of the set

$$S_w = \left\{ \sum_{j=0}^{w-1} \binom{n}{j}, \sum_{j=0}^{w-1} \binom{n}{j} + 1, \ldots \sum_{j=0}^{w-1} \binom{n}{j} + \binom{n}{w} - 1 \right\}. \tag{6.60}$$

The proof of the lemma is therefore complete. □

Lemma 6.1.4. *The function f_3 is invertible.*

Proof. It is proved that function f_3 is both injective and surjective. All n–bit sequences of equal weight w, where $w \in \{0, 1, \ldots n\}$, are mapped injectively and surjectively to the elements of subset S_w of the codomain of function f_3, as proved in lemma 6.1.3. The function f_3 is therefore injective, as the subsets S_w, where $w \in \{0, 1, \ldots n\}$, are disjoint (refer to lemma 6.1.3). The function f_3 is surjective, as the union of all subsets S_w, where $w \in \{0, 1, \ldots n\}$, equals the codomain of function f_3, and all sequences of equal weight w are surjectively mapped to the same subset S_w. The conclusion that function f_3 is invertible follows from the fact that it is both injective and surjective. □

Theorem 6.1.3. *The variable weight, fixed–to–fixed length source code is uniquely decodable if $f_3(\mathbf{x}) < 2^k$ (i.e. if the index of the source sequence in the ordered set S_b is smaller than 2^k, where k is the codeword length).*

Proof. It is proved that the source code is uniquely decodable by proving that function g_2 of equation 6.47 is invertible over all source sequences $\mathbf{x}^n$ for which the condition $f_3(\mathbf{x}) < 2^k$ holds. If it is assumed that the inequality $f_3(\mathbf{x}) < 2^k$ holds, the function β_k may be replaced with the function B_k in equation 6.47. As B_k and f_3 are invertible (refer to lemma 6.1.4), function g_2 is invertible. The inverse of function g_2 is obtained as

$$\begin{aligned} \mathbf{x} &= f_3^{-1}(B_k^{-1}(\mathbf{c_x})) \\ &= f^{-1}\left(B_k^{-1}(\mathbf{c_x}) - \sum_{j=0}^{w-1} \binom{n}{j}\right). \end{aligned} \tag{6.61}$$

The weight w of the source sequence $\mathbf{x}^n$ may be recovered from the codeword $\mathbf{c_x}$ using the equation

$$w = \max(r) : \left[B_k^{-1}(\mathbf{c_x}) - \sum_{i=0}^{r-1} \binom{n}{i}\right] \geq 0.$$

□

6.1.2.4 Proof of optimality

The variable weight, fixed–to–fixed length source code achieves the minimum block–error probability when used to encode sequences from certain binary memoryless sources. This property of the source code is subsequently proved.

Theorem 6.1.4. *The variable weight, fixed–to–fixed length source code is optimal in terms of its block–error probability when used to encode sequences $\mathbf{x}^n$ of i.i.d. bits from a stationary binary memoryless source, where $\Pr(X_i = 1) < 0.5$ for all $1 \leq i \leq n$.*

Proof. Assume that the variable weight, fixed–to–fixed length source code does not achieve the minimum block–error probability, but that an alternative fixed–to–fixed length source code does. Denote the sets of source sequences that are correctly decoded by the variable weight, fixed–to–fixed length source code and the alternative source code as T_2 and T_2', respectively. If the assumption holds, then there has to exist two source sequences $\mathbf{x}^n$ and $\mathbf{y}^n$ that satisfy the conditions

1. $\mathbf{x}^n \in T_2'$,
2. $\mathbf{x}^n \notin T_2$, and
3. $\exists\, \mathbf{y}^n \in T_2 : \Pr(\mathbf{x}^n) > \Pr(\mathbf{y}^n)$.

The inequality $\Pr(\mathbf{x}^n) > \Pr(\mathbf{y}^n)$ implies that

$$\Pr(\mathbf{x}^n) > \min_{\mathbf{z}^n \in T_2} \Pr(\mathbf{z}^n). \tag{6.62}$$

Let the weight of the bit sequence $\mathbf{x}^n$ be denoted by $W(\mathbf{x})$. Equation 6.62 implies that

$$W(\mathbf{x}) \quad < \quad \max_{\mathbf{z} \in T_2} W(\mathbf{z}), \tag{6.63}$$

due to the fact that $\Pr(Z_i = 1) < 0.5$ for all $1 \leq i \leq n$. Let the weight w' be defined according to the equation

$$w' = \max_{\mathbf{z} \in T_2} W(\mathbf{z}). \tag{6.64}$$

It follows from lemma 6.1.3 that all sequences $\mathbf{q} : f_3(\mathbf{q}) \leq \sum_{j=0}^{w'-1} \binom{n}{j} - 1$ belong to the set T_2, as the inequality

$$f_3(\mathbf{z}) > \sum_{j=0}^{w'-1} \binom{n}{j} - 1 \tag{6.65}$$

holds for any $\mathbf{z} : W(\mathbf{z}) = w'$. As all sequences $\mathbf{q} : W(\mathbf{q}) < w'$ satisfy the inequality

$$f_3(\mathbf{q}) \leq \sum_{j=0}^{w'-1} \binom{n}{j} - 1, \tag{6.66}$$

it follows that all sequences $\mathbf{q} : W(\mathbf{q}) < w'$ belong to the set T_2. This fact proves that equation 6.63 cannot be satisfied for any $\mathbf{x}^n \notin T_2$, which contradicts the assumption that the alternative fixed–to–fixed length source code achieves the minimum block–error probability. The variable weight, fixed–to–fixed length source code therefore achieves the minimum block–error probability. □

6.1.2.5 Theoretical performance

The theoretical block–error probability of the variable weight, fixed–to–fixed length source code is derived in this section. It is assumed that the source code is used to encode n–bit sequences $\mathbf{x}^n$ from stationary binary memoryless sources with $\Pr(X_i = 1) < 0.5$ for all $1 \leq i \leq n$. The source coding of both finite–length source sequences, as well as source sequences with lengths that tend to infinity, are considered.

Performance w.r.t. finite–length sequences Let the weight w' be defined as

$$w' = \max_{\mathbf{y} \in T_2} W(\mathbf{y}), \tag{6.67}$$

where T_2 is the set of all n–bit source sequences correctly decoded by the variable weight, fixed–to–fixed length source code with codewords of length k bits, and $W(\mathbf{y})$

is the weight of the source sequence $\mathbf{y}^n$. The number of source sequences of weight w' that are not elements of the set T_2 equals

$$q = \sum_{i=0}^{w'} \binom{n}{i} - 2^k. \tag{6.68}$$

The exact block–error probability is therefore derived as

$$P_b = qp^{w'}(1-p)^{n-w'} + \sum_{i=w'+1}^{n} \binom{n}{i} p^i (1-p)^{n-i}, \tag{6.69}$$

where $p \triangleq \Pr(X_i = 1)$.

An upper bound on the block–error probability of the variable weight, fixed–to–fixed length source code is also derived. The derivation uses the inequality [155]

$$\sum_{i=\delta n}^{n} \binom{n}{i} p^i (1-p)^{n-i} \leq 2^{-nD(\delta||p)}, \tag{6.70}$$

which is valid if the condition $0 < p < \delta \leq 1$ is satisfied. The function $D(p||q)$ is the binary divergence function (or the Kullback–Leibler divergence function), which is defined as

$$D(p||q) \triangleq p \log_2 \left(\frac{p}{q}\right) + (1-p) \log_2 \left(\frac{1-p}{1-q}\right). \tag{6.71}$$

Let the weight w' be defined according to equation 6.67. If all weight–w' source sequences that are elements of the set T_2 are associated with block errors, the block–error probability may be bounded as

$$\begin{aligned} P_b &\leq \sum_{i=w'}^{n} \binom{n}{i} p^i (1-p)^{n-i} \\ &\leq 2^{-nD(\hat{p}||p)}, \end{aligned} \tag{6.72}$$

where

$$\hat{p} \triangleq \frac{w'}{n}. \tag{6.73}$$

The bound on the block–error probability is valid provided that the inequality

$$0 < p < \hat{p} \leq 1 \tag{6.74}$$

is satisfied.

Asymptotic performance The performance of the variable weight, fixed–to–fixed length source code, when used to encode source sequences with lengths that tend to infinity, is characterized according to Shannon's source coding theorem [44]. The source coding theorem[4] states that it is possible for a fixed–to–fixed length source code with a rate of

$$\frac{k}{n} = h(p) + \delta \tag{6.75}$$

[4] All references to the source coding theorem in this section are made under the assumption that sequences from a stationary binary memoryless source are encoded.

codeword bits per source bit to achieve a block–error probability P_b that tends to zero if sequences of n bits, where n tends to infinity, are encoded (δ denotes an arbitrarily small positive constant). The function $h(\cdot)$ is the binary entropy function. The variable weight, fixed–to–fixed length source code is an example of a fixed–to–fixed length source code predicted to exist by the source coding theorem, due to its optimal block–error probability (refer to theorem 6.1.4).

The strong converse source coding theorem states that the block–error probability of any fixed–to–fixed length source code that is used to encode source sequences of n bits, where n tends to infinity, approaches unity if the rate of the source code satisfies

$$\frac{k}{n} < h(p). \tag{6.76}$$

The block–error probability of the variable weight, fixed–to–fixed length source code, when used to encode sequences with lengths that tend to infinity, is therefore derived as

$$P_b \rightarrow \begin{cases} 0 & \text{if } R_a > h(p), \\ 1 & \text{if } R_a < h(p), \end{cases}$$

where the code rate R_a is expressed as

$$R_a = \frac{k}{n}. \tag{6.77}$$

6.1.2.6 Practical results

The variable weight, fixed–to–fixed length source code was used to encode sequences of i.i.d. bits in two configurations. The block–error probability of the source code was investigated in each case. Both of the configurations, as well as the block–error probabilities, are presented in what follows. The variable weight, fixed–to–fixed length source code is referred to as the proposed source code in the remainder of this section.

Configuration 1: Variable source sequence length The first configuration of the proposed source code involved three binary memoryless sources. The output of each source was encoded using four instances of the proposed source code. Each source code instance was used to encode source sequences of a different length, but the source code instances were selected to have approximately the same code rate. The details of the three binary memoryless sources are provided in table 6.3, and the parameters and quantities of the source codes[5] are provided in table 6.4. The weight w' associated with each source code, as defined by equation 6.67, is provided in table 6.4.

The source coding results associated with the proposed source code in the first configuration are presented in figure 6.4 on page 163. This figure contains a plot of the block–error probabilities of the source codes, which were used to encode sequences from each source. The block–error probability is plotted as a function of the source sequence length. Each curve is associated with one of the three sources, and either a theoretical calculation or the practical implementations.

[5]The term 'instance' is omitted in the remainder of this section.

Table 6.3: Source parameters and quantities associated with the first configuration of the variable weight, fixed–to–fixed length code.

Parameter / Quantity	Symbol	Unit	Source 1	Source 2	Source 3
Prob. nonzero–valued bit	p	—	0.089	0.109	0.16
Per–symbol entropy	$H(X)$	bits	0.4331	0.4969	0.6343

Table 6.4: Source code parameters and quantities associated with the first configuration of the variable weight, fixed–to–fixed length code.

Parameter / Quantity	Symbol	Unit	Code 1	Code 2	Code 3	Code 4
Src. sequence length	n	bits	10	50	100	500
Codeword length	k	bits	6	30	60	303
Code rate	R_a	—	0.6	0.6	0.6	0.606
Maximum decodable src. sequence weight	w'	bits	3	9	16	76

The figure reveals that the curves associated with the practical implementations overlap the curves of the theoretically–optimal block–error probability (i.e. the exact block–error probability of equation 6.69). This observation implies that the source codes were correctly implemented, and that the performance of each source code is indeed optimal.

Figure 6.4 reveals that the block–error probability curves associated with source one and source two decrease with an increase in the source sequence length, but that the block–error probability curve associated with source three increases with an increase in the source sequence length. This observation may be justified by considering the rates of the codes, as well as the per–symbol entropy of each of the sources. All the source codes have rates approximately equal to 0.6. Shannon's source coding theorem [44] states that the block–error probability of a source code cannot approach zero asymptotically if the rate of the source code is lower than the entropy of the source. Tables 6.3 and 6.4 reveal that all the source codes have rates that exceed the per–symbol entropy of sources one and two, but that all the source codes have rates lower than the per–symbol entropy of source three, which equals 0.6343 bits. The block–error probabilities of the source codes, when used to encode sequences from source three, asymptotically approach unity due to the strong converse source coding theorem.

Figure 6.4 reveals that the curves associated with the upper bounds on the block–error probabilities of the source codes (equation 6.72), when used to encode sequences from sources one and two, are not exceeded by the curves of the practical implementations. No upper bound on the block–error probabilities associated with the source coding of sequences from source three is provided, as source codes three and four do not satisfy equation 6.74 if it is evaluated in terms of source three.

Configuration 2: Variable source entropy The second configuration of the proposed source code involved the source coding of sequences from a single binary memo-

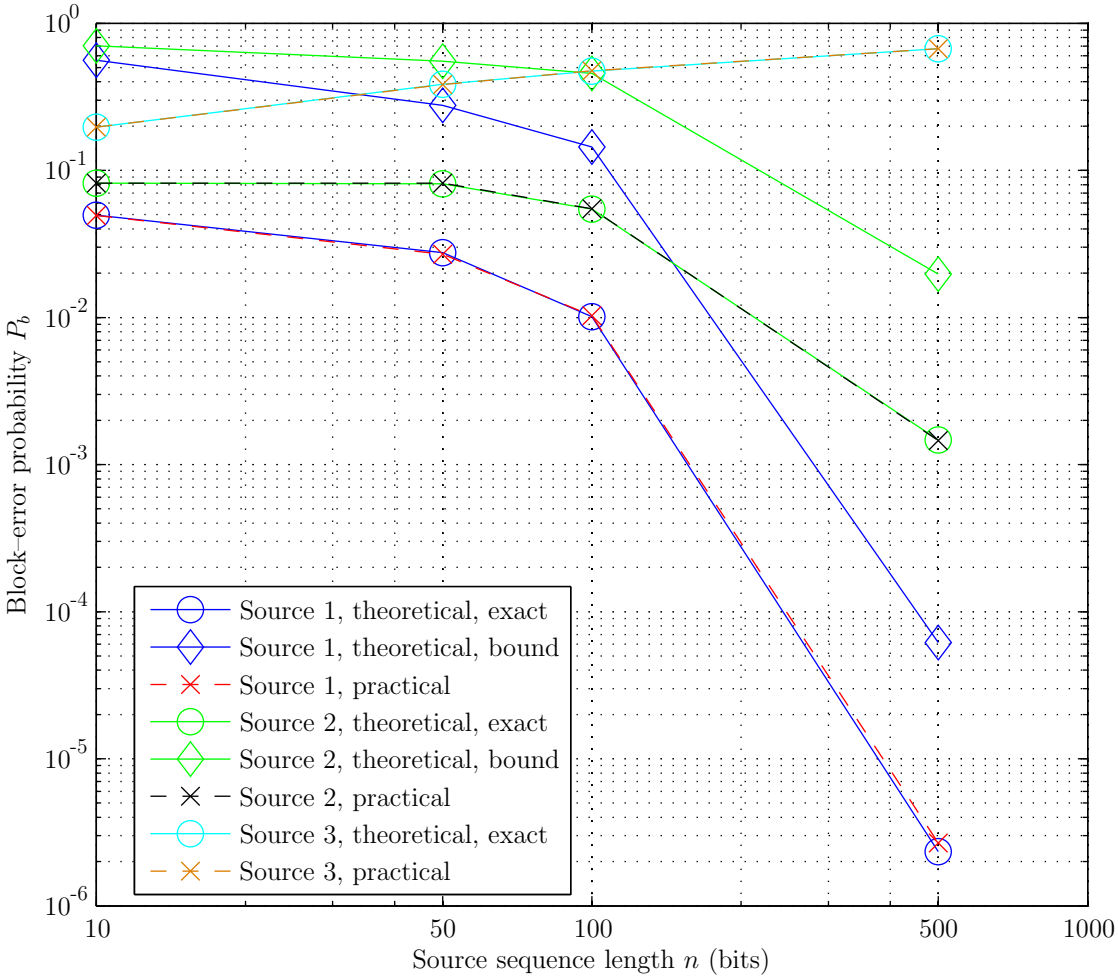


Figure 6.4: Block–error probabilities of the variable weight, fixed–to–fixed length source code in the first configuration, as a function of the source sequence length.

ryless source. The proposed source code was implemented with a fixed code rate. The per–symbol entropy of the source was varied by changing the probability distribution of the bits of the source sequences for each successive set of trials. The purpose of this configuration was to characterize the block–error probability of the proposed source code as a function of the source entropy. The parameters and quantities associated with the proposed source code in the second configuration are provided in table 6.5.

The source coding results associated with the proposed source code in the second configuration are presented in figure 6.5 on page 165. This figure contains a plot of the block–error probability of the proposed source code, as a function of the entropy–normalized redundancy of the source code. The entropy–normalized redundancy of the source code is defined as the normalized difference between the rate of the code and the per–symbol entropy of the source, or

$$R_n = \frac{R_a - H(X)}{H(X)}. \tag{6.78}$$

Figure 6.5 includes the curve of the theoretical block–error probability of the pro-

Table 6.5: Parameters and quantities associated with the second configuration of the variable weight, fixed–to–fixed length source code.

Parameter / Quantity	Symbol	Value	Unit
Source sequence length	n	500	bits
Codeword length	k	303	bits
Code rate	R_a	0.606	—
Maximum decodable source sequence weight	w'	76	bits
Prob. nonzero–valued source bit	p	0.089 – 0.109	—
Per–symbol source entropy	$H(X)$	0.4331 – 0.4969	bits

posed code (equation 6.69), the curve of the upper bound on the block–error probability of the proposed code (equation 6.72), as well as the block–error probability curve of the practical implementation of the proposed code. The curves of the block–error probability decrease with an increase in the redundancy of the proposed source code, as expected. The curve of the theoretical block–error probability of the proposed source code overlaps with the block–error probability curve of the practical implementation of the proposed source code. This observation implies that the proposed source code was correctly implemented. The block–error probability of the proposed source code does not exceed the upper bound of equation 6.72.

Figure 6.5 includes two curves that are associated with fixed–to–variable length source codes. These source codes were applied as fixed–to–fixed length source codes. The variable–length codes were used to encode n–bit fixed–length source sequences. A block error was declared for each variable–length codeword that was longer than k bits, where k is the codeword length of the proposed source code. All variable–length codewords shorter than k bits were padded to a length of k bits using zero–valued bits. A curve of the theoretical block–error probability of the ideal variable–length source code, as well as a curve of the block–error probability of a Matlab arithmetic code implementation, are included in the figure.

The ideal variable–length source code produces a codeword of length $-\log_2(\Pr(\mathbf{x}^n))$ bits when used to encode a source sequence $\mathbf{x}^n$ with a probability of occurrence of $\Pr(\mathbf{x}^n)$. Figure 6.5 reveals that the block–error probability of this source code is significantly higher than the block–error probability of the proposed source code. This observation confirms the observation of Caire et. al. [85] regarding the suboptimal nature of fixed–to–variable length source codes that are applied as fixed–to–fixed length source codes.

The curve associated with the Matlab implementation of the arithmetic code was obtained from reference [85]. The arithmetic code was implemented as a nonuniversal code that had exact apriori knowledge of the probability distribution of the source bits. Figure 6.5 reveals that the arithmetic code has a higher block–error probability than the ideal variable–length code, as well as the optimal fixed–to–fixed length source code. The suboptimal nature of the arithmetic code, when used as a fixed–to–fixed length source code, is apparent.

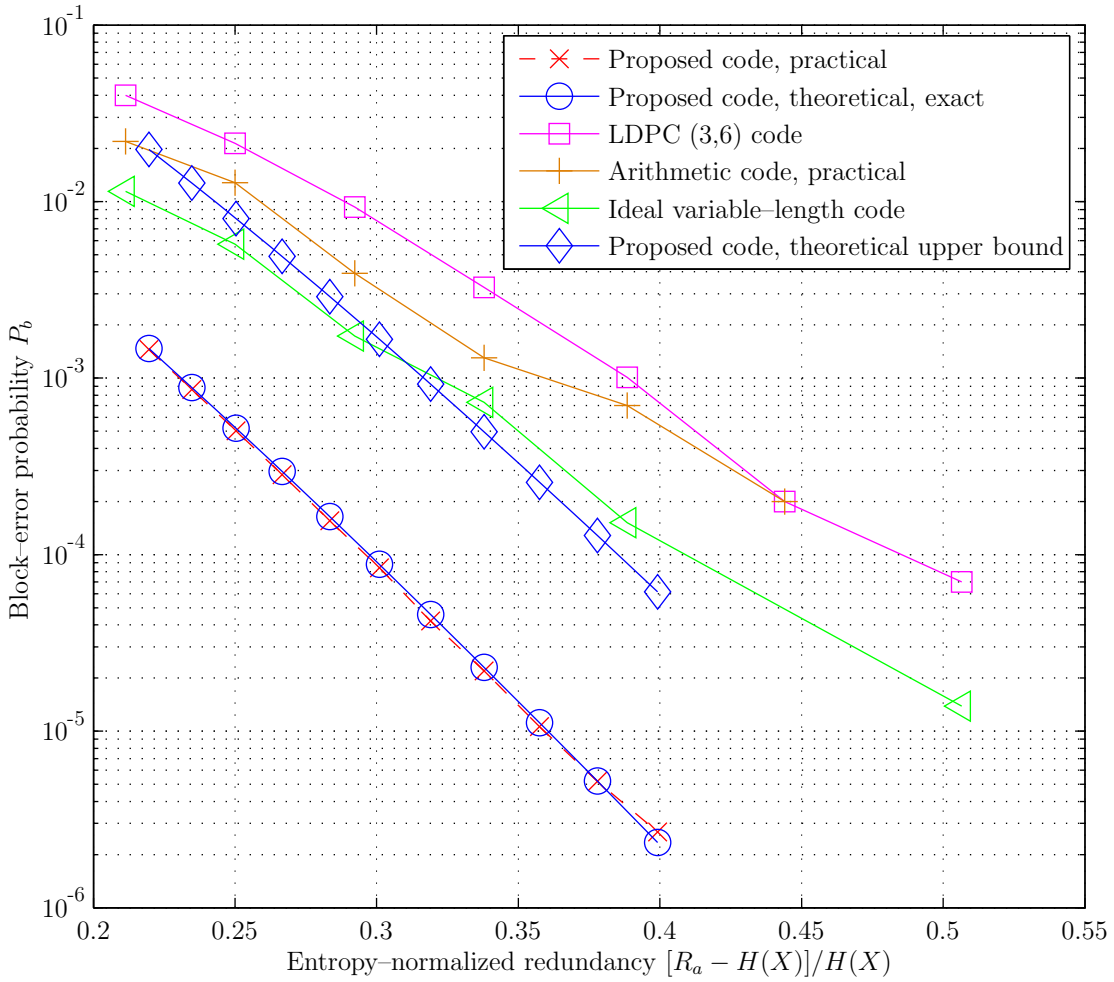


Figure 6.5: Block–error probability of the variable weight, fixed–to–fixed length source code in the second configuration, as a function of the code redundancy. Certain parts of this figure were adapted from reference [85].

Computational complexity The average processing time of a practical implementation of the variable weight, fixed–to–fixed length source code was measured in order to investigate the computational complexity of the prime factor decomposition approach to computing large binomial coefficients (refer to section 5.3 on page 136). The implementation encoded source sequences of $n = 10^4$ bits, and produced codewords of $k = 5000$ bits. It was therefore required that the implementation be able to compute all binomial coefficients $\binom{a}{b}$, where $a \in \{0, 1, \ldots 10^4\}$ and $b \in \{0, 1, \ldots a\}$. The source sequences were generated by a stationary binary memoryless source, and the probability of occurrence of a nonzero–valued bit was set equal to $p = 0.1$.

The variable weight, fixed–to–fixed length source code was implemented on a modern laptop PC. The processing times associated with encoding the source sequences and decoding the codewords were measured. The practical implementation of the source encoder required an average of $\mu_c = 0.434$ seconds to encode a source sequence, with a standard deviation of $\sigma_c = 0.018$ seconds. The practical implementation of the source

decoder required an average of $\mu_d = 2.388$ seconds to decode a codeword, with a standard deviation of $\sigma_d = 0.038$ seconds. The time required to encode a source sequence and decode the codeword is not excessive if the magnitude of the binomial coefficients that the implementation computes is taken into consideration.

6.2 Multi–field enumerative source codes

This section contains a summary of two lossless enumerative source codes that are more sophisticated than the single–field enumerative source codes. The enumerative source codes that are introduced in this section are referred to as multi–field enumerative codes, as these codes have variable–length codewords that consist of more than one field. One of the multi–field enumerative source codes presented in this section is novel.

The summary of each source code starts with the derivation of the code, as well as the mathematical definition of the code. Each summary includes a proof of the unique decodability of the source code, as well as bounds on the performance of the source code. Each summary concludes with a presentation and a discussion of the source coding results that were obtained using a practical implementation of the source code.

6.2.1 The weight–and–index variable–length code

The weight–and–index source code is a fixed–to–variable length, lossless, blockwise enumerative source code. The source code may be used to effectively encode bit sequences in a universal manner from a variety of sources. The first field of each codeword of this source code represents the weight of the corresponding fixed–length source sequence. The second field of each codeword represents the index of the corresponding fixed–length source sequence in an ordered set that contains all source sequences with the same length and weight.

6.2.1.1 Derivation of the source code

The weight–and–index source code may be used to separately encode bit sequences from binary memoryless sources and binary context–tree sources[6] in a universal manner. This statement implies that the source code produces codewords with a normalized average redundancy that tends to zero as the source sequence length n tends to infinity. The source code has no apriori knowledge of the distribution of the source bits. The exact structure of the source code depends on whether it is applied to sequences from a binary memoryless source or a binary context–tree source, and whether it is used in a universal or nonuniversal configuration[7]. The weight–and–index source code for sequences from binary memoryless sources is solely a universal source code.

The derivation of the weight–and–index source code is divided into two sections. The first section concerns the source coding of sequences from binary memoryless

[6] These sources are referred to as tree sources in section 3.1.1.3 on page 40.

[7] The source encoder requires information on whether it is encoding sequences from a binary memoryless source or a binary context–tree source in order to universally encode sequences from the specific source.

sources, and the second section concerns the source coding of sequences from binary context–tree sources.

Binary memoryless sources The first field of each codeword of the weight–and–index source code for binary memoryless sources represents the weight of the n–bit source sequence that is associated with the codeword. The weight of each source sequence assumes one of $n+1$ distinct values. The weights are encoded using the conventional binary–coded representation of the integers. The weight of each source sequence $\mathbf{x}^n$ is encoded in a total of

$$l_1(\mathbf{x}) = \lceil \log_2(n+1) \rceil \tag{6.79}$$

bits.

Two observations regarding the effectiveness of the conventional binary–coded representation of the source sequence weights are relevant. The first observation concerns the overhead that is associated with representing the weight of each source sequence in an integer number of bits. This overhead does not exceed one bit, but may become significant if the source code is used to encode short source sequences (the overhead contributes significantly to the length of each codeword).

The second observation concerns the assumption regarding the probability distribution of the source sequence weights. By encoding all distinct source sequence weights in an equal number of bits, the source encoder implicitly assumes that the weights are equiprobable. The optimal source encoder encodes the sequence weight w in $-\log_2(\Pr(w))$ bits, where $\Pr(w)$ denotes the probability of occurrence of the sequence weight w. The distinct source sequence weights are not equiprobable — in the case of the binary memoryless source, the weight W of each source sequence has a binomial distribution

$$\Pr(W = w) = \binom{n}{w} p^w (1-p)^{n-w}, \tag{6.80}$$

where $p \triangleq \Pr(X_i = 1)$. A source encoder that encodes each distinct source sequence weight in the appropriate number of bits would improve the performance of the weight–and–index source code.

Two constraints are imposed on a possible source code for the weights of the source sequences. The first constraint is related to the universal nature of the weight–and–index source code. In order to preserve the universal nature of the weight–and–index source code, any source code for the weights of the source sequences must not require apriori knowledge of the distribution of the sequence weights. The second constraint limits the computational complexity of the encoder and decoder for the sequence weights, as both the weight and index of each n–bit source sequence need to be encoded and successfully decoded. Universal integer codes such as the Elias–γ code [136] satisfy both of these requirements. A universal integer code may be used to encode the weights of the source sequences, provided that the distribution implied by the code matches the actual distribution of the source sequence weight. If this condition is not satisfied, the weight–and–index source code may not be effective.

Davisson [6] observed that a universal source code may be constructed by encoding the sequence weights using the conventional binary–coded representation of the integers. This observation justifies the decision of using the conventional binary–coded

representation of the source sequence weights in the codewords of the multi–field enumerative source codes considered in this chapter. The source decoder may uniquely decode the source sequence weight from the first field of any codeword of the weight–and–index source code, provided it has apriori knowledge of the length n of the source sequence.

The second field of each codeword of the weight–and–index variable–length source code represents the index of the corresponding n–bit source sequence $\mathbf{x}^n$ in the ordered set S_b of all n–bit sequences having the same weight w as the source sequence. Each index is identical to the index that the constant weight, fixed–to–fixed length source code[8] would produce if it were used to encode the corresponding source sequence. The encoder of the weight–and–index source code encodes each index using the conventional binary–coded representation of an integer — the second field of each codeword is therefore identical to the corresponding codeword of the lossless, constant weight, fixed–to–fixed length source code. The second field of each codeword has a length of

$$l_2(\mathbf{x}) = \left\lceil \log_2 \binom{n}{w} \right\rceil \tag{6.81}$$

bits, where w is the weight of the source sequence associated with the codeword.

In order to justify the use of the same number of bits to represent the index of each distinct source sequence, one may consider the probability distribution of a source sequence, with the distribution conditioned on the sequence weight. As a stationary binary memoryless source produces i.i.d. bits, the distribution of a source sequence, conditioned on the sequence weight w, is a uniform distribution. This observation suggests that the use of the same number of bits to represent each index is an effective strategy.

The unique decodability of the weight–and–index variable–length source code depends on whether the source decoder is able to delimit the second field of each codeword. The source decoder must be able to delimit the second field of each codeword as it must identify the start of the following codeword in the sequence of codewords. The second field of any codeword may be delimited by the source decoder after it recovers the weight of the source sequence from the first field of the codeword — the length of the second field is calculated using equation 6.81.

Binary context–tree sources The derivation of a universal enumerative source code for sequences from binary context–tree sources is significantly more challenging than the derivation of a universal enumerative code for sequences from binary memoryless sources. One approach to deriving a universal enumerative code for sequences from binary context–tree sources involves the division of all possible source sequences into type classes, and the development of an algorithm for enumerating the sequences within each type class [81]. A type class is essentially an ordered set that contains equiprobable source sequences. Each codeword of a universal source code of this type would indicate the type class to which the corresponding source sequence belongs, as well as its index in the type class.

[8] Refer to section 6.1.1 on page 139 for a summary of this source code.

A simpler universal enumerative source code for sequences from binary context–tree sources was derived and implemented as part of this thesis. This source code uses the Burrows–Wheeler transform [9], which was introduced in chapter 3. The Burrows–Wheeler transform, or BWT, produces sequences of p.i.i.d. bits[9] if applied to source sequences from binary context–tree sources [10]. Each sequence of p.i.i.d. bits may be encoded by using the weight–and–index variable–length source code for binary memoryless sources to independently encode each piecewise segment of i.i.d. bits.

The expected positions of the transition points between the segments of i.i.d. bits in the BWT output sequences depend on the statistics of the source. If the source encoder is aware of the structure of the context–tree source (i.e. its minimum suffix set) and a certain number of the initial states of the source, it may calculate the exact positions of the transition points between the segments of the BWT output sequence. It independently encodes each segment of the BWT output sequence using the weight–and–index variable–length source code for binary memoryless sources. This source code is clearly not universal, as it requires apriori knowledge of the source. It is referred to as the type–one nonuniversal weight–and–index source code for binary context–tree sources.

If the source encoder is aware of the bit distribution that is associated with each state of the source, it may calculate the expected positions of the transition points between the segments of i.i.d. bits in the BWT output sequence, and independently encode the sequences between the expected transition points. Each sequence is encoded using the weight–and–index variable–length source code for binary memoryless sources. This source code is not universal, as it requires apriori knowledge of the source — it is referred to as the type–two nonuniversal weight–and–index source code for binary context–tree sources.

If the source encoder has no apriori knowledge regarding the structure of the binary context–tree source, or the bit distribution associated with each of the source states, it may estimate the positions of the transition points between the segments of i.i.d. bits in the BWT output sequence. A universal source code has to accurately estimate both the number of transition points and the position of each transition point. The sequences between the estimated positions of the transition points are independently encoded using the weight–and–index variable–length source code for binary memoryless sources. This source code is referred to as the universal weight–and–index source code for binary context–tree sources. Both the nonuniversal and universal weight–and–index source codes are discussed in what follows.

The type–one nonuniversal weight–and–index code A block diagram of the type–one nonuniversal weight–and–index source encoder for sequences from binary context–tree sources is presented in figure 6.6. The source encoder reverses the source sequence $\mathbf{x}^n$ and appends the `EOF` symbol to the reversed sequence. It subsequently applies the BWT to the sequence. The BWT output sequence $\mathbf{z}^{n+1}$ and the BWT index I may be expressed as

$$(\mathbf{z}^{n+1}, I) = \mathrm{BWT}(\mathcal{R}(\mathbf{x}^n)\$), \tag{6.82}$$

[9]The BWT output is asymptotically p.i.i.d. For the sake of brevity, the output is described as only being p.i.i.d. in the remainder of this section.

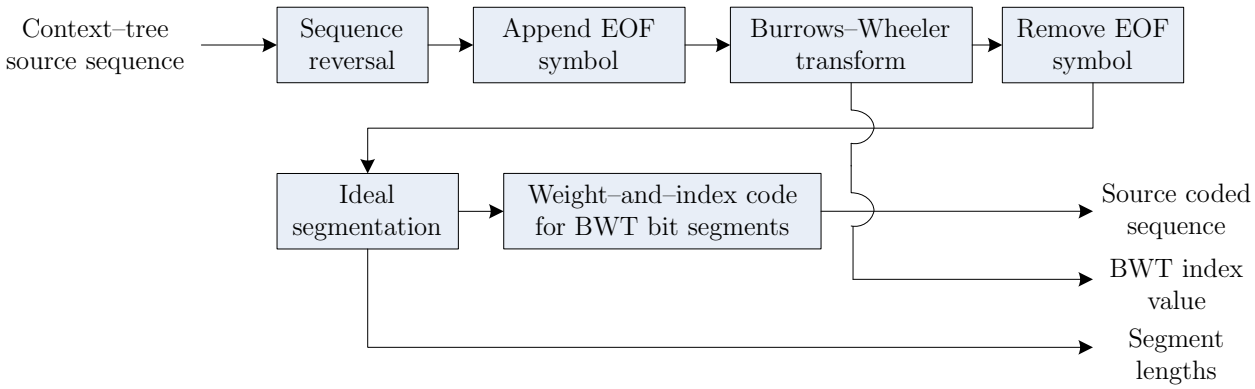


Figure 6.6: A block diagram of the type–one nonuniversal weight–and–index source encoder for binary context–tree sources.

where \$ denotes the `EOF` symbol, and $\mathcal{R}(\cdot)$ denotes sequence reversal. The source encoder removes the `EOF` symbol from the BWT output sequence to obtain the sequence $\mathbf{w}^n$, which is expressed as

$$\begin{aligned} \mathbf{w}^n &= (z_1, z_2, \ldots z_{I-1}, z_{I+1}, \ldots z_{n+1}) \\ &= (\mathbf{z}_1^{I-1}, \mathbf{z}_{I+1}^{n+1}). \end{aligned} \tag{6.83}$$

The reversal of the source sequence causes the BWT to sort the reversed preceding contexts of the original source sequence relative to one another. Those bits that follow similar preceding contexts[10] in the original source sequence are therefore placed in neighbouring segments of the BWT output sequence. The preceding contexts of the source symbols are identical to the reversed states (i.e. the suffixes of the minimum suffix set) of the context–tree source.

In order to find the exact positions of the transition points between the i.i.d. bit segments of the BWT output sequence, the source encoder derives the sequence of state transitions that corresponds to the original source sequence. It derives the sequence of state transitions by using its apriori knowledge of the source's minimum suffix set and one or more of its initial states. It counts the number of bits that the source produced in each of its states. The encoder proceeds by sorting the source states (i.e. the reversed preceding contexts) and assigning a bit count to each of the source states. Each bit count is the total number of bits that the source produced in the corresponding state. The sequence of sorted states and the bit counts of the states are equivalent to the sequence of segments in the BWT output sequence and the segment lengths. The exact positions of the transition points are obtained from the segment lengths.

The discussion up to this point disregarded the fact that up to m of the initial bits of the source sequence may be placed in segments that do not correspond to states of the context–tree source, where m denotes the length of the longest preceding context of the source. The source encoder is able to derive the positions of these bits in the BWT output sequence, and insert transition points on either side of each of these bits. This

[10]The m–bit preceding context $\mathbf{x}_{i-m}^{i-1}$ of bit x_i is said to be similar to the preceding context of bit y_i if the bits closest to x_i match the bits closest to y_i.

approach may significantly increase the average codeword redundancy of the source code, however.

By inserting transition points on either side of segments that do not correspond to the states of the context–tree source, and encoding these additional segments independently from the remaining segments of the BWT output sequence, the source encoder could potentially add $m+1$ additional fields to a codeword. If the source encoder is not to use more complex integer codes, it must either encode the position of each additional segment in the BWT output sequence (and add up to m fields of $\lceil \log_2(n) \rceil$ bits each to the codeword), or encode the lengths of all the segments in the BWT output sequence (and add up to m fields of $\lceil \log_2(n) \rceil$ bits each to the codeword, as each length is encoded in $\lceil \log_2(n) \rceil$ bits). An additional field that represents the number of segments in the BWT output sequence would have to be added to each codeword, as the source decoder is unaware of the number of additional segments in each BWT output sequence.

Instead of encoding the additional segments of the BWT output sequence independently from the remaining segments, the source encoder may merge the additional segments with the segments that correspond to the states of the context–tree source. The addition of a maximum of m bits to the remaining segments, each consisting of bits with a distinct probability distribution, does increase the average redundancy of the codewords assigned to these segments. It is assumed, however, that the source sequence length n is much larger than the length of the longest source context, m. This assumption implies that the redundancy increase associated with the segment–merge approach is small when compared to the redundancy increase associated with independently encoding the additional segments. The encoder of the type–one nonuniversal weight–and–index source code therefore merges the additional segments with the segments that correspond to the states of the context–tree source.

The source encoder independently encodes the sequences between the transition points in the BWT output sequence using the weight–and–index source code for binary memoryless sources. It concatenates the codewords to produce the source–coded sequence. The first field of codeword i represents the weight of segment i of the BWT output sequence. This field has a length of

$$l_{1,i}(\mathbf{x}) = \lceil \log_2(v'_i + 1) \rceil \tag{6.84}$$

bits, where $\mathbf{x}$ denotes the source sequence, and v'_i denotes the length of segment i of the BWT output sequence (which may include up to m merged bits from segments that do not correspond to the states of the context–tree source). This expression holds for $i \in \{1, 2, \ldots |\mathcal{S}|\}$. The second field of codeword i represents the index of segment i of the BWT output sequence in the ordered set of all sequences with the same length and weight as the segment. This field has a length of

$$l_{2,i}(\mathbf{x}) = \left\lceil \log_2 \binom{v'_i}{v''_i} \right\rceil \tag{6.85}$$

bits, where v''_i denotes the weight of segment i of the BWT output sequence (which may include up to m merged bits), and $i \in \{1, 2, \ldots |\mathcal{S}|\}$.

In order to successfully recover the source sequence from the source–coded sequence, the source decoder requires the BWT index, as well as the lengths of the segments in

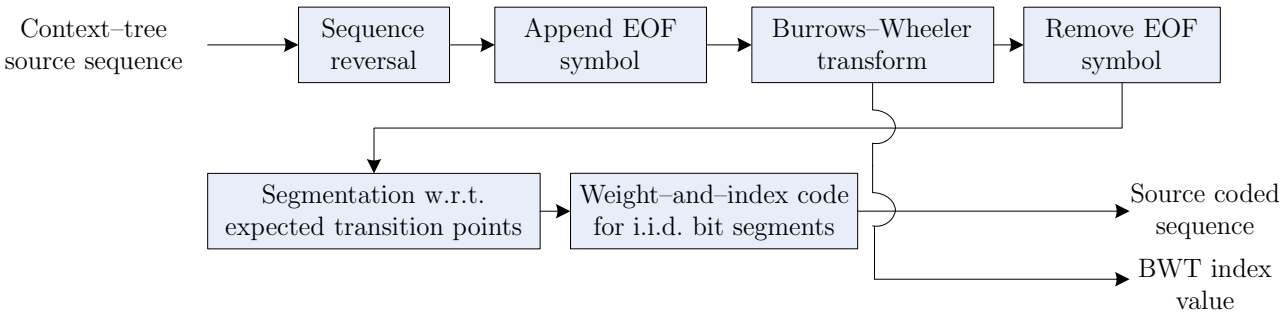


Figure 6.7: A block diagram of the type–two nonuniversal weight–and–index source encoder for binary context–tree sources.

the BWT output sequence. The source encoder encodes the BWT index using the conventional binary–coded representation of an integer. It is encoded in a total of

$$l_3(\mathbf{x}) = \lceil \log_2(n+1) \rceil \tag{6.86}$$

bits, as the BWT index may assume $n+1$ distinct values with the addition of the `EOF` symbol to the reversed source sequence.

The lengths of the segments in the BWT output sequence are encoded as follows. Each BWT output sequence has exactly $|\mathcal{S}|$ segments, as the additional segments are merged with the segments that correspond to the states of the context–tree source. The source encoder therefore encodes a total of $|\mathcal{S}| - 1$ segment lengths for each BWT output sequence, as the length of the final segment may be derived from the sequence length n. Each segment length is encoded in a total of $\lceil \log_2(n+1) \rceil$ bits using the conventional binary–coded representation of an integer, as each segment length may assume a value in the set $\{0, 1, 2, \ldots n\}$. The total number of bits that are required to encode the segment lengths equals

$$l_4(\mathbf{x}) = (|\mathcal{S}| - 1) \lceil \log_2(n+1) \rceil . \tag{6.87}$$

The type–two nonuniversal weight–and–index code A block diagram of the type–two nonuniversal weight–and–index source encoder for sequences from binary context–tree sources is presented in figure 6.7. The encoder performs the same initial steps as the encoder of the type–one nonuniversal weight–and–index code for binary context–tree sources; however, it does not find the exact positions of the transition points between the segments of each BWT output sequence. It calculates the expected positions of the transition points.

The expected positions of the transition points between the segments of the BWT output sequence may be obtained by calculating the expected lengths of the segments. Each segment of the BWT output sequence that is longer than one bit[11] corresponds

[11] Certain single–bit segments of the BWT output sequence may not correspond to the states of the context–tree source. These segments appear due to the addition of the `EOF` symbol to the reversed source sequence, and are disregarded in the calculation of the expected segment lengths. The single–bit segments have little impact on the performance of the source code, as a maximum of m of these segments are present in each BWT output sequence (where m is the length of the longest source state).

to one of the states of the context–tree source. The expected length of each of these segments equals the product of the corresponding state probability and the source sequence length.

Consider a binary context–tree source with a state set $\mathcal{S}$. The expected length $\overline{n}_i$ of the segment in the BWT output sequence that corresponds to the ith lexicographically–ranked state may be expressed as

$$\overline{n}_i = nq_i, \tag{6.88}$$

where q_i is the probability associated with the ith lexicographically–ranked state of the context–tree source, and $i \in \{1, 2, \ldots |\mathcal{S}|\}$. The expected index of the first bit in segment i of the BWT output sequence is expressed as

$$\overline{T}_i = \begin{cases} 1 & \text{if } i = 1, \\ \sum_{j=1}^{i-1} \overline{n}_j + 1 & \text{if } 1 < i \leq |\mathcal{S}|. \end{cases} \tag{6.89}$$

The probability associated with each state of the binary context–tree source is calculated using its FSM closure. The construction of the FSM closure of a context–tree source was summarized in section 3.1.1.3 on page 42 and is not repeated here. Let the state set of the FSM closure of the context–tree source be denoted by $\mathcal{S}'$, and let the states of the state set be numbered from one to $|\mathcal{S}'|$. The state–transition probability matrix $\mathbf{S}$ of the FSM closure is defined as

$$\mathbf{S} = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,|\mathcal{S}'|} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,|\mathcal{S}'|} \\ \vdots & \vdots & \ddots & \\ p_{|\mathcal{S}'|,1} & p_{|\mathcal{S}'|,2} & & p_{|\mathcal{S}'|,|\mathcal{S}'|} \end{pmatrix}, \tag{6.90}$$

where $p_{x,y}$ denotes the probability of a transition to state $\mathbf{s}'_y$, conditioned on the FSM closure being in state $\mathbf{s}'_x$. The state probability vector of the FSM closure is defined as

$$\mathbf{P} = [p_1, p_2, \ldots p_{|\mathcal{S}'|}]^T, \tag{6.91}$$

where p_i denotes the probability of the FSM closure being in state $\mathbf{s}'_i$, and T denotes the transpose of the vector. If the FSM closure of the binary context–tree source is stationary, the equation

$$\mathbf{S}^T\mathbf{P} = \mathbf{P} \tag{6.92}$$

holds. It follows that the state probability vector equals the normalized eigenvector, of the transposed state–transition probability matrix, that corresponds to the eigenvalue of unity.

Having calculated the probabilities p_i that are associated with the states of the FSM closure, the source encoder calculates the probabilities q_j that are associated with the states of the context–tree source. Let all the FSM closure states with a suffix that equals the same context–tree source state be referred to as the derived states of the context–tree source state. The probability associated with a context–tree source state equals the sum of the probabilities associated with its derived states.

After calculating the expected positions of the transition points between the segments of the BWT output sequence, the encoder uses the weight–and–index source

code for binary memoryless sources to independently encode the bit sequences between the expected transition points. The encoder concatenates the codewords of the bit sequences to produce the source–coded sequence. The first field of codeword i represents the weight w_i of segment i of the BWT output sequence, where $i \in \{1, 2, \ldots |\mathcal{S}|\}$. Each weight is encoded using the binary–coded representation of an integer. The first field of codeword i has a length of

$$l_{1,i}(\mathbf{x}) = \lceil \log_2(\overline{n}_i + 1) \rceil \tag{6.93}$$

bits, as the weight w_i may assume any value in the set $\{0, 1, \ldots \overline{n}_i\}$. The second field of codeword i represents the index of segment i in the ordered set of all sequences with the same length and weight as the segment. This field has a length of

$$l_{2,i}(\mathbf{x}) = \left\lceil \log_2 \binom{\overline{n}_i}{w_i} \right\rceil \tag{6.94}$$

bits. The source encoder encodes the BWT index of each n–bit source sequence as part of the source–coded sequence. Each BWT index is encoded using the conventional binary–coded representation of an integer. Each encoded index has a length of

$$l_3(\mathbf{x}) = \lceil \log_2(n + 1) \rceil \tag{6.95}$$

bits, as the index may assume $n+1$ distinct values with the addition of the `EOF` symbol to the reversed source sequence.

It is assumed that the source decoder of the type–two nonuniversal weight–and–index source code has apriori knowledge regarding the bit distribution associated with each state of the context–tree source. It is therefore unnecessary to encode the expected positions of the transition points between the segments of the BWT output sequence, as the source decoder may calculate the expected positions of the transition points.

The universal weight–and–index code A block diagram of the universal weight–and–index source encoder for sequences from binary context–tree sources is presented in figure 6.8 on page 175. The source encoder follows the same initial steps to encode a source sequence as the encoders of the nonuniversal weight–and–index source codes. To encode a source sequence, the source encoder first reverses the sequence and appends the `EOF` symbol to the reversed sequence. It applies the BWT to the reversed source sequence that is appended with the `EOF` symbol. The source encoder next removes the `EOF` symbol from the BWT output sequence.

The source encoder of the universal weight–and–index source code for binary context–tree sources has no apriori knowledge of the source structure (i.e. its minimum suffix set) or the bit distributions that are associated with the source states. The source encoder requires apriori knowledge of the source in order to analytically derive the expected positions of the transition points between the segments in the BWT output sequence.

One approach to encoding the BWT output sequences in a universal fashion involves the estimation of the number of transition points between the i.i.d. bit segments of each BWT output sequence, as well as the positions of the transition points. The

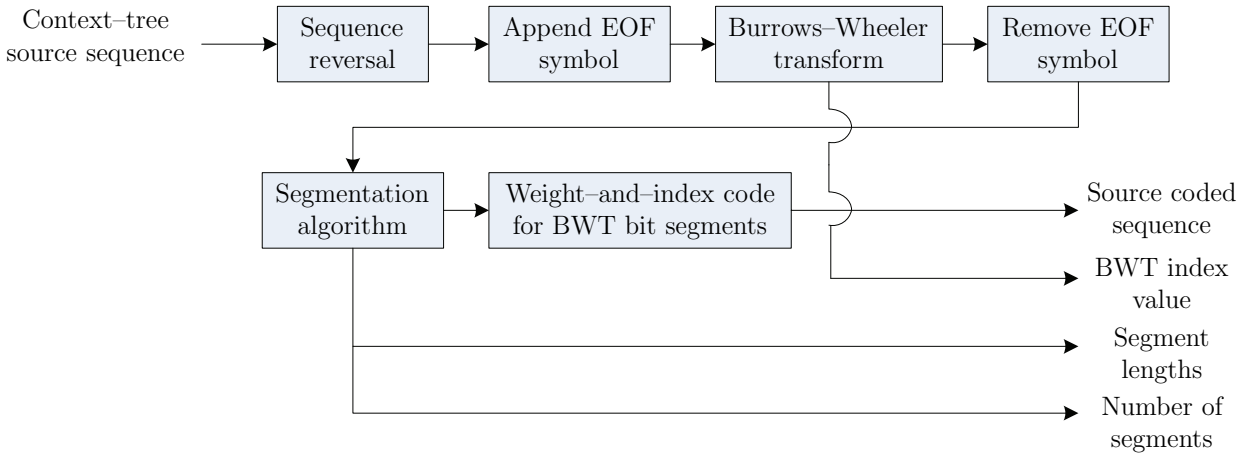


Figure 6.8: A block diagram of the universal weight–and–index source encoder for binary context–tree sources.

source encoder of the universal weight–and–index source code for binary context–tree sources follows this approach. After estimating the number of transition points in a certain BWT output sequence, as well as their positions in the sequence, the source encoder independently encodes the bit sequences between the estimated positions of the transition points. The source encoder uses the weight–and–index source code for binary memoryless sources to encode these sequences.

The universal weight–and–index source code for binary context–tree sources estimates the positions of the transition points between the segments of each BWT output sequence using a segmentation algorithm. The segmentation algorithm locates regions within each BWT output sequence where the empirical distribution of the local bits undergoes a significant change. It assumes that transition points are located within these regions. The segmentation algorithm divides each BWT output sequence into segments according to the estimated positions of the transition points. These segments should ideally correspond to the actual segments of i.i.d. bits in each BWT output sequence.

Shamir et. al. [159] proposed a universal source code for piecewise–stationary source sequences that uses a segmentation algorithm. The purpose of the segmentation algorithm, as used by the source code of Shamir et. al., is to divide each source sequence into its stationary segments. This segmentation algorithm was incorporated into the universal weight–and–index source code for binary context–tree sources, and is summarized in what follows.

The segmentation algorithm proposed by Shamir et. al. [159] estimates the number and positions of the transition points in a BWT output sequence in two stages. The BWT output sequence is divided into blocks of length $k_1(n)$ bits during the first stage, where n is the length of the BWT output sequence. It is divided into blocks of length $k_2(n)$ bits during the second stage, where $k_2(n) < k_1(n)$. The blocks of length $k_1(n)$ bits and $k_2(n)$ bits are referred to as the level–one and level–two blocks, respectively.

The segmentation algorithm processes the level–one blocks during its first stage to obtain rough estimates of the locations of possible transition points in a BWT output

sequence [159]. It refines its estimates of the transition point locations during the second stage by processing the level–two blocks.

The segmentation algorithm calculates a metric for each level–one block as it processes these blocks during the first stage, with the exception of the first block and the last block of the BWT output sequence [159]. The metric of a block is a measure of the dissimilarity between the empirical distribution of the bits in its preceding block and the empirical distribution of the bits in its succeeding block.

Let the frequency counts of the zero–valued and nonzero–valued bits in the ith level–one block of the BWT output sequence be denoted by $N_i(0)$ and $N_i(1)$, where[12] $i \in \{1, 2, \ldots n/k_1(n)\}$. The metric of the rth level–one block, $M(r)$, is expressed as [159]

$$M(r) = \hat{H}(r-1, r+1) - \frac{1}{2}\hat{H}(r-1) - \frac{1}{2}\hat{H}(r+1), \tag{6.96}$$

where

$$\hat{H}(i) = -\sum_{x \in \{0,1\}} \frac{N_i(x)}{k_1(n)} \log_2 \left(\frac{N_i(x)}{k_1(n)} \right) \tag{6.97}$$

and

$$\hat{H}(i,j) = -\sum_{x \in \{0,1\}} \frac{N_i(x) + N_j(x)}{2k_1(n)} \log_2 \left(\frac{N_i(x) + N_j(x)}{2k_1(n)} \right), \tag{6.98}$$

with $r \in \{2, 3, \ldots n/k_1(n) - 1\}$ and $i, j \in \{1, 2, \ldots n/k_1(n)\}$. This metric is an asymptotically optimal statistic for the purpose of determining whether or not the bits of block $r-1$ and block $r+1$ share the same probability distribution.

The segmentation algorithm calculates the metrics of all the level–one blocks of the BWT output sequence, except its first block and its last block [159]. It proceeds by iteratively retaining certain level–one blocks, and rejecting other level–one blocks. At the start of each iteration, the segmentation algorithm finds the level–one block with the largest metric among all the level–one blocks that were neither retained nor rejected in a previous iteration. This level–one block is retained, and the four blocks of the BWT output sequence that neighbour this retained block (i.e. two blocks on either side of the retained block) are rejected. The segmentation algorithm continues to iterate in this fashion until all the level–one blocks of the BWT output sequence, with the exception of its first block and its last block, are either retained or rejected. At this point, the first stage of the segmentation algorithm is complete.

The segmentation algorithm independently processes each of the retained level–one blocks during its second stage [159]. It places a transition point within each retained level–one block, or within one of its two neighbouring level–one blocks (i.e. its preceding or succeeding block). The segmentation algorithm divides each retained level–one block and its two neighbouring level–one blocks into level–two blocks of length $k_2(n)$ bits in order to determine an appropriate position for a transition point within these blocks. The algorithm calculates the metric[13] (equation 6.96) of each level–two block

[12] It is assumed that $n/k_1(n)$ is an integer in order to simplify the discussion.

[13] The block length $k_1(n)$ of equations 6.97 and 6.98 is replaced by $k_2(n)$ when the segmentation algorithm evaluates these equations during its second stage. The terms $N_j(0)$ and $N_j(1)$ of equations 6.97 and 6.98 denote the frequency counts of the jth level–two block when the segmentation algorithm evaluates these equations during its second stage.

within each retained level–one block and its two neighbouring level–one blocks (with the exception of the first level–two block of the preceding neighbour, and the last level–two block of the succeeding neighbour). It finds the level–two block with the largest metric in each retained level–one block and its two neighbouring level–one blocks, and places a transition point in the center of the level–two block with the largest metric.

The accuracy of the segmentation algorithm is dependent on the lengths of the level–one and level–two blocks. Shamir et. al. [159] found that accurate segmentation may be performed by using level–one and level–two blocks with lengths of

$$k_1(n) = (\log_2(n))^{1+\mu} \tag{6.99}$$

and

$$k_2(n) = (\log_2 \log_2(n))^{1+\gamma} \tag{6.100}$$

bits, where $\mu, \gamma > 0$.

The lengths of the source–coded sequences that the universal weight–and–index source encoder for binary context–tree sources produces are subsequently derived. The derivation is carried out under the assumption that the segmentation algorithm is perfectly accurate (i.e. the estimated number of transition points and their estimated positions equal the actual number of transition points and their actual positions). This assumption does not hold in practice — the derived lengths only approximate the source–coded sequence lengths of a practical implementation of the source code. An analytical investigation of the effect that imperfect segmentation has on the lengths of the source–coded sequences is beyond the scope of this thesis.

Let $\mathcal{S}$ denote the state set of the binary context–tree source, and m denote the length of the longest state. Each BWT output sequence of the universal weight–and–index source code has a maximum of $|\mathcal{S}| + m$ i.i.d. bit segments if the code is used to encode sequences from the binary context–tree source. The encoder of the universal weight–and–index source code for binary context–tree sources independently applies the weight–and–index source code for binary memoryless sources to each segment of each BWT output sequence. A source–coded sequence is produced by concatenating the codewords that correspond to the segments of a single BWT output sequence.

Consider a BWT output sequence with a total of C i.i.d. bit segments, where $C \leq |\mathcal{S}| + m$. The first field of codeword i of its source–coded sequence represents the weight of the ith segment of the BWT output sequence, where $i \in \{1, 2, \ldots C\}$. This field has a length of

$$l_{1,i}(\mathbf{x}) = \lceil \log_2(n_i + 1) \rceil \tag{6.101}$$

bits, where n_i denotes the length of the ith segment of the BWT output sequence. The second field of codeword i represents the index of the ith segment of the BWT output sequence in the ordered set of all sequences with the same length and weight as the segment. This field has a length of

$$l_{2,i}(\mathbf{x}) = \left\lceil \log_2 \binom{n_i}{w_i} \right\rceil \tag{6.102}$$

bits, where w_i denotes the weight of the ith segment of the BWT output sequence.

The source encoder encodes the BWT index that is associated with the BWT output sequence as part of the source–coded sequence. The BWT index is encoded using the conventional binary–coded representation of an integer. The codeword of the index has a length of

$$l_3(\mathbf{x}) = \lceil \log_2(n+1) \rceil \tag{6.103}$$

bits, as the BWT index may assume $n+1$ distinct values with the addition of the `EOF` symbol to the reversed source sequence.

In order to guarantee the unique decodability of the source–coded sequence, the source encoder encodes both the number of i.i.d. bit segments in the BWT output sequence, as well the lengths of the segments. These codewords form part of the source–coded sequence. The length of each segment in the BWT output sequence is encoded using the conventional binary–coded representation of an integer. Each segment length is encoded in a total of $\lceil \log_2(n) \rceil$ bits, as a segment length may assume a value in the set $\{1, 2, \ldots n\}$. The C segment lengths are encoded in a total of

$$l_4(\mathbf{x}) = (C-1) \lceil \log_2(n) \rceil \tag{6.104}$$

bits, as the final segment length may be derived from the known sequence length n. The number of segments in the BWT output sequence is encoded in a total of

$$l_5(\mathbf{x}) = \lceil \log_2(n) \rceil \tag{6.105}$$

bits, as the source decoder has no apriori knowledge regarding the state set of the binary context–tree source.

6.2.1.2 Definition of the source code

Three functions are introduced prior to the mathematical definition of the weight–and–index source code for sequences from binary memoryless sources. The first function $\mathcal{C}_{n,m} : \mathbb{B}^n \times \mathbb{B}^m \mapsto \mathbb{B}^{n+m}$ concatenates an n–bit sequence and an m–bit sequence. The second function $\mathcal{D}_{n,m} : \mathbb{B}^{n+m} \mapsto \mathbb{B}^n \times \mathbb{B}^m$ divides an $(n+m)$–bit sequence into an n–bit sequence (its initial n bits) and an m–bit sequence (its final m bits). The third function $\mathcal{H}_n : \mathbb{B}^m \mapsto \mathbb{B}^n$, for all $m \geq n$, maps each m–bit sequence in its domain to the n–bit prefix of the sequence.

The weight–and–index variable–length source code for sequences $\mathbf{x}^n$ from binary memoryless sources is defined in what follows. Let the encoder of the source code correspond to the function $g_3 : \mathbb{B}^n \mapsto \mathbb{B}^*$. The function g_3 is defined as

$$\begin{aligned} \mathbf{c}_{\mathbf{x}} &= g_3(\mathbf{x}) \\ &= \mathcal{C}_{l_1,l_2}\Big(\beta_{l_1}\big(w\big), \beta_{l_2}\big(f(\mathbf{x})\big)\Big), \end{aligned} \tag{6.106}$$

where $\mathbf{c}_{\mathbf{x}}$ denotes the codeword of the source sequence $\mathbf{x}^n$, w denotes its weight, and where $l_1 \triangleq l_1(\mathbf{x})$ and $l_2 \triangleq l_2(\mathbf{x})$ are defined in equations 6.79 and 6.81. The functions f and β_q are defined in equations 6.6 and 6.8.

Mathematical definitions of the universal and nonuniversal weight–and–index source codes for sequences from binary context–tree sources are not provided, as these source

codes use the weight–and–index source code for sequences from binary memoryless sources. The unique decodability of the universal and nonuniversal weight–and–index source codes for binary context–tree sources depends on the unique decodability of the weight–and–index source code for binary memoryless sources, which is subsequently proved.

6.2.1.3 Proof of unique decodability

A proof of the unique decodability of the weight–and–index source code for sequences from binary memoryless sources is provided in this section.

Theorem 6.2.1. *The weight–and–index source code for sequences from binary memoryless sources is uniquely decodable if its source decoder is aware of the source sequence length n.*

Proof. It is proved that an arbitrary source sequence $\mathbf{x}^n$ can be recovered from its codeword $\mathbf{c_x}$. The source decoder does not have apriori knowledge of the length of each codeword that is to be decoded, as the source encoder produces variable–length codewords. The proof is therefore carried out without apriori knowledge of the codeword length.

The proof of the source code's unique decodability relies on the source decoder being able to calculate the length $l_1(\mathbf{x})$ of the codeword's first field, as defined in equation 6.79. The source decoder is able to calculate the length of the codeword's first field, as it is aware of the source sequence length n. The first field of the codeword (i.e. the binary–coded source sequence weight) is obtained by evaluating the expression

$$\beta_{l_1}(w) = \mathcal{H}_{l_1}(\mathbf{c_x}). \tag{6.107}$$

The function β_{l_1} in equation 6.107 may be replaced with the function B_{l_1}, which was defined below equation 6.8 on page 142. This replacement is valid, as $0 \leq w \leq n$ and

$$\begin{aligned} 2^{l_1} &= 2^{\lceil \log_2(n+1) \rceil} \\ &\geq 2^{\log_2(n+1)} \\ &\geq n+1, \end{aligned} \tag{6.108}$$

which proves that $n \leq 2^{l_1} - 1$ and that $w \leq 2^{l_1} - 1$. The weight of the source sequence is recovered by evaluating the expression

$$w = B_{l_1}^{-1}(\mathcal{H}_{l_1}(\mathbf{c_x})). \tag{6.109}$$

The length $l_2(\mathbf{x})$ of the codeword's second field is obtained by evaluating equation 6.81 using the recovered source sequence weight.

The function $\mathcal{C}_{l_1,l_2}$ can be inverted, as both $l_1(\mathbf{x})$ and $l_2(\mathbf{x})$ are known. The inverse of this function is the function $\mathcal{D}_{l_1,l_2}$. It follows that the equation

$$\Big(\beta_{l_1}\big(w\big), \beta_{l_2}\big(f(\mathbf{x})\big)\Big) = \mathcal{D}_{l_1,l_2}(\mathbf{c_x}) \tag{6.110}$$

holds. The function β_{l_2} in equation 6.110 may be replaced with the function B_{l_2}, as

$$\begin{aligned} l_2(\mathbf{x}) &= \left\lceil \log_2 \binom{n}{w} \right\rceil \\ &\geq \log_2 \binom{n}{w} \end{aligned} \tag{6.111}$$

and

$$\begin{aligned} f(\mathbf{x}) &\leq \binom{n}{w} - 1 \\ &\leq 2^{l_2} - 1. \end{aligned} \tag{6.112}$$

The source sequence $\mathbf{x}^n$ is recovered by evaluating the expression

$$\mathbf{x} = f^{-1}(B_{l_2}^{-1}(\mathbf{z})), \tag{6.113}$$

where $\mathbf{z} = \beta_{l_2}(f(\mathbf{x}))$. The inverse of the function f was defined in equation 6.29 on page 146. □

6.2.1.4 Theoretical performance

The average per–codeword redundancies of two weight–and–index source codes are derived in this section. The universal source code for sequences from binary memoryless sources, as well as the universal source code for sequences from binary context–tree sources are considered. The source coding of finite–length source sequences and sequences with lengths that tend to infinity is investigated.

Performance w.r.t. binary memoryless sources A bound on the average per–codeword redundancy of the universal weight–and–index source code for sequences from binary memoryless sources is derived in this section. This source code was introduced in section 6.2.1.1 on page 167.

Consider a finite–length n–bit sequence $\mathbf{x}^n$ from a binary memoryless source, and assume that $n > 1$. Let w denote the weight of the sequence. The length of the codeword that is assigned to the source sequence equals

$$\begin{aligned} l(\mathbf{x}) &= l_1(\mathbf{x}) + l_2(\mathbf{x}) \\ &= \lceil \log_2(n+1) \rceil + \left\lceil \log_2 \binom{n}{w} \right\rceil \end{aligned} \tag{6.114}$$

bits. An upper bound on the length of the codeword is derived as

$$l(\mathbf{x}) < \log_2(n+1) + \log_2 \binom{n}{w} + 2. \tag{6.115}$$

A more informative upper bound is derived by first using Stirling's approximation of the factorial of an integer to derive an upper bound on the logarithm of a binomial coefficient. An initial bound on the logarithm of the binomial coefficient $\binom{n}{w}$, which

holds for all $w \in \{1, 2, \ldots n-1\}$ with $n > 1$, is derived. This initial bound is generalized to the case where $w \in \{0, 1, \ldots n\}$.

Consider the case where the source sequence weight w is an element of the set $\{1, 2, \ldots n-1\}$, and where $n > 1$. Stirling's approximation of the factorial $n!$,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\lambda_n}, \tag{6.116}$$

which holds for a certain number λ_n that satisfies the inequality

$$\frac{1}{12n+1} < \lambda_n < \frac{1}{12n}, \tag{6.117}$$

is used to derive an upper bound and a lower bound on the base–two logarithm of the factorial of an integer. The upper bound on the base–two logarithm of the factorial $n!$ is derived as

$$\log_2(n!) < \frac{1}{2}\log_2(2\pi n) + n\log_2\left(\frac{n}{e}\right) + \frac{1}{12n}\log_2(e), \tag{6.118}$$

and the lower bound is derived as

$$\log_2(n!) > \frac{1}{2}\log_2(2\pi n) + n\log_2\left(\frac{n}{e}\right) + \frac{1}{12n+1}\log_2(e). \tag{6.119}$$

An upper bound on the base–two logarithm of a binomial coefficient is derived by using the upper and lower bounds on the logarithm of the factorial of an integer. The upper bound on the base–two logarithm of a binomial coefficient is derived as

$$\begin{aligned}
\log_2\binom{n}{w} &= \log_2\left(\frac{n!}{w!(n-w)!}\right) \\
&< \frac{1}{2}\log_2\left(\frac{n}{2\pi w(n-w)}\right) + n\log_2\left(\frac{n}{e}\right) - w\log_2\left(\frac{w}{e}\right) \\
&\quad - (n-w)\log_2\left(\frac{n-w}{e}\right) + \left(\frac{1}{12n} - \frac{1}{12w+1} - \frac{1}{12(n-w)+1}\right)\log_2(e) \\
&< \frac{1}{2}\log_2\left(\frac{n}{2\pi w(n-w)}\right) + \frac{1}{12n}\log_2(e) + nh\left(\frac{w}{n}\right) \\
&< \frac{1}{2}\log_2\left(\frac{1}{2\pi n\hat{p}(1-\hat{p})}\right) + \frac{1}{12n}\log_2(e) + nh(\hat{p}),
\end{aligned} \tag{6.120}$$

where $h(\cdot)$ denotes the binary entropy function, and

$$\hat{p} \triangleq \frac{w}{n}. \tag{6.121}$$

The inequality

$$\frac{1}{n} \leq \hat{p} \leq \frac{n-1}{n}, \tag{6.122}$$

which holds for $w \in \{1, 2, \ldots n-1\}$ with $n > 1$, is used to derive the inequality

$$\log_2(\hat{p}(1-\hat{p})) \geq \log_2\left(\frac{n-1}{n^2}\right). \tag{6.123}$$

This inequality is substituted into equation 6.120 to derive the upper bound

$$\log_2 \binom{n}{w} < \log_2 \left(\frac{n}{\sqrt{2\pi n(n-1)}} \right) + \frac{1}{12n} \log_2(e) + nh(\hat{p}), \tag{6.124}$$

which holds for $w \in \{1, 2, \ldots n-1\}$ with $n > 1$.

In order to generalize the initial bound of equation 6.124 to the case where $w \in \{0, 1, \ldots n\}$, it is determined that the function

$$v(n) = \log_2 \left(\frac{n}{\sqrt{2\pi n(n-1)}} \right) + \frac{1}{12n} \log_2(e) \tag{6.125}$$

is a monotonically decreasing function with a minimum value of

$$\lim_{n \to \infty} v(n) = -\log_2(\sqrt{2\pi}). \tag{6.126}$$

As the equation

$$\log_2 \binom{n}{w} = 0 \tag{6.127}$$

holds for all $n > 1$ if $w \in \{0, n\}$, and

$$\log_2(e) > \log_2(\sqrt{2\pi}), \tag{6.128}$$

the upper bound

$$\log_2 \binom{n}{w} < \log_2 \left(\frac{n}{\sqrt{2\pi n(n-1)}} \right) + \left(\frac{1}{12n} + 1 \right) \log_2(e) + nh(\hat{p}) \tag{6.129}$$

holds for $w \in \{0, 1, \ldots n\}$ with $n > 1$.

An upper bound on the length of the codeword that is assigned to the source sequence $\mathbf{x}^n$ is derived by substituting equation 6.129 into equation 6.115. This upper bound is derived as

$$l(\mathbf{x}) < \log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \left(\frac{1}{12n} + 1 \right) \log_2(e) + nh(\hat{p}) + 2, \tag{6.130}$$

where $n > 1$ and $w \in \{0, 1, \ldots n\}$.

The average length $L(\mathbf{X}^n)$ of a codeword assigned to a random source sequence $\mathbf{X}^n$ has the upper bound

$$\begin{aligned} L(\mathbf{X}^n) &= \sum_{\mathbf{x} \in \{0,1\}^n} \Pr(\mathbf{x}) l(\mathbf{x}) \\ &< \sum_{w=0}^{n} \Pr(w) \left[\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \left(\frac{1}{12n} + 1 \right) \log_2(e) + nh\left(\frac{w}{n} \right) + 2 \right], \end{aligned} \tag{6.131}$$

where $\Pr(w) \triangleq \Pr(W = w)$ denotes the probability that a random source sequence has a weight of w bits. The binary entropy function is a concave function — Jensen's inequality is therefore used to derive the inequality

$$\begin{aligned}\sum_{w=0}^{n} \Pr(w) h\left(\frac{w}{n}\right) &\leq h\left(\sum_{w=0}^{n} \Pr(w)\frac{w}{n}\right) \\ &\leq h(p), \end{aligned} \tag{6.132}$$

where the last step follows from the fact that the expected weight of a source sequence equals np bits, with $p \triangleq \Pr(X_i = 1)$.

An upper bound on the average codeword length is derived by substituting equation 6.132 into equation 6.131. This upper bound is derived as

$$L(\mathbf{X}^n) < \log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \left(\frac{1}{12n} + 1\right)\log_2(e) + nh(p) + 2. \tag{6.133}$$

The upper bound on the average codeword length is used to derive an upper bound on average per–codeword redundancy $R(\mathbf{X}^n)$. This upper bound is derived as

$$\begin{aligned} R(\mathbf{X}^n) &= L(\mathbf{X}^n) - H(\mathbf{X}^n) \\ &= L(\mathbf{X}^n) - nh(p) \\ &< \log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \left(\frac{1}{12n} + 1\right)\log_2(e) + 2. \end{aligned} \tag{6.134}$$

The limit of the normalized average per–codeword redundancy, as the source sequence length n tends to infinity, is derived as follows. The average per–codeword redundancy satisfies the inequality

$$0 \leq R(\mathbf{X}^n) < \log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \left(\frac{1}{12n} + 1\right)\log_2(e) + 2 \tag{6.135}$$

for all $n > 1$. The average per–codeword redundancy is therefore expressed as

$$R(\mathbf{X}^n) = K_n q(n), \tag{6.136}$$

where

$$q(n) = \log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \left(\frac{1}{12n} + 1\right)\log_2(e) + 2 \tag{6.137}$$

and $0 \leq K_n < 1$ for all $n > 1$. The limit

$$\begin{aligned} \lim_{n\to\infty} \frac{1}{n} q(n) &= \lim_{n\to\infty} \frac{1}{n} \log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) \\ &= \lim_{n\to\infty} \frac{1}{n} \log_2\left(\frac{(n+1)^2}{(n-1)}\right) - \lim_{n\to\infty} \frac{1}{2n}\log_2(2\pi) \\ &= \lim_{n\to\infty} \frac{1}{\ln(2)} \frac{n-1}{n^2+2n+1}\left(\frac{n^2-2n-3}{(n-1)^2}\right) \\ &= \lim_{n\to\infty} \frac{1}{\ln(2)} \frac{n-3}{n^2-1} \\ &= 0 \end{aligned} \tag{6.138}$$

is used to prove that

$$\begin{aligned}\lim_{n\to\infty}\frac{1}{n}R(\mathbf{X}^n) &= \lim_{n\to\infty}\frac{1}{n}K_n q(n)\\ &= 0.\end{aligned} \tag{6.139}$$

The limit of the normalized average per–codeword redundancy, as the source sequence length tends to infinity, is therefore equal to zero. It is concluded that the weight–and–index source code for binary memoryless sources is a universal source code.

Performance w.r.t. binary context–tree sources A bound on the average per–codeword redundancy of the universal weight–and–index source code for sequences from binary context–tree sources is derived in this section. This source code was introduced in section 6.2.1.1 on page 174. The derivation of the bound is carried out under the assumption that the segmentation algorithm of the source code predicts the number of transition points between the segments of each BWT output sequence, as well as the positions of the transition points, with perfect accuracy.

Consider a binary context–tree source with a state set $\mathcal{S}$, and let the length of the longest source context be denoted by m. Each BWT output sequence of the source encoder has a maximum of $|\mathcal{S}|+m$ i.i.d. bit segments when the encoder is used to encode sequences from the source. Each of the additional segments of each BWT output sequence (i.e. each segment in addition to the $|\mathcal{S}|$ segments per sequence that correspond to the states of the source) has a length of one bit.

Let C denote the true number of i.i.d. bit segments in the BWT output sequence that is produced when a finite–length source sequence $\mathbf{x}^n$ is encoded, where $n>1$. An upper bound on the length of the codeword assigned to the sequence $\mathbf{x}^n$ is derived as

$$\begin{aligned}l(\mathbf{x}) &= \sum_{i=1}^{C}\left(l_{1,i}(\mathbf{x})+l_{2,i}(\mathbf{x})\right)+\sum_{j=3}^{5}l_j(\mathbf{x})\\ &\leq \sum_{i=1}^{|\mathcal{S}|}\left(\lceil\log_2(n_i+1)\rceil+\left\lceil\log_2\binom{n_i}{w_i}\right\rceil\right)+\lceil\log_2(n+1)\rceil+C\lceil\log_2(n)\rceil\\ &\quad +m,\end{aligned} \tag{6.140}$$

where it is assumed that the additional i.i.d. bit segments are numbered from $|\mathcal{S}|+1$ to C, and where the field lengths $l_{1,i}(\mathbf{x})$ to $l_5(\mathbf{x})$ are defined in equations 6.101 to 6.105. The last step of equation 6.140 follows from the fact that each additional i.i.d. bit segment is assigned a codeword of one bit (refer to equations 6.101 and 6.102).

A more informative upper bound is derived using the upper bound on the codeword length of the weight–and–index source code for sequences from binary memoryless sources (equation 6.130). The length of the codeword produced by the encoder of the weight–and–index source code for sequences from binary memoryless sources, when the encoder is used to encode segment i of the BWT output sequence, has the upper bound

$$l_{1,i}(\mathbf{x})+l_{2,i}(\mathbf{x}) < \log_2\left(\frac{n_i(n_i+1)}{\sqrt{2\pi n_i(n_i-1)}}\right)+\left(\frac{1}{12n_i}+1\right)\log_2(e)+n_i h\left(\frac{w_i}{n_i}\right)+2, \tag{6.141}$$

under the condition that the segment length n_i exceeds one bit. It is assumed that this condition is satisfied for all $i \in \{1, 2, \ldots |S|\}$ in order to derive an initial bound on the codeword length of the universal weight–and–index source code for binary context–tree sources. The initial bound is generalized to the case where $n_i \geq 1$ after its derivation.

The initial bound on the codeword length of the weight–and–index source code for binary context–tree sources is derived using the inequality $\log_2(n) < \log_2(n+1)$. The initial bound is derived as

$$\begin{aligned} l(\mathbf{x}) &< \sum_{i=1}^{|\mathcal{S}|} \left(\log_2 \left(\frac{n_i(n_i+1)}{\sqrt{2\pi n_i(n_i-1)}} \right) + \left(\frac{1}{12n_i} + 1 \right) \log_2(e) + n_i h\left(\frac{w_i}{n_i} \right) + 2 \right) \\ &\quad + (C+1)\log_2(n+1) + C + 1 + m, \end{aligned} \tag{6.142}$$

where it is assumed that n_i exceeds one bit for all $i \in \{1, 2, \ldots |S|\}$. The function

$$v(n_i) = \log_2 \left(\frac{n_i(n_i+1)}{\sqrt{2\pi n_i(n_i-1)}} \right) \tag{6.143}$$

is a monotonically increasing function of the segment length n_i. Due to the monotonic nature of the function v, and the fact that $n_i \leq n$, the inequality

$$v(n_i) \leq v(n) \tag{6.144}$$

holds for all $i \in \{1, 2, \ldots |\mathcal{S}|\}$. This inequality is used to derive the upper bound

$$\begin{aligned} l(\mathbf{x}) &< \sum_{i=1}^{|\mathcal{S}|} \left(\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \frac{13}{12} \log_2(e) + n_i h\left(\frac{w_i}{n_i} \right) + 2 \right) \\ &\quad + (C+1)\log_2(n+1) + C + 1 + m, \end{aligned} \tag{6.145}$$

where it is assumed that n and n_i exceed one bit for all $i \in \{1, 2, \ldots |S|\}$.

It is subsequently proved that the bound of equation 6.145 remains valid if one or more of the segments in the BWT output sequence consist of a single bit. Each segment that consists of a single bit is assigned a codeword of one bit by the encoder of the weight–and–index source code for binary memoryless sources (refer to equations 6.101 and 6.102). Therefore, in order to prove that the bound of equation 6.145 remains valid, it is proved that the function

$$z(n) = \log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \frac{13}{12} \log_2(e) + n_i h\left(\frac{w_i}{n_i} \right) + 2 \tag{6.146}$$

exceeds one bit for all $n > 1$ and all $n_i \geq 1$.

The inequality

$$\begin{aligned} v(n) &= \log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) \\ &> 0 \end{aligned} \tag{6.147}$$

holds for all $n > 1$, as $v(n)$ is a monotonically increasing function and $v(2) > 0$. The inequality

$$h\left(\frac{w_i}{n_i}\right) \geq 0 \tag{6.148}$$

holds for all $n_i \geq 1$, as the range of the binary entropy function is nonnegative. The inequalities of equations 6.147 and 6.148, as well as the fact that

$$\frac{13}{12}\log_2(e) > 0, \tag{6.149}$$

are used to derive the inequality

$$\begin{aligned} z(n) &= \log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + n_i h\left(\frac{w_i}{n_i}\right) + 2 \\ &> 1. \end{aligned} \tag{6.150}$$

This inequality proves that the inequality of equation 6.145 holds for all BWT output sequences with segment lengths n_i, where $n_i \geq 1$ for all $i \in \{1, 2, \ldots |S|\}$.

In order to derive an upper bound on the average length of the codewords assigned to sequences from the binary context–tree source, the inequality

$$\begin{aligned} \sum_{i=1}^{|\mathcal{S}|} n_i h\left(\frac{w_i}{n_i}\right) &\leq \sum_{i=1}^{C} n_i h\left(\frac{w_i}{n_i}\right) \\ &\leq n\hat{H}_{\mathbf{x}}(\mathcal{X}) \end{aligned} \tag{6.151}$$

is derived. The term $\hat{H}_{\mathbf{x}}(\mathcal{X})$ of equation 6.151 denotes the entropy rate of the empirical distribution of the source sequence $\mathbf{x}^n$. An upper bound on the average codeword length $L(\mathbf{X}^n)$ is derived by substituting the inequality of equation 6.151 into the upper bound of equation 6.145, and averaging with respect to the source sequence probability. The upper bound is derived as

$$\begin{aligned} L(\mathbf{X}^n) &< \sum_{\mathbf{x}\in\{0,1\}^n} \Pr(\mathbf{x}) \Bigg[\sum_{i=1}^{|\mathcal{S}|} \left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2 \right) \\ &\quad + n\hat{H}_{\mathbf{x}}(\mathcal{X}) + (C+1)\log_2(n+1) + C + 1 + m \Bigg] \\ &< \sum_{i=1}^{|\mathcal{S}|} \left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2 \right) + nH(\mathcal{X}) \\ &\quad + (C+1)\log_2(n+1) + C + 1 + m, \end{aligned} \tag{6.152}$$

where $H(\mathcal{X})$ denotes the entropy rate of the binary context–tree source. The last step of equation 6.152 follows from Jensen's inequality (refer to reference [10]). The average length of the codewords assigned to sequences from the binary context–tree source is therefore bounded as

$$\begin{aligned} L(\mathbf{X}^n) &< |\mathcal{S}| \left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2 \right) + nH(\mathcal{X}) \\ &\quad + (C+1)\log_2(n+1) + C + 1 + m. \end{aligned} \tag{6.153}$$

The normalized average per–codeword redundancy of the universal weight–and–index source code for binary context–tree sources, with respect to the entropy rate of the source[14], is bounded as

$$\begin{aligned} R'(\mathbf{X}^n) &= \frac{1}{n}L(\mathbf{X}^n) - H(\mathcal{X}) \\ &< \frac{|\mathcal{S}|}{n}\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right) \\ &\quad + \frac{(C+1)\log_2(n+1)}{n} + \frac{C+1+m}{n}. \end{aligned} \tag{6.154}$$

An upper bound on the normalized average per–codeword redundancy, as the source sequence length n tends to infinity, is derived as follows. The normalized average per–codeword redundancy satisfies the inequality

$$0 \leq R'(\mathbf{X}^n) < q(n) \tag{6.155}$$

for all $n > 1$, where

$$\begin{aligned} q(n) &= \frac{|\mathcal{S}|}{n}\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right) \\ &\quad + \frac{(C+1)\log_2(n+1)}{n} + \frac{C+1+m}{n}. \end{aligned} \tag{6.156}$$

The normalized average per–codeword redundancy is therefore expressed as

$$R'(\mathbf{X}^n) = K_n q(n), \tag{6.157}$$

where $0 \leq K_n < 1$ for all $n > 1$. Consider the limit

$$\lim_{n\to\infty} q(n) = \lim_{n\to\infty} \frac{|\mathcal{S}|}{n}\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \lim_{n\to\infty}\frac{(C+1)\log_2(n+1)}{n}. \tag{6.158}$$

Equation 6.138 is used to prove that

$$\begin{aligned} \lim_{n\to\infty} \frac{|\mathcal{S}|}{n}\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) &= \lim_{n\to\infty}\frac{1}{n}\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right). \\ &= 0. \end{aligned} \tag{6.159}$$

The second term on the right–hand side of equation 6.158 is simplified as

$$\begin{aligned} \lim_{n\to\infty}\frac{(C+1)\log_2(n+1)}{n} &= \lim_{n\to\infty}\frac{C+1}{\ln(2)(n+1)} \\ &= 0. \end{aligned} \tag{6.160}$$

It follows that the inequality

$$\begin{aligned} \lim_{n\to\infty} R'(\mathbf{X}^n) &= \lim_{n\to\infty} K_n q(n) \\ &= 0 \end{aligned} \tag{6.161}$$

holds. It is concluded that the proposed weight–and–index source code for binary context–tree sources is universal.

[14]Effros et. al. [10] defined the redundancy of BWT–based source codes for context–tree sources with respect to the entropy rate of the source. This definition is used in the performance analysis of each source code for context–tree sources that is proposed in this thesis.

6.2.1.5 Practical results

Practical implementations of the weight–and–index source codes for binary memoryless sources and binary context–tree sources were used to encode sequences from their respective sources. The source coding results for each source type are presented in this section.

Binary memoryless sources The weight–and–index source code for binary memoryless sources was used to encode sequences from several sources in two configurations. This source code is referred to as the proposed source code in what follows. Both configurations of the proposed source code are discussed in this section.

Configuration 1: Fixed source sequence length The first configuration of the proposed source code involved the source coding of fixed–length sequences from a single binary memoryless source. The source parameter p (i.e. the probability of the source producing a nonzero–valued bit) was changed for each set of simulation trials. The practical implementation of the source code had no apriori knowledge of the source parameter p. The parameters and quantities associated with the proposed source code in the first configuration are presented in table 6.6 on page 189.

The source coding results associated with the proposed source code in the first configuration are presented in figure 6.9 on page 190. This figure contains a plot of the average codeword length of the practical implementation of the proposed source code, as a function of the source parameter p. It includes a curve that represents the per–sequence entropy of the source.

Figure 6.9 reveals that the average codeword length of the practical implementation of the proposed source code approaches the per–sequence entropy of the binary memoryless source over the entire range of the source parameter p. This observation implies that the source code may be used to encode 500–bit sequences from any stationary binary memoryless source with a reasonable degree of effectiveness, and without prior knowledge of the source parameter p.

The average per–codeword redundancy of the practical implementation of the proposed source code is plotted in figure 6.10 on page 191 as function of the source parameter p. The figure includes a curve that represents the analytically–derived upper bound on the average per–codeword redundancy of the proposed source code (i.e. equation 6.134). The figure reveals that the average redundancy of the practical implementation does not exceed its upper bound. This observation suggests that the proposed source code was correctly implemented.

Figure 6.10 reveals that the average per–codeword redundancy of the practical implementation does not vary significantly over the source parameter interval $0.2 \leq p \leq 0.8$. The codewords of the practical implementation become marginally less redundant if the implementation is used to encode sequences from a source with a per–symbol entropy that approaches one bit. As the per–symbol entropy of the source approaches zero bits, the codewords of the practical implementation become more redundant. This observation is motivated by considering the fact that the first codeword field (i.e. the binary–coded source sequence weight) becomes more redundant when the implementation is used to encode source sequences that are more redundant. The first codeword

Table 6.6: Parameters and quantities associated with the first configuration of the weight–and–index source code for binary memoryless sources.

Parameter / Quantity	Symbol	Value	Unit
Source sequence length	n	500	bits
Prob. nonzero–valued source bit	p	0 – 1	—
Per–sequence source entropy	$H(\mathbf{X}^n)$	0 – 500	bits
Trial runs per value of p	—	50 000	trials

field has a fixed length, and is optimal when used to represent equiprobable sequence weights — the weight distribution of highly–redundant source sequences is biased, however.

Figure 6.10 includes curves that represent upper bounds on the redundancy of certain nonuniversal source codes, assuming the codes were used to encode the same sequences as the proposed source code. The nonuniversal source codes consist of a Huffman code that was extended to 500 bits, and an arithmetic code that independently encodes sequences of 500 bits. It is assumed that both of these source codes had apriori knowledge of the source parameter p. The upper bound on the redundancy of the extended Huffman code was derived using equation 2.2 on page 9, and the upper bound on the redundancy of the arithmetic code was derived using equation 2.8 on page 16. Neither of the nonuniversal source codes were implemented.

The figure reveals that both the extended Huffman code and the arithmetic code produce codewords that are, on average, significantly less redundant than the codewords of the proposed source code. The extended Huffman code and the arithmetic code are not universal, in contrast to the proposed source code — both the Huffman code and the arithmetic code require apriori knowledge of the source parameter p. The computational complexity of the extended Huffman code's construction is exceedingly high, and renders the source code impractical.

Configuration 2: Variable source sequence length The second configuration of the proposed source code involved the source coding of variable–length sequences from three binary memoryless sources. Each binary memoryless source had a distinct source parameter p that remained fixed. The parameters and quantities associated with the proposed source code in the second configuration are presented in table 6.7 on page 192.

The source coding results associated with the proposed source code in the second configuration are presented in figure 6.11 on page 193. This figure contains a plot of the normalized average per–codeword redundancy of the practical implementation of the proposed source code, as a function of the source sequence length. It also contains curves that represent the normalized average redundancy of two nonuniversal source codes, as well as a curve that represents an analytically–derived upper bound on the normalized average redundancy of the proposed source code. The upper bound on the normalized average redundancy of the proposed source code was derived by dividing equation 6.134 on page 183 by the source sequence length n.

The normalized average redundancy of the practical implementation of the pro-

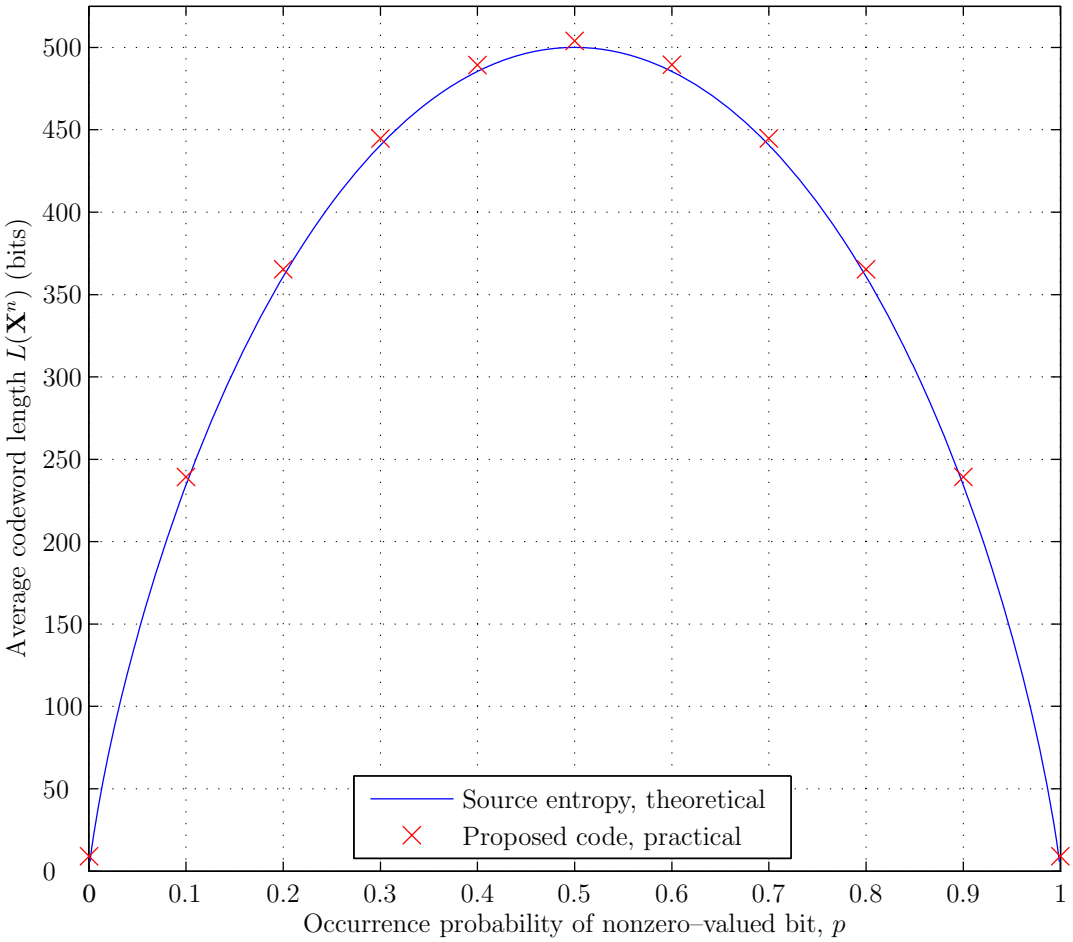


Figure 6.9: Average codeword length of the weight–and–index source code for binary memoryless sources in the first configuration, as a function of the source parameter p. The figure includes a curve that represents the per–sequence entropy of the source.

posed source code is represented by three distinct curves in figure 6.11 — each curve corresponds to the source coding of sequences from one of the three binary memoryless sources. The figure reveals that the normalized average redundancy of the practical implementation decreases w.r.t. the source sequence length, as expected for a universal source code. The three curves that represent the proposed code's normalized average redundancy start to overlap as the length of the source sequences is increased.

All of the curves of figure 6.11 that represent the redundancy of the practical implementation of the proposed source code lie below the curve that represents the analytically–derived upper bound on the redundancy of the proposed source code. The curves that represent the normalized average redundancy of the practical implementation of the proposed source code have approximately the same gradient as the curve that represents the upper bound (as the source sequence length approaches 1000 bits). These observations suggest that the source code was correctly implemented.

Figure 6.11 includes curves that represent upper bounds on the redundancy of a nonuniversal extended Huffman code, as well as a nonuniversal arithmetic code. Neither

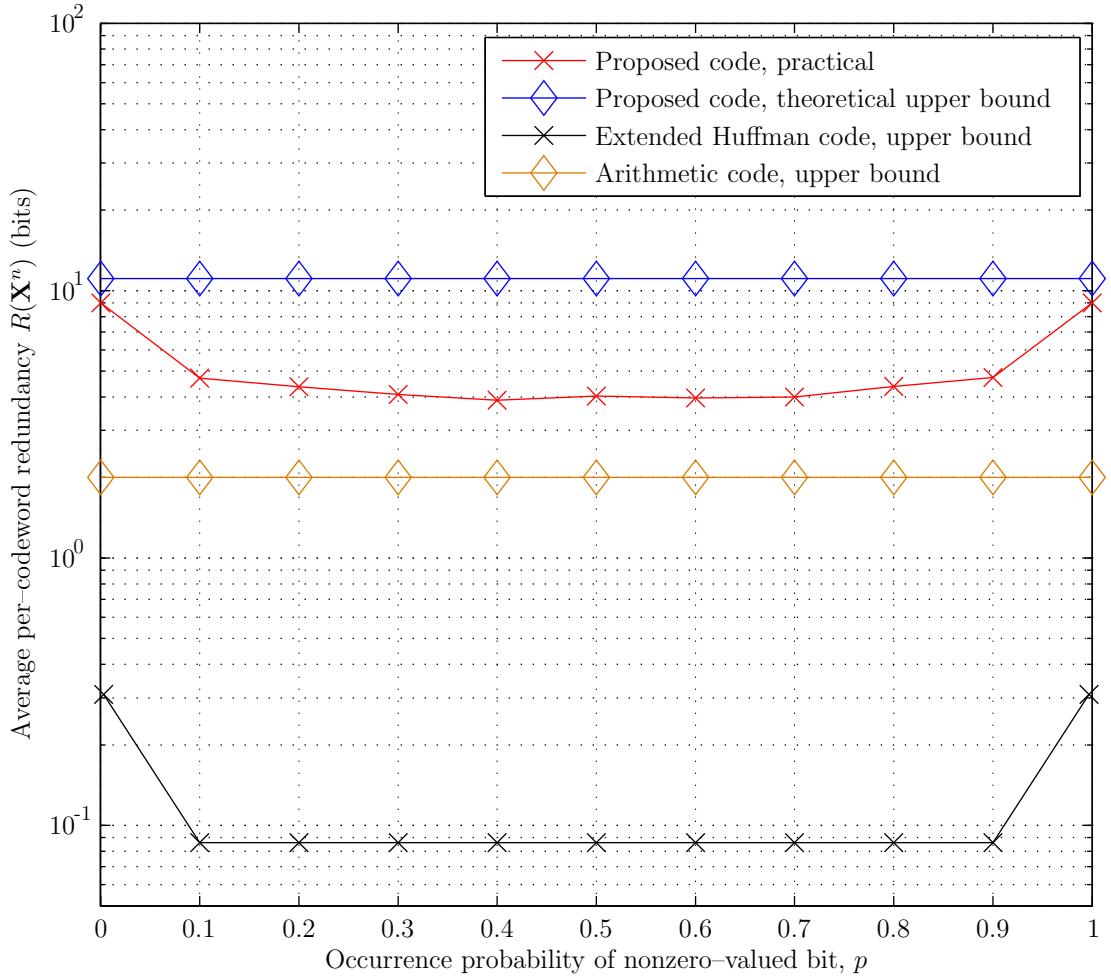


Figure 6.10: Normalized average per–codeword redundancy of the weight–and–index source code for binary memoryless sources in the first configuration, as a function of the source parameter p. The figure includes curves that represent upper bounds on the redundancy of several source codes.

of the nonuniversal source codes were implemented. It is assumed that the nonuniversal source codes were used to encode the same sequences as the proposed source code, and that these source codes had apriori knowledge of each source parameter p. The Huffman code was extended to the length of the source sequence in each case. It is assumed that the arithmetic code was used to independently encode sequences of the relevant length.

Figure 6.11 reveals that the proposed source code would be outperformed by both the nonuniversal extended Huffman code and the nonuniversal arithmetic code if used to encode source sequences longer than 50 bits. Both the extended Huffman code and the arithmetic require apriori knowledge of the source parameter p, however. The extended Huffman code is also impractical, as the computational complexity of its construction is exceedingly high.

Table 6.7: Parameters and quantities associated with the second configuration of the weight–and–index source code for binary memoryless sources.

Parameter / Quantity	Symbol	Value	Unit
Source sequence length	n	$10 - 10^3$	bits
Prob. nonzero–valued src. bit, src. 1	p	0.2	—
Prob. nonzero–valued src. bit, src. 2	p	0.5	—
Prob. nonzero–valued src. bit, src. 3	p	0.7	—
Per–symbol src. entropy, src. 1	$H(X)$	0.7219	bits
Per–symbol src. entropy, src. 2	$H(X)$	1	bits
Per–symbol src. entropy, src. 3	$H(X)$	0.8813	bits

Binary context–tree sources The three weight–and–index source codes for binary context–tree sources were implemented and used to encode sequences from two stationary binary context–tree sources. The two binary context–tree sources that were considered are referred to as the high–entropy source and the medium–entropy source. These sources are defined in what follows.

Source 1: The high–entropy source The first binary context–tree source that was considered is referred to as the high–entropy source, as its entropy rate is approximately equal to one bit per source bit. The tree of the high–entropy source is presented in figure 6.12 on page 194.
The high–entropy source has a total of five states. Its state set $\mathcal{S}$ is defined as

$$\mathcal{S} = \{\mathtt{00}, \mathtt{01}, \mathtt{100}, \mathtt{101}, \mathtt{11}\}. \tag{6.162}$$

Let m denote the length of the longest context that is associated with the high–entropy source. It follows that

$$m = 3, \tag{6.163}$$

as the source has no context longer than three bits.

Let the states of the high–entropy source be numbered from one to five in the order

$$\begin{aligned} \mathbf{s}_1 &= \mathtt{00} \\ \mathbf{s}_2 &= \mathtt{100} \\ \mathbf{s}_3 &= \mathtt{01} \\ \mathbf{s}_4 &= \mathtt{11} \\ \mathbf{s}_5 &= \mathtt{101}. \end{aligned} \tag{6.164}$$

The distribution of the bits that the high–entropy source produces in each of its states is presented in table 6.8 on page 195.

The FSM closure of the high–entropy binary context–tree source is subsequently derived. The state set $\mathcal{S}'$ of the FSM closure is derived as

$$\mathcal{S}' = \{\mathtt{00}, \mathtt{10}, \mathtt{001}, \mathtt{101}, \mathtt{11}\}. \tag{6.165}$$

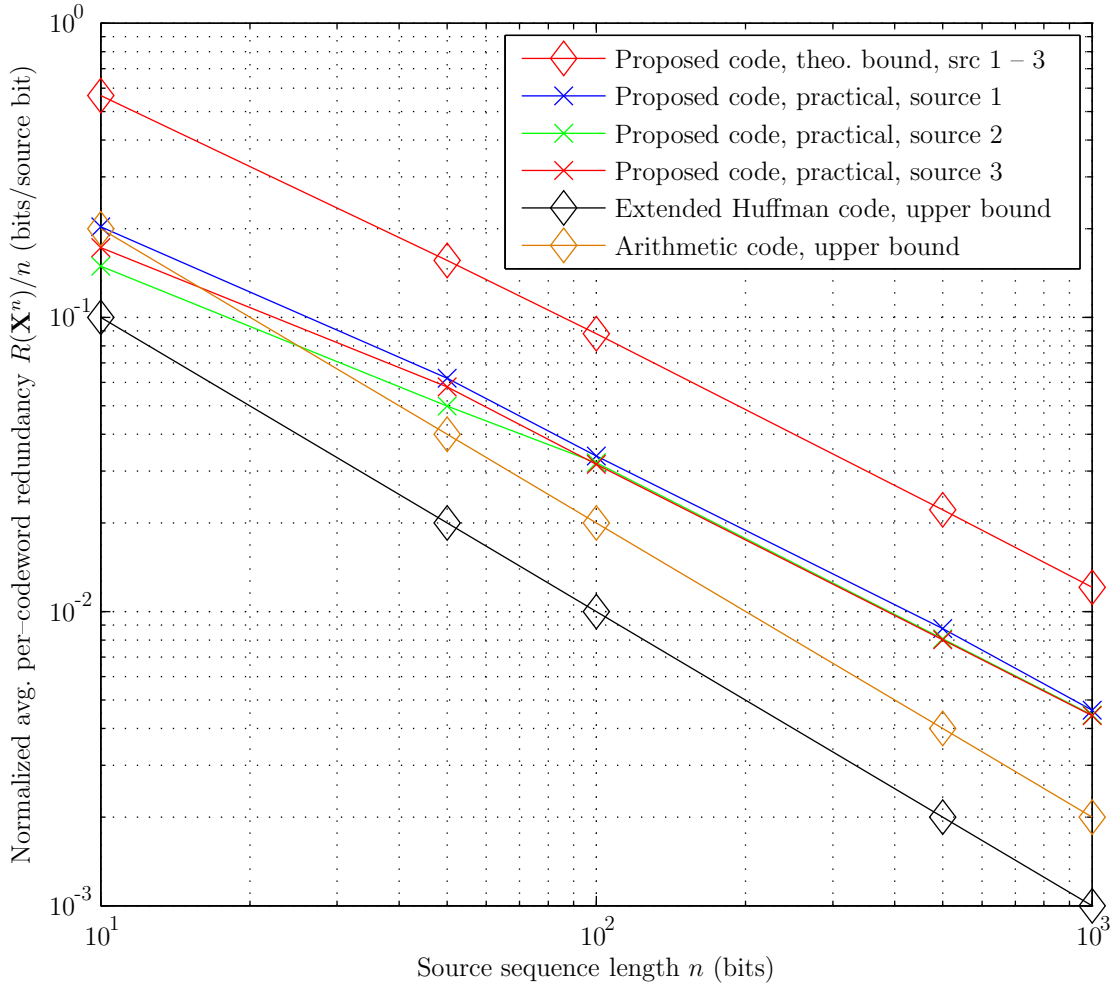


Figure 6.11: Normalized average per–codeword redundancy of the weight–and–index source code for binary memoryless sources in the second configuration, as a function of the source sequence length n. The figure includes curves that represent upper bounds on the redundancy of several source codes.

Let the states of the FSM closure of the high–entropy binary context–tree source be numbered from one to five in the order

$$\begin{aligned} \mathbf{s}'_1 &= \texttt{00} \\ \mathbf{s}'_2 &= \texttt{001} \\ \mathbf{s}'_3 &= \texttt{10} \\ \mathbf{s}'_4 &= \texttt{11} \\ \mathbf{s}'_5 &= \texttt{101}. \end{aligned} \tag{6.166}$$

The state–transition diagram of the FSM closure of the high–entropy binary context–tree source is presented in figure 6.13 on page 194.

The state–transition probability matrix $\mathbf{S}$ of the FSM closure of the high–entropy

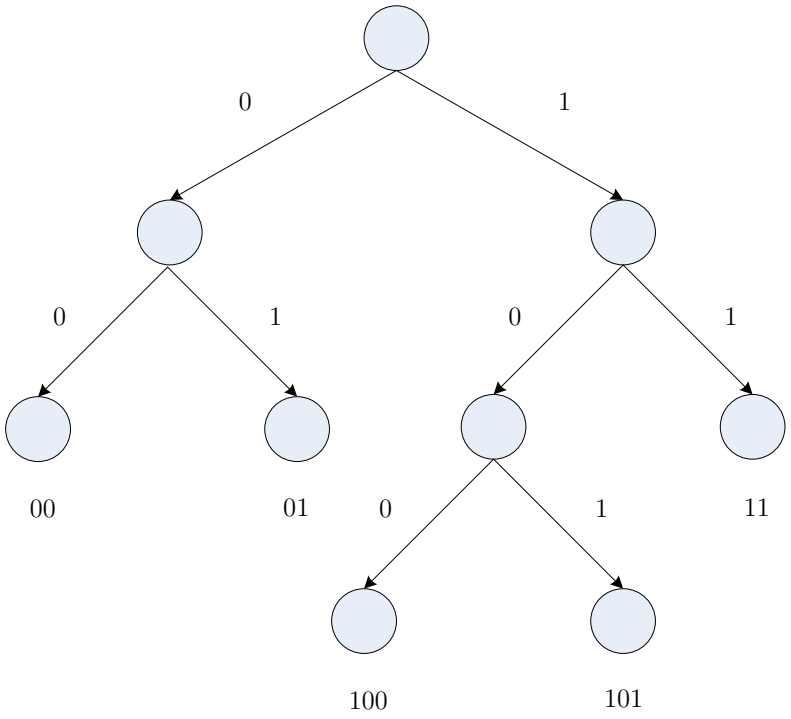


Figure 6.12: The tree of the high–entropy source. The leaves of the tree represent the states of the source.

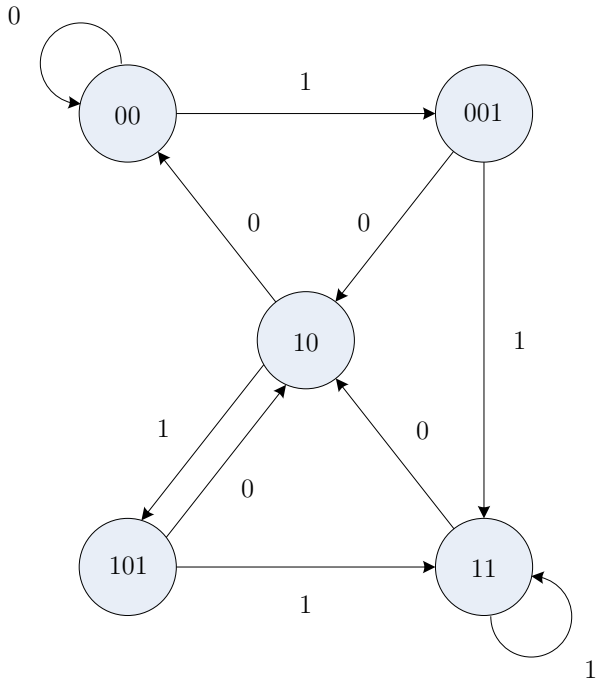


Figure 6.13: The state–transition diagram of the FSM closure of the high–entropy binary context–tree source.

Table 6.8: The bit distributions that are associated with the states of the high–entropy source.

	$\Pr(X = 0\|\mathbf{s})$	$\Pr(X = 1\|\mathbf{s})$
$\mathbf{s} = \mathbf{s}_1$	0.6926	0.3074
$\mathbf{s} = \mathbf{s}_2$	0.4365	0.5635
$\mathbf{s} = \mathbf{s}_3$	0.4823	0.5177
$\mathbf{s} = \mathbf{s}_4$	0.7822	0.2178
$\mathbf{s} = \mathbf{s}_5$	0.3650	0.6350

binary context–tree source is derived as

$$\begin{aligned}\mathbf{S} &= \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,|\mathcal{S}'|} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,|\mathcal{S}'|} \\ \vdots & \vdots & \ddots & \\ p_{|\mathcal{S}'|,1} & p_{|\mathcal{S}'|,2} & & p_{|\mathcal{S}'|,|\mathcal{S}'|} \end{pmatrix} \\ &= \begin{pmatrix} 0.6926 & 0.3074 & 0 & 0 & 0 \\ 0 & 0 & 0.4365 & 0.5635 & 0 \\ 0.4823 & 0 & 0 & 0 & 0.5177 \\ 0 & 0 & 0.7822 & 0.2178 & 0 \\ 0 & 0 & 0.3650 & 0.6350 & 0 \end{pmatrix}, \end{aligned} \tag{6.167}$$

where $p_{y,z}$ denotes the probability of a transition to state $\mathbf{s}'_z$ of the FSM closure, conditioned on the source being in state $\mathbf{s}'_y$.

Equation 6.92 is used to derive the state probability vector of the FSM closure of the high–entropy binary context–tree source. The state probability vector $\mathbf{P}$ is derived as

$$\begin{aligned}\mathbf{P} &= [p_1, p_2, \ldots p_{|\mathcal{S}'|}]^T \\ &= [0.3618, 0.1112, 0.2306, 0.1770, 0.1194]^T, \end{aligned} \tag{6.168}$$

where p_y denotes the probability of the FSM closure being in state $\mathbf{s}'_y$.

The entropy $H_y(X)$ associated with the distribution of the bits that are produced in state $\mathbf{s}'_y$ of the FSM closure is calculated using the equation

$$H_y(X) = -\sum_{z=1}^{|\mathcal{S}'|} p_{y,z} \log_2(p_{y,z}). \tag{6.169}$$

The entropy rate $H(\mathcal{X})$ of the FSM closure of the binary context–tree source is calculated as

$$\begin{aligned} H(\mathcal{X}) &= \sum_{y=1}^{|\mathcal{S}'|} p_y H_y(X) \\ &= 0.9092 \end{aligned} \tag{6.170}$$

bits per source bit. The entropy rate of the high–entropy binary context–tree source equals the entropy rate of its FSM closure.

Table 6.9: The bit distributions that are associated with the states of the medium–entropy source.

	$\Pr(X = 0\|\mathbf{s})$	$\Pr(X = 1\|\mathbf{s})$
$\mathbf{s} = \mathbf{s}_1$	0.2502	0.7498
$\mathbf{s} = \mathbf{s}_2$	0.3880	0.6120
$\mathbf{s} = \mathbf{s}_3$	0.5920	0.4080
$\mathbf{s} = \mathbf{s}_4$	0.0524	0.9476
$\mathbf{s} = \mathbf{s}_5$	0.4874	0.5126

Source 2: The medium–entropy source The second binary context–tree source that was considered is referred to as the medium–entropy source, as its entropy rate is approximately equal to 0.5 bits per source bit. It has the same tree as the high–entropy binary context–tree source (figure 6.12 on page 194), and therefore the same state set as the high–entropy source (equations 6.162 and 6.164). It is assumed that the states of the medium–entropy source are numbered in the same manner as the states of the high–entropy source.

The distribution of the bits that the medium–entropy source produces in each of its states differs from the corresponding distribution of the high–entropy source. The distribution of the bits that the medium–entropy source produces in each state is presented in table 6.9 on page 196.

The FSM closure of the medium–entropy source has the same state set as the FSM closure of the high–entropy source (equations 6.165 and 6.166), as well as the same state–transition diagram (figure 6.13 on page 194). Let the states of the FSM closure of the medium–entropy source be numbered in the same manner as the states of the FSM closure of the high–entropy source. The state–transition probability matrix[15] $\mathbf{S}$ of the FSM closure of the medium–entropy binary context–tree source is derived as

$$\mathbf{S} = \begin{pmatrix} 0.2502 & 0.7498 & 0 & 0 & 0 \\ 0 & 0 & 0.3880 & 0.6120 & 0 \\ 0.5920 & 0 & 0 & 0 & 0.4080 \\ 0 & 0 & 0.0524 & 0.9476 & 0 \\ 0 & 0 & 0.4874 & 0.5126 & 0 \end{pmatrix}. \tag{6.171}$$

The state probability vector $\mathbf{P}$ of the FSM closure of the medium–entropy binary context–tree source is derived as

$$\mathbf{P} = [0.0577, 0.0432, 0.0730, 0.7963, 0.0298]^T \tag{6.172}$$

using equation 6.92. The entropy rate $H(\mathcal{X})$ of the FSM closure of the medium–entropy binary context–tree source is calculated as

$$H(\mathcal{X}) = 0.4256 \tag{6.173}$$

bits per source bit using equations 6.169 and 6.170. The entropy rate of the medium–entropy binary context–tree source equals the entropy rate of its FSM closure.

[15] Refer to equation 6.167 for the definition of the state–transition probability matrix.

Table 6.10: Parameters and quantities associated with the high–entropy and medium–entropy binary context–tree sources.

Parameter / Quantity	Symbol	Value	Unit
Source sequence length	n	256 – 8192	bits
Entropy rate, high–entropy src.	$H(\mathcal{X})$	0.9092	bits per source bit
Entropy rate, medium–entropy src.	$H(\mathcal{X})$	0.4256	bits per source bit

Table 6.11: Block lengths and parameters associated with the segmentation algorithm of the universal weight–and–index source code for binary context–tree sources.

Source sequence length n (bits)	Level–one block length $k_1(n)$ (bits)	Level–two block length $k_2(n)$ (bits)	Parameter μ	Parameter γ
256	75	15	1.0763	1.4650
512	100	20	1.0959	1.5966
1024	126	21	1.1004	1.5359
2048	150	25	1.0896	1.5936
4096	180	30	1.0898	1.6640
8192	210	30	1.0847	1.5994

Source coding results The three weight–and–index source codes for binary context–tree sources were implemented and used to encode sequences from the high–entropy and medium–entropy sources. These source codes are the type–one and type–two nonuniversal weight–and–index source codes (introduced in section 6.2.1.1 on pages 169 and 172), as well as the universal weight–and–index source code (introduced in section 6.2.1.1 on page 174). The length of the source sequences that were encoded was changed for each set of simulation trials in order to characterize the source code redundancy as a function of the source sequence length.

The parameters and quantities associated the high–entropy and medium–entropy binary context–tree sources are presented in table 6.10 on page 197. Table 6.11 on page 197 contains the block lengths and parameters that are associated with the practical implementation of the segmentation algorithm, which is part of the universal weight–and–index source code for binary context–tree sources. The parameters μ and γ, as defined in equations 6.99 and 6.100, were assigned values by trial and error.

The source coding results are presented in figure 6.14 on page 198. This figure contains a plot of the normalized average per–codeword redundancy of the practical implementation of each source code, as a function of the source sequence length. The figure also contains a curve that represents the analytically–derived upper bound on the redundancy of the universal weight–and–index source code. This bound was derived under the assumption that the segmentation algorithm of the source code is perfectly accurate, and is expressed in equation 6.154 on page 187.

Figure 6.14 reveals that all the curves which represent the redundancy of the source codes' practical implementations lie below the curve that represents the upper bound

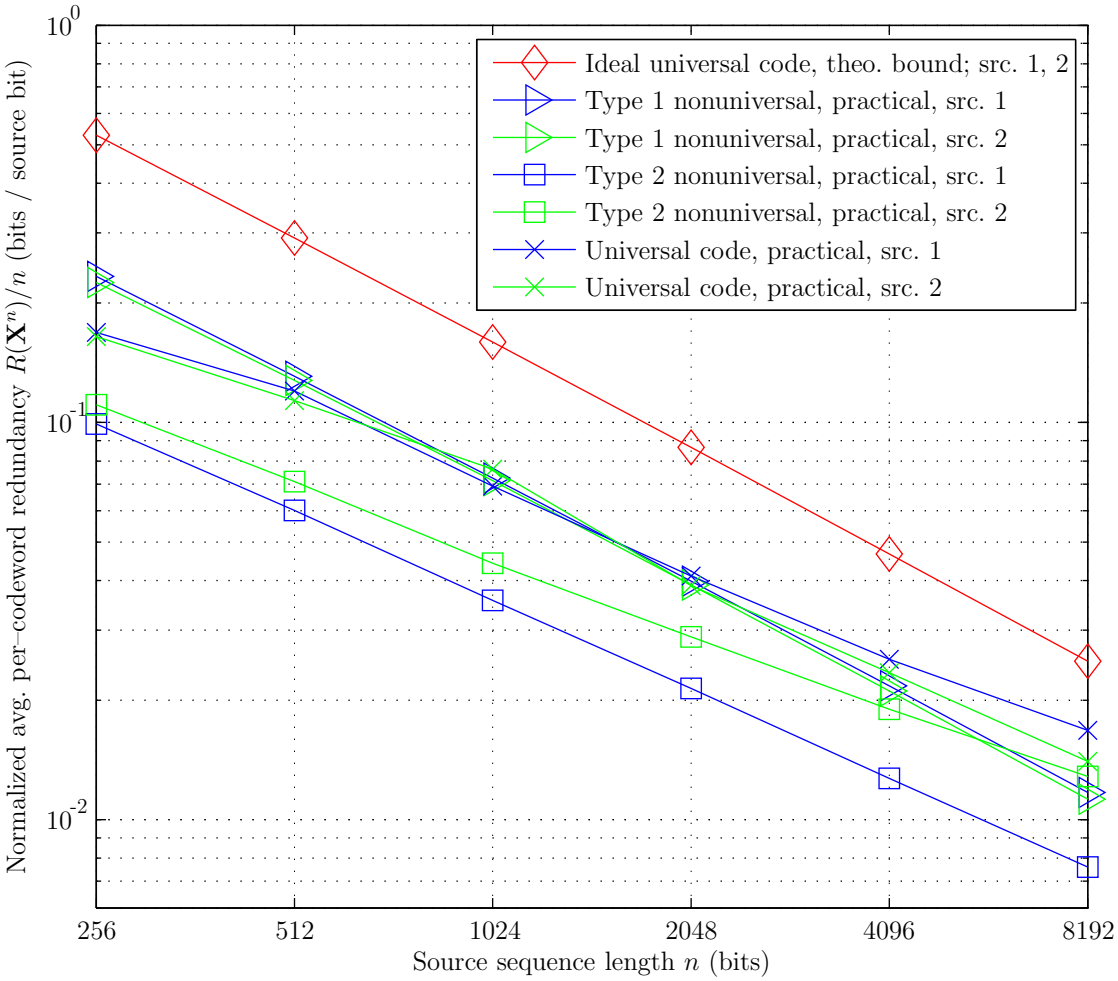


Figure 6.14: The normalized average per–codeword redundancy of the nonuniversal and universal weight–and–index source codes for binary context–tree sources, as a function of the source sequence length n. Source one is the high–entropy source, and source two is the medium–entropy source.

on the redundancy of the universal weight–and–index source code with a perfectly accurate segmentation algorithm. The curves that represent the redundancy of the type–one nonuniversal source code decrease at approximately the same rate w.r.t. the source sequence length as the curve that represents the upper bound. This observation suggests that the type–one nonuniversal source code was correctly implemented, as the redundancy of the type–one nonuniversal source code is nearly equal to the redundancy of the universal source code with a perfectly accurate segmentation algorithm (compare equations 6.84 to 6.87 with equations 6.101 to 6.105).

The redundancy of the type–two nonuniversal weight–and–index source code, when the source code is used to encode sequences from both the high–entropy and medium–entropy sources, does not decrease at the same rate w.r.t. the source sequence length as the upper bound on the redundancy of the universal source code with a perfectly accurate segmentation algorithm. It appears that the approach of independently en-

coding the sequences between the expected positions of the transition points in the BWT output sequence may cause the redundancy of the source code to decrease at a lesser rate in certain cases.

The curves that represent the redundancy of the type–one nonuniversal source code nearly overlap the curves that represent the redundancy of the universal source code, assuming that source sequences of between 512 bits and 2048 bits were encoded. This observation is indicative of the effectiveness of the universal source code, as the redundancy of the type–one nonuniversal source code is approximately equal to the redundancy of the universal source code with a perfectly accurate segmentation algorithm.

The redundancy of the universal source code decreases at a lesser rate w.r.t. the source sequence length than the upper bound and the redundancy of the type–one nonuniversal source code, assuming that source sequences in excess of 2048 bits were encoded. This observation implies that the segmentation algorithm of the universal code does not produce perfect estimates of the positions of the transition points in longer BWT output sequences. The performance of the source code may be improved by searching for values of the parameters $k_1(n)$ and $k_2(n)$ that yield more accurate estimates of the positions of the transition points. It was found that the accuracy of the segmentation algorithm is very sensitive to changes in the values of these parameters.

Figure 6.14 reveals that the universal weight–and–index source code has a smaller codeword redundancy than the type–one nonuniversal weight–and–index source code when used to encode sequences of 256 bits. It was found that the segmentation algorithm of the universal source code underestimated the number of segments in shorter BWT output sequences (on average). This implies that fewer segment lengths have to be encoded — as this overhead forms a significant part of the codewords of shorter source sequences, the effectiveness of the code is improved.

6.2.2 The weight–and–index variable–length code for q–ary sources

The weight–and–index variable–length source code for q–ary sources is a generalization of the weight–and–index variable–length source code for binary sources, which was introduced in section 6.2.1 on page 166. This source code may be used to encode symbol sequences from certain q–ary sources[16] in a universal fashion, where $q \geq 2$. The weight–and–index source code for q–ary sources, with q equal to two, is identical to the weight–and–index source code for binary sources, as introduced in section 6.2.1.

The weight–and–index source code for q–ary sources may be used to separately encode sequences from q–ary memoryless sources or q–ary context–tree sources. The structure of the source code depends on whether it encodes sequences from a memoryless source or a context–tree source, and whether it encodes sequences in a universal or a nonuniversal fashion. The weight–and–index source code for q–ary memoryless sources is a universal source code. Three weight–and–index source codes for q–ary context–tree sources are defined — two of these source codes are nonuniversal, and the remaining code is universal.

[16] A q–ary source produces symbols from an alphabet of q distinct symbols.

6.2.2.1 Derivation of the source code

The derivation of the weight–and–index variable–length source code for q–ary sources is divided into two sections. The first section concerns the derivation of the universal weight–and–index source code for q–ary memoryless sources. The second section concerns the derivation of the nonuniversal and universal weight–and–index source codes for q–ary context–tree sources.

It is assumed that the integer q is a power of two in order to simplify the subsequent discussion. The integer m', which is defined as

$$m' = \log_2(q), \tag{6.174}$$

is used in the remainder of this section.

Q–ary memoryless sources The encoder of the weight–and–index source code for q–ary memoryless sources numbers the symbols of the alphabet from zero to $q-1$, and assigns a unique binary word of $\log_2(q)$ bits to each distinct alphabet symbol. It transforms each source sequence of n symbols into a sequence of n binary words by replacing each q–ary symbol of the source sequence with the binary word assigned to it.

The encoder of the weight–and–index source code for q–ary memoryless sources divides each sequence of n binary words into shorter subsequences of bits, and uses the weight–and–index source code for binary memoryless sources to independently encode each subsequence. The steps that the source encoder follows to divide a sequence of binary words into shorter subsequences are summarized in what follows.

The source encoder iteratively generates a total of m' blocks of bits. Each block contains n bits of the (nm')–bit sequence of binary words. The jth n–bit block is obtained by concatenating the jth bits of the binary words in the sequence of binary words, where $j \in \{1, 2, \ldots m'\}$. The bits are concatenated in the same order in which their binary words appear in the sequence of binary words. Each of the m' blocks of n bits is divided further by the source encoder.

Consider the jth n–bit block that the encoder of the weight–and–index source code for q–ary memoryless sources generates, where $j \in \{1, 2, \ldots m'\}$. Each bit of this block may be associated with the $(j-1)$–bit prefix of the m'–bit binary word from which it originated[17]. The source encoder divides the bits of the jth n–bit block it generated into sets, so that all of the bits in the same set are associated with the same distinct $(j-1)$–bit prefix. It follows that the source encoder produces a maximum of 2^{j-1} nonempty sets of bits during the division of the jth n–bit block it generated.

The source encoder concatenates the bits in each nonempty set that it produced during the division of the jth n–bit block, thereby producing a maximum of 2^{j-1} binary subsequences. The bits of each set are concatenated in the same order in which they appeared in the jth n–bit block. Each binary subsequence is encoded independently from the remaining binary subsequences using the weight–and–index source code for binary memoryless sources.

[17] In the case where $j = 1$, all of the bits of the block are associated with the same prefix. This prefix is the zero–length, null sequence.

After encoding all the subsequences of the jth n–bit block, the encoder of the weight–and–index source code for q–ary memoryless sources concatenates the subsequences' codewords in the same order in which the subsequences were generated. The sequence of concatenated codewords may be interpreted as the codeword assigned to the jth n–bit block generated by the source encoder. After generating a codeword for each of the m' blocks of n bits, the source encoder concatenates these codewords in the same order in which their corresponding n–bit blocks were generated. The sequence of concatenated codewords may be interpreted as the codeword of the entire source sequence.

In order to justify the weight–and–index source code for q–ary memoryless sources, the per–symbol entropy $H(X)$ of a q–ary memoryless source is expressed in terms of joint and conditional entropy functions. Let the random binary word assigned to the random q–ary source symbol X be denoted by $\mathbf{B}^{m'} = \{B_1, B_2, \ldots B_{m'}\}$, where $B_i \in \mathbb{B}$ for all $i \in \{1, 2, \ldots m'\}$. The per–symbol entropy of the q–ary memoryless source may be expressed as [160]

$$\begin{aligned} H(X) &= H(B_1, B_2, \ldots B_{m'}) \\ &= H(B_1) + \sum_{j=2}^{m'} H(B_j|B_1, B_2, \ldots B_{j-1}). \end{aligned} \tag{6.175}$$

The expression for the per–symbol entropy of the q–ary memoryless source is expanded with respect to its conditional entropy terms as

$$\begin{aligned} H(X) &= H(B_1) + \sum_{j=2}^{m'} H(B_j|\mathbf{B}_1^{j-1}) \\ &= H(B_1) + \sum_{j=2}^{m'} \sum_{\mathbf{b}_1^{j-1} \in \{0,1\}^{j-1}} \Pr(\mathbf{b}_1^{j-1}) H(B_j|\mathbf{b}_1^{j-1}), \end{aligned} \tag{6.176}$$

where $\Pr(\mathbf{b}_1^{j-1}) \triangleq \Pr(\mathbf{B}_1^{j-1} = \mathbf{b}_1^{j-1})$.

Suppose that the ideal variable–length source code, which was introduced in section 6.1.1.6 on page 149, is used to encode the subsequences of the jth n–bit block generated by the source encoder. The codeword that is assigned to the subsequence associated with the prefix $\mathbf{b}_1^{j-1}$ would have a normalized average length of $H(B_j|\mathbf{b}_1^{j-1})$ bits per subsequence bit. As an average of $n\Pr(\mathbf{b}_1^{j-1})$ binary words assigned to a random n–symbol source sequence have the $(j-1)$–bit prefix $\mathbf{b}_1^{j-1}$, the encoder of the ideal variable–length source code assigns a codeword with an average length of

$$l_{\mathbf{b}^{j-1}} = n\Pr(\mathbf{b}_1^{j-1}) H(B_j|\mathbf{b}_1^{j-1}) \tag{6.177}$$

bits to the subsequence associated with the prefix $\mathbf{b}_1^{j-1}$. The right–hand side of equation 6.176 is a sum of the average codeword lengths of equation 6.177, normalized by n.

Suppose that the ideal variable–length source code is used to encode all the subsequences of each source sequence. The codeword of each q–ary source sequence is obtained by concatenating all the codewords that were assigned to the subsequences of

the source sequence. Using equation 6.176, the average length of the codeword assigned to a source sequence is derived as $nH(X)$ bits, which equals the per–sequence entropy of the source. It follows that the weight–and–index source code for q–ary memoryless sources would be able to optimally encode symbol sequences from a q–ary memoryless source (i.e. with the minimum average codeword length) by using the ideal variable–length source code to encode each subsequence. The proposed weight–and–index source code for q–ary memoryless sources uses the weight–and–index source code for binary memoryless sources to encode each subsequence, as the ideal variable–length source code cannot be implemented. The observation regarding the effectiveness of the proposed approach to encoding q–ary symbol sequences from memoryless sources justifies the proposed source code, however.

Several definitions are required in order to derive an expression for the length of each codeword produced by the encoder of the weight–and–index source code for q–ary memoryless sources. Let $\mathcal{A}$ denote the symbol alphabet, and let the function $S_{j,k} : \mathcal{A}^n \mapsto \{0,1\}^*$ produce a specific subsequence of the jth n–bit block generated by the source encoder, which is used to encode a source sequence $\mathbf{x}^n \in \mathcal{A}^n$. The bits of the subsequence that the function $S_{j,k}$ produces are associated with the prefix $\mathbf{b}_1^{j-1}$, where $\mathbf{b}_1^{j-1} = \beta_{j-1}(k)$ (the function β_{j-1} is defined in equation 6.8 on page 142). The function $S_{j,k}$ is defined for the parameters $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

Let bit j of the binary word assigned to symbol t of the source sequence $\mathbf{x}^n$ be denoted by $b_{j,t}$, where $j \in \{1, 2, \ldots m'\}$ and $t \in \{1, 2, \ldots n\}$. The function $S_{j,k}$ is formally defined as

$$S_{j,k}(\mathbf{x}) = \begin{cases} b_{j,y} : y \in \{1, 2, \ldots n\} & \text{if } j = 1, \\ b_{j,y} : y \in \{1, 2, \ldots n\} \wedge \{b_{1,y}, b_{2,y}, \ldots b_{j-1,y}\} = \beta_{j-1}(k) & \text{if } j \in \{2, 3, \ldots m'\}, \end{cases} \tag{6.178}$$

where $k \in \{0, 1, \ldots 2^{j-1} - 1\}$. It is assumed that the multiset of elements $S_{j,k}(\mathbf{x})$ is ordered so that $b_{i,v}$ precedes $b_{j,w}$ for all $i < j$, and that $b_{i,v}$ precedes $b_{i,w}$ for all $v < w$.

The derivation of the expression for the codeword length requires that the ordered multiset of elements $S_{j,k}(\mathbf{x})$ be considered equivalent to a sequence. The sequence that is equivalent to an ordered multiset of elements is constructed by concatenating the elements of the multiset in the same order in which they appear in the multiset. The sequence has a length that is equal to the cardinality of the multiset.

The length of the codeword that is assigned to a source sequence $\mathbf{x}^n$ by the weight–and–index source code for q–ary memoryless sources is derived as follows. The first field of the codeword produced by the encoder of the weight–and–index source code for binary memoryless sources, when the encoder is used to encode the subsequence $S_{j,k}(\mathbf{x})$, represents the weight of the subsequence. The first field has a length of

$$l_{1,j,k}(\mathbf{x}) = \lceil \log_2(|S_{j,k}(\mathbf{x})| + 1) \rceil \tag{6.179}$$

bits, as the subsequence weight may assume any integer value from zero to $|S_{j,k}(\mathbf{x})|$. This expression is valid for all $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

The second field of the same codeword represents the index of the subsequence in an ordered set of all sequences with the same length and weight as the subsequence.

The second field has a length of

$$l_{2,j,k}(\mathbf{x}) = \left\lceil \log_2 \binom{|S_{j,k}(\mathbf{x})|}{w_{j,k}(\mathbf{x})} \right\rceil \tag{6.180}$$

bits, where $w_{j,k}(\mathbf{x})$ denotes the weight of the subsequence $S_{j,k}(\mathbf{x})$. This expression is valid for all $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

The length of the codeword assigned to the source sequence $\mathbf{x}^n$ equals the sum of the lengths of the codewords that were assigned to the subsequences of the source sequence. The source sequence $\mathbf{x}^n$ is therefore assigned a codeword of

$$l(\mathbf{x}) = \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left(l_{1,j,k}(\mathbf{x}) + l_{2,j,k}(\mathbf{x}) \right) \tag{6.181}$$

bits.

The decoder of the weight–and–index source code for q–ary memoryless sources decodes the codeword of each source sequence in an iterative fashion. The source decoder decodes a codeword that was assigned to a subsequence of the source sequence during each iteration. These codewords are decoded in the same order in which they were produced by the encoder of the weight–and–index source code for q–ary memoryless sources.

The codeword assigned to any subsequence of the jth n–bit block generated by the source encoder may be uniquely decoded if the length of the subsequence is known[18]. The length of each subsequence of the jth block depends on the $(j-1)$–bit prefixes of the binary words that were assigned to the q–ary symbols of the source sequence. As the source decoder decodes the codewords of the subsequences in the same order in which the subsequences were encoded, it has access to the $(j-1)$–bit prefixes of all the source sequence's binary words as it starts to decode the codewords of the jth block's subsequences. It can therefore calculate the length of each subsequence, and subsequently decode the codeword of each subsequence.

Q–ary context–tree sources Three weight–and–index variable–length source codes for q–ary context–tree sources are derived in this section. Two of the source codes are nonuniversal — these codes are referred to as the type–one and type–two nonuniversal weight–and–index source codes for q–ary context–tree sources. One of the source codes is universal, and is referred to as the universal weight–and–index source code for q–ary context–tree sources.

The three weight–and–index source codes for q–ary context–tree sources are similar to the three weight–and–index source codes for binary context–tree sources that were introduced in section 6.2.1.1 on page 168. All of the weight–and–index source codes for binary and q–ary context–tree sources make use of the Burrows–Wheeler transform. The weight–and–index source codes for q–ary context–tree sources use the weight–and–index source code for q–ary memoryless sources to independently encode each segment of i.i.d. symbols in the BWT output sequence.

[18] This statement follows from theorem 6.2.1 on page 179, which states that the weight–and–index source code for binary memoryless sources is uniquely decodable if the source sequence length is known.

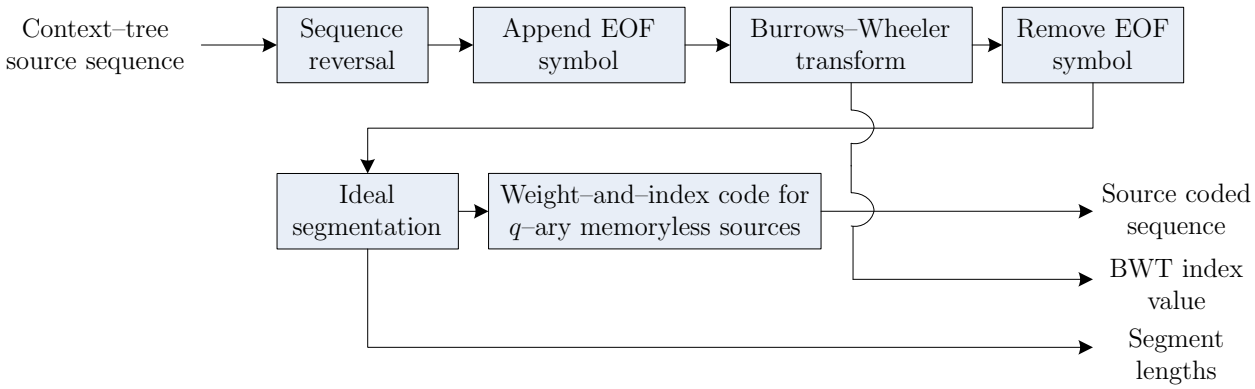


Figure 6.15: A block diagram of the type–one nonuniversal weight–and–index source encoder for q–ary context–tree sources.

It is assumed that sequences from a q–ary context–tree source with a state set $\mathcal{S}$ are source coded in the derivation that follows. The length of the longest source context is denoted by m.

The type–one nonuniversal weight–and–index code The derivation of the type–one nonuniversal weight–and–index code for q–ary context–tree sources is presented in this section. It is assumed that the encoder of this source code has exact apriori knowledge of the source structure (i.e. its minimum suffix set) and a certain number of the initial states of the source. This source code encodes the actual i.i.d. symbol segments in the BWT output sequence using the weight–and–index source code for q–ary memoryless sources.

A block diagram of the type–one nonuniversal weight–and–index source encoder for q–ary context–tree sources is presented in figure 6.15 on page 204. The encoder of this source code reverses the n–symbol source sequence $\mathbf{x}^n$ and appends the `EOF` symbol to the reversed sequence prior to applying the BWT to the sequence. The BWT output sequence $\mathbf{z}^{n+1}$ and the BWT index I may be expressed as

$$(\mathbf{z}^{n+1}, I) = \text{BWT}(\mathcal{R}(\mathbf{x}^n)\$), \tag{6.182}$$

where \$ denotes the `EOF` symbol, and $\mathcal{R}(\cdot)$ denotes sequence reversal. The source encoder removes the `EOF` symbol from the BWT output sequence to obtain the sequence $\mathbf{w}^n$, which is expressed as

$$\begin{aligned} \mathbf{w}^n &= (z_1, z_2, \ldots z_{I-1}, z_{I+1}, \ldots z_{n+1}) \\ &= (\mathbf{z}_1^{I-1}, \mathbf{z}_{I+1}^{n+1}). \end{aligned} \tag{6.183}$$

The reversal of the source sequence causes the BWT to sort the reversed preceding contexts of the original source sequence relative to one another. Those symbols that follow similar preceding contexts[19] in the original source sequence are therefore placed

[19]The m–symbol preceding context $\mathbf{x}_{i-m}^{i-1}$ of symbol x_i is said to be similar to the preceding context of symbol y_i if the symbols closest to x_i match the symbols closest to y_i.

in neighbouring segments of the BWT output sequence. The preceding contexts of the source symbols are identical to the reversed states (i.e. the suffixes of the minimum suffix set) of the context–tree source.

In order to find the exact positions of the transition points between the i.i.d. symbol segments of the BWT output sequence, the source encoder derives the sequence of state transitions that corresponds to the original source sequence. It derives the sequence of state transitions by using its apriori knowledge of the source's minimum suffix set and one or more of its initial states. It counts the number of symbols that the source produced in each of its states. The encoder proceeds by sorting the source states (i.e. the reversed preceding contexts) and assigning a symbol count to each of the source states. Each symbol count is the total number of symbols that the source produced in the corresponding state. The sequence of sorted states and the symbol counts of the states are equivalent to the sequence of segments in the BWT output sequence and the segment lengths. The exact positions of the transition points are obtained from the segment lengths.

Up to m of the initial symbols of the source sequence may be placed in segments that do not correspond to states of the context–tree source, where m denotes the length of the longest preceding context of the source. The source encoder is able to derive the positions of these symbols in the BWT output sequence, and insert transition points on either side of each of these symbols. This approach may significantly increase the average codeword redundancy of the source code, however.

By inserting transition points on either side of segments that do not correspond to the states of the context–tree source, and encoding these additional segments independently from the remaining segments of the BWT output sequence, the source encoder could potentially add $m + 1$ additional fields to a codeword. If the source encoder is not to use more complex integer codes, it must either encode the position of each additional segment in the BWT output sequence (and add up to m fields of $\lceil \log_2(n) \rceil$ bits each to the codeword), or encode the lengths of all the segments in the BWT output sequence (and add up to m fields of $\lceil \log_2(n) \rceil$ bits each to the codeword, as each length is encoded in $\lceil \log_2(n) \rceil$ bits). An additional field that represents the number of segments in the BWT output sequence would have to be added to each codeword, as the source decoder is unaware of the number of additional segments in each BWT output sequence.

Instead of encoding the additional segments of the BWT output sequence independently from the remaining segments, the source encoder may merge the additional segments with the segments that correspond to the states of the context–tree source. The addition of a maximum of m symbols to the remaining segments, each consisting of symbols with a distinct probability distribution, does increase the average redundancy of the codewords assigned to these segments. It is assumed, however, that the source sequence length n is much larger than the length of the longest source context, m. This assumption implies that the redundancy increase associated with the segment–merge approach is small when compared to the redundancy increase associated with independently encoding the additional segments. The encoder of the type–one nonuniversal weight–and–index source code therefore merges the additional segments with the segments that correspond to the states of the context–tree source.

The source encoder independently encodes the segments of symbols using the weight–and–index source code for q–ary memoryless sources. The codewords that are assigned to the segments are concatenated to obtain the source–coded sequence. Let $\mathbf{y}'_i$ denote segment i of the BWT output sequence, which may include up to m merged symbols from segments that do not correspond to the states of the context–tree source. The symbol $\mathbf{y}'_i$ is defined for all $i \in \{1, 2, \ldots |\mathcal{S}|\}$. The codeword assigned to segment i of the BWT output sequence has a length of

$$l_{1,i}^{(s)}(\mathbf{x}) = \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left(l'_{1,i,j,k}(\mathbf{x}) + l'_{2,i,j,k}(\mathbf{x}) \right) \tag{6.184}$$

bits, where $l'_{1,i,j,k}(\mathbf{x})$ and $l'_{2,i,j,k}(\mathbf{x})$ denote the lengths of the first and second fields of the codeword assigned to the subsequence $S_{j,k}(\mathbf{y}'_i)$. The first field of the codeword assigned to subsequence $S_{j,k}(\mathbf{y}'_i)$ represents the weight of the subsequence, and has a length of

$$l'_{1,i,j,k}(\mathbf{x}) = \lceil \log_2(|S_{j,k}(\mathbf{y}'_i)| + 1) \rceil \tag{6.185}$$

bits. This expression is valid for all $i \in \{1, 2, \ldots |\mathcal{S}|\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

The second field of the same codeword represents the index of the subsequence in an ordered set of all sequences with the same length and weight as the subsequence. This field has a length of

$$l'_{2,i,j,k}(\mathbf{x}) = \left\lceil \log_2 \binom{|S_{j,k}(\mathbf{y}'_i)|}{w_{j,k}(\mathbf{y}'_i)} \right\rceil \tag{6.186}$$

bits, where $w_{j,k}(\mathbf{y}'_i)$ denotes the weight of the subsequence $S_{j,k}(\mathbf{y}'_i)$. This expression is valid for all $i \in \{1, 2, \ldots |\mathcal{S}|\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

In order to successfully recover the source sequence from the source–coded sequence, the source decoder requires the BWT index, as well as the lengths of the segments in the BWT output sequence. The source encoder encodes the BWT index using the conventional binary–coded representation of an integer. It is encoded in a total of

$$l_2(\mathbf{x}) = \lceil \log_2(n + 1) \rceil \tag{6.187}$$

bits, as the BWT index may assume $n + 1$ distinct values with the addition of the `EOF` symbol to the reversed source sequence.

The lengths of the segments in the BWT output sequence are encoded as follows. Each BWT output sequence has exactly $|\mathcal{S}|$ segments, as the additional segments are merged with the segments that correspond to the states of the context–tree source. The source encoder therefore encodes a total of $|\mathcal{S}| - 1$ segment lengths for each BWT output sequence, as the length of the final segment may be derived from the sequence length n. Each segment length is encoded in a total of $\lceil \log_2(n + 1) \rceil$ bits using the conventional binary–coded representation of an integer, as each segment length may assume a value in the set $\{0, 1, 2, \ldots n\}$. The total number of bits that are required to encode the segment lengths equals

$$l_3(\mathbf{x}) = (|\mathcal{S}| - 1) \lceil \log_2(n + 1) \rceil . \tag{6.188}$$

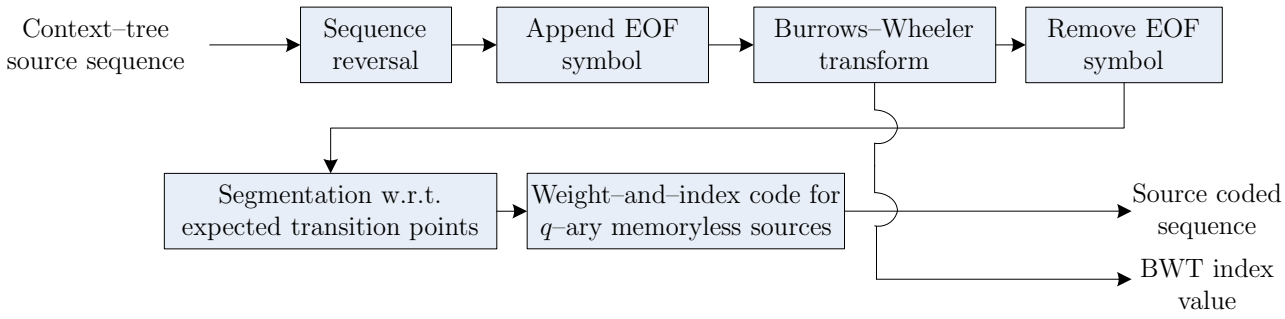


Figure 6.16: A block diagram of the type–two nonuniversal weight–and–index source encoder for q–ary context–tree sources.

The type–two nonuniversal weight–and–index code The encoder and decoder of the type–two nonuniversal weight–and–index source code for q–ary context–tree sources are assumed to have apriori knowledge of the symbol distribution that is associated with each state of the q–ary context–tree source. The encoder of the source code calculates the expected positions of the transition points between the segments of i.i.d. symbols in the BWT output sequence. It independently encodes the segments that are located between consecutive transition points using the weight–and–index source code for q–ary memoryless sources.

A block diagram of the type–two nonuniversal weight–and–index source encoder for q–ary context–tree sources is presented in figure 6.16 on page 207. The encoder reverses the n–symbol source sequence, and appends the `EOF` symbol to the reversed sequence prior to applying the BWT to the sequence. The encoder next removes the `EOF` symbol from the BWT output sequence, and calculates the expected positions of the transition points between the segments of i.i.d. symbols in the BWT output sequence.

The encoder calculates the expected positions of the transition points in the same manner as the encoder of the type–two nonuniversal weight–and–index source code for binary context–tree sources[20]. The encoder first derives the FSM closure of the q–ary context–tree source. It next calculates the state probabilities of the FSM closure using equations 6.90 to 6.92 on page 173. After calculating the probabilities of the FSM closure's states, the source encoder calculates the probabilities of the context–tree source's states, and the expected lengths of the segments in the BWT output sequence. The expected positions of the transition points are calculated using equation 6.89.

The type–two nonuniversal weight–and–index source code for q–ary context–tree sources uses the weight–and–index source code for q–ary memoryless sources to independently encode the segments between the expected positions of consecutive transition points in the BWT output sequence. The codeword that is assigned to the ith expected

[20]Certain single–symbol segments of the BWT output sequence may not correspond to the states of the context–tree source. These segments appear due to the addition of the `EOF` symbol to the reversed source sequence, and are disregarded in the calculation of the expected segment lengths. The single–symbol segments have little impact on the performance of the source code, as a maximum of m of these segments are present in each BWT output sequence.

segment of the BWT output sequence has a length of

$$l_{1,i}^{(s)}(\mathbf{x}) = \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} (\overline{l}_{1,i,j,k}(\mathbf{x}) + \overline{l}_{2,i,j,k}(\mathbf{x})) \tag{6.189}$$

bits. The field lengths $\overline{l}_{1,i,j,k}(\mathbf{x})$ and $\overline{l}_{2,i,j,k}(\mathbf{x})$ are expressed as

$$\overline{l}_{1,i,j,k}(\mathbf{x}) = \lceil \log_2(|S_{j,k}(\overline{\mathbf{y}_i})| + 1) \rceil \tag{6.190}$$

and

$$\overline{l}_{2,i,j,k}(\mathbf{x}) = \left\lceil \log_2 \binom{|S_{j,k}(\overline{\mathbf{y}_i})|}{w_{j,k}(\overline{\mathbf{y}_i})} \right\rceil, \tag{6.191}$$

where $\overline{\mathbf{y}_i}$ denotes the ith expected segment of the BWT output sequence, and $w_{j,k}(\overline{\mathbf{y}_i})$ denotes the weight of the segment. These expressions are valid for all $i \in \{1, 2, \ldots |\mathcal{S}|\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

The encoder of the type–two nonuniversal weight–and–index source code for q–ary context–tree sources encodes the BWT index using the conventional binary–coded representation of an integer. The codeword of the BWT index has a length of

$$l_2(\mathbf{x}) = \lceil \log_2(n + 1) \rceil \tag{6.192}$$

bits.

The decoder of the type–two nonuniversal weight–and–index source code for q–ary context–tree sources determines the expected positions of the transition points in the BWT output sequence, as well as the length of each segment that was encoded by the source encoder. It is therefore able to decode the codeword assigned to each segment. The source encoder does not encode the expected positions of the transition points between the segments of the BWT output sequence.

The universal weight–and–index code The derivation of the universal weight–and–index source code for q–ary context–tree sources is presented in this section. It is assumed that both the encoder and decoder of this source code have no apriori knowledge of the source structure (i.e. its minimum suffix set) or the symbol distributions that are associated with the source states. The source encoder and decoder are assumed to be aware of the length n of the source sequence, as well as the alphabet of the source symbols.

A block diagram of the universal weight–and–index source encoder for q–ary context–tree sources is presented in figure 6.17 on page 209. The encoder of the universal weight–and–index source code for q–ary context–tree sources follows the same initial steps as the encoder of the universal weight–and–index source code for binary context–tree sources when used to encode a source sequence. It reverses the n–symbol source sequence, and appends the `EOF` symbol to the reversed sequence prior to applying the BWT to the sequence. The encoder next removes the `EOF` symbol from the BWT output sequence.

The encoder of the universal weight–and–index source code for q–ary context–tree sources uses the segmentation algorithm that was proposed by Shamir et. al. [159]

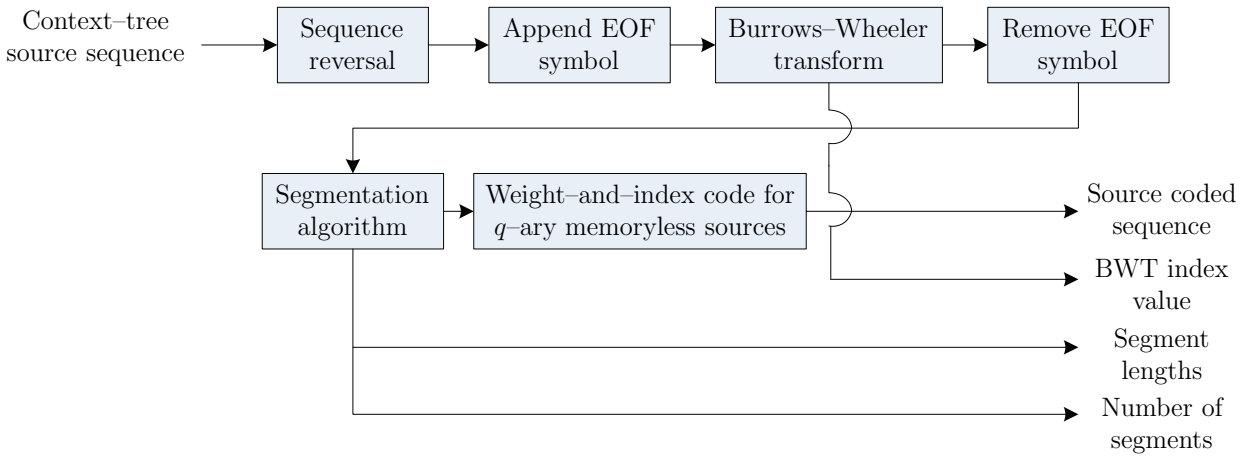


Figure 6.17: A block diagram of the universal weight–and–index source encoder for q–ary context–tree sources.

to estimate the positions of the transition points in the BWT output sequence. This algorithm is nearly identical to the segmentation algorithm used by the encoder of the universal weight–and–index source code for binary context–tree sources (refer to the summary of the algorithm in section 6.2.1.1 on page 175). The summary is not repeated in this section — only the expressions for the level–one and level–two blocks' metrics are provided in this section, as these expressions are redefined if blocks of q–ary symbols are considered (where $q > 2$).

Let the alphabet of the source symbols be denoted by $\mathcal{A}$, and let the lengths of the level–one and level–two blocks be denoted by $k_1(n)$ and $k_2(n)$. Let the frequency count of symbol x in the ith level–one block of the BWT output sequence be denoted by $N_i(x)$, where $i \in \{1, 2, \ldots n/k_1(n)\}$ and $x \in \{1, 2, \ldots q\}$. It is assumed that $n/k_1(n)$ is an integer in order to simplify the discussion. The metric of the rth level–one block, $M(r)$, is expressed as [159]

$$M(r) = \hat{H}(r-1, r+1) - \frac{1}{2}\hat{H}(r-1) - \frac{1}{2}\hat{H}(r+1), \tag{6.193}$$

where

$$\hat{H}(i) = -\sum_{x \in \mathcal{A}} \frac{N_i(x)}{k_1(n)} \log_2 \left(\frac{N_i(x)}{k_1(n)} \right) \tag{6.194}$$

and

$$\hat{H}(i,j) = -\sum_{x \in \mathcal{A}} \frac{N_i(x) + N_j(x)}{2k_1(n)} \log_2 \left(\frac{N_i(x) + N_j(x)}{2k_1(n)} \right), \tag{6.195}$$

with $r \in \{2, 3, \ldots n/k_1(n) - 1\}$ and $i, j \in \{1, 2, \ldots n/k_1(n)\}$. In order to calculate the metrics of the level–two blocks using equations 6.193 to 6.195, the length $k_1(n)$ is replaced by the length $k_2(n)$ in equations 6.194 and 6.195. The frequency count $N_i(x)$ is defined as the frequency count of symbol x in the ith level–two block during the calculation of the level–two blocks' metrics.

After estimating the transition points' positions in the BWT output sequence, the source encoder uses the universal weight–and–index source code for q–ary memoryless

sources to independently encode the symbol sequences between the estimated positions of consecutive transition points. The codeword assigned to segment i of the BWT output sequence has a length of

$$l_{1,i}^{(s)}(\mathbf{x}) = \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left(l_{1,i,j,k}(\mathbf{x}) + l_{2,i,j,k}(\mathbf{x}) \right) \tag{6.196}$$

bits[21], where $i \in \{1, 2, \ldots C\}$ (C denotes the total number of i.i.d. symbol segments that are present in the BWT output sequence associated with the source sequence $\mathbf{x}$). The quantities $l_{1,i,j,k}(\mathbf{x})$ and $l_{2,i,j,k}(\mathbf{x})$ denote the lengths of the first and second fields of the codeword assigned to the subsequence $S_{j,k}(\mathbf{y}_i)$, where $\mathbf{y}_i$ denotes the ith segment of the BWT output sequence.

The first field of the codeword assigned to subsequence $S_{j,k}(\mathbf{y}_i)$ represents the weight of the subsequence, and has a length of

$$l_{1,i,j,k}(\mathbf{x}) = \lceil \log_2(|S_{j,k}(\mathbf{y}_i)| + 1) \rceil \tag{6.197}$$

bits. This expression is valid for all $i \in \{1, 2, \ldots C\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

The second field of the same codeword represents the index of the subsequence in an ordered set of all sequences with the same length and weight as the subsequence. This field has a length of

$$l_{2,i,j,k}(\mathbf{x}) = \left\lceil \log_2 \binom{|S_{j,k}(\mathbf{y}_i)|}{w_{j,k}(\mathbf{y}_i)} \right\rceil \tag{6.198}$$

bits, where $w_{j,k}(\mathbf{y}_i)$ denotes the weight of the subsequence $S_{j,k}(\mathbf{y}_i)$. This expression is valid for all $i \in \{1, 2, \ldots C\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

The source encoder encodes the BWT index using the conventional binary–coded representation of an integer. The codeword assigned to the BWT index has a length of

$$l_2(\mathbf{x}) = \lceil \log_2(n + 1) \rceil \tag{6.199}$$

bits.

In order to successfully decode the codeword of each source sequence, the source decoder of the universal weight–and–index source code for q–ary context–tree sources requires both the number of i.i.d. symbol segments in the BWT output sequence and their lengths. The source encoder encodes each segment length using the conventional binary–coded representation of an integer. The segment lengths are therefore encoded in a total of

$$l_3(\mathbf{x}) = (C - 1) \lceil \log_2(n) \rceil \tag{6.200}$$

bits, as the final segment length may be derived from the source sequence length n. The number of i.i.d. symbol segments in the BWT output sequence is encoded in a total of

$$l_4(\mathbf{x}) = \lceil \log_2(n) \rceil \tag{6.201}$$

bits, as the source encoder has no apriori knowledge regarding the state set of the source.

[21] The codeword lengths that are presented in this section are derived under the assumption that the segmentation algorithm produces perfectly accurate estimates of the transition points' positions in the BWT output sequence.

6.2.2.2 Definition of the source code

A mathematical definition of the universal weight–and–index source code for q–ary memoryless sources is provided in what follows. The source code is defined under the assumption that the number of distinct symbols in the symbol alphabet is a power of two, where $m' = \log_2(q)$.

Let Λ denote the concatenation of two or more bit sequences, and let the encoder of the universal weight–and–index source code for q–ary memoryless sources correspond to the function $g_4 : \mathcal{A}^n \mapsto \{0,1\}^*$. The codeword produced by the source encoder, when used to encode the source sequence $\mathbf{x}^n$, is expressed as

$$\begin{aligned} \mathbf{c}_{\mathbf{x}} &= g_4(\mathbf{x}) \\ &= \Lambda_{j=1}^{m'} \Lambda_{k=0}^{2^{j-1}-1} g_3(S_{j,k}(\mathbf{x})), \end{aligned} \tag{6.202}$$

where the function g_3 corresponds to the encoder of the weight–and–index source code for binary memoryless sources (refer to equation 6.106 on page 178). The sequence $S_{j,k}(\mathbf{x})$ is a specific subsequence of the jth n–bit block generated by the source encoder (refer to equation 6.178 on page 202).

Mathematical definitions of the nonuniversal and universal weight–and–index source codes for q–ary context–tree sources are not provided, as these source codes use the weight–and–index source code for q–ary memoryless sources. The unique decodability of the nonuniversal and universal weight–and–index source codes for q–ary context–tree sources depends on the unique decodability of the weight–and–index source code for q–ary memoryless sources, which is subsequently proved.

6.2.2.3 Proof of unique decodability

This section contains a proof of the unique decodability of the universal weight–and–index source code for q–ary memoryless sources. Several lemmas that are required to prove its unique decodability are presented in what follows.

Lemma 6.2.1. *The subsequence length $|S_{j,k}(\mathbf{x})|$, where $j \in \{1,2,\ldots m'\}$ and $k \in \{0,1,\ldots 2^{j-1}-1\}$, may be calculated if the subsequences $S_{v,w}(\mathbf{x})$, for all $v \in \{1,2,\ldots j-1\}$ and all $w \in \{0,1,\ldots 2^{v-1}-1\}$, are known.*

Proof. The subsequence $S_{j,k}(\mathbf{x})$, where $j = 1$ and $k = 0$, has a length of

$$|S_{1,0}(\mathbf{x})| = n \tag{6.203}$$

bits (refer to equation 6.178 on page 202). The length of each subsequence $S_{j,k}(\mathbf{x})$, where $j \in \{2,3,\ldots m'\}$ and $k \in \{0,1,\ldots 2^{j-1}-1\}$, may be calculated using equation 6.178 if the source bits of the multiset $\{b_{v,q} : 1 \leq v < j \wedge 1 \leq q \leq n\}$ are known. As

$$\bigcup_{v=1}^{j-1} \bigcup_{w=0}^{2^{v-1}-1} S_{v,w}(\mathbf{x}) = \{b_{v,q} : 1 \leq v < j \wedge 1 \leq q \leq n\}, \tag{6.204}$$

it follows that the subsequence lengths may be calculated. □

Lemma 6.2.2. *The subsequence codeword $g_3(S_{j,k}(\mathbf{x}))$, where $j \in \{1, 2, \dots m'\}$ and $k \in \{0, 1, \dots 2^{j-1}-1\}$, may be delimited if the subsequences $S_{v,w}(\mathbf{x})$, for all $v \in \{1, 2, \dots j-1\}$ and all $w \in \{0, 1, \dots 2^{v-1}-1\}$, are known.*

Proof. Each subsequence codeword may be delimited by calculating its length. The length of the codeword $g_3(S_{j,k}(\mathbf{x}))$ is calculated using the equation

$$|g_3(S_{j,k}(\mathbf{x}))| = \lceil \log_2(|S_{j,k}(\mathbf{x})| + 1) \rceil + \left\lceil \log_2 \binom{|S_{j,k}(\mathbf{x})|}{w_{j,k}(\mathbf{x})} \right\rceil, \tag{6.205}$$

where $w_{j,k}(\mathbf{x})$ denotes the number of nonzero–valued bits in the subsequence $S_{j,k}(\mathbf{x})$. Knowledge of both the subsequence length $|S_{j,k}(\mathbf{x})|$ and the subsequence weight $w_{j,k}(\mathbf{x})$ is required in order to evaluate equation 6.205.

It is subsequently proved that the length and the weight of the subsequence may be calculated. Lemma 6.2.1 on page 211 states that the subsequence length $|S_{j,k}(\mathbf{x})|$ may be calculated if the subsequences $S_{v,w}(\mathbf{x})$, for all $v \in \{1, 2, \dots j-1\}$ and all $w \in \{0, 1, \dots 2^{v-1}-1\}$, are known. This condition is satisfied — the remainder of the proof concerns the calculation of the subsequence weight $w_{j,k}(\mathbf{x})$.

The length of the first field of the codeword $g_3(S_{j,k}(\mathbf{x}))$ may be calculated using the equation

$$l_{1,j,k}(\mathbf{x}) = \lceil \log_2(|S_{j,k}(\mathbf{x})| + 1) \rceil, \tag{6.206}$$

as the subsequence length $|S_{j,k}(\mathbf{x})|$ is known. Let the bit sequence $\mathbf{c}_{j,k}$ be defined as

$$\mathbf{c}_{j,k} \triangleq g_3(S_{j,k}(\mathbf{x})), \tag{6.207}$$

and let the bit sequence $\mathbf{c}_{1,j,k}$ be defined as

$$\mathbf{c}_{1,j,k} \triangleq \mathcal{H}_{l_{1,j,k}(\mathbf{x})}(\mathbf{c}_{j,k}), \tag{6.208}$$

where the function $\mathcal{H}_z$ is defined in section 6.2.1.2 on page 178. The equation

$$w_{j,k}(\mathbf{x}) = B^{-1}_{l_{1,j,k}(\mathbf{x})}(\mathbf{c}_{1,j,k}) \tag{6.209}$$

holds, where the function B_z is defined in equation 6.8 on page 142. The weight of the subsequence may therefore be calculated, and equation 6.205 may be evaluated in order to delimit the codeword. □

Lemma 6.2.3. *The weight–and–index source code for sequences from binary memoryless sources is uniquely decodable when used to encode the subsequences $S_{j,k}(\mathbf{x})$, where $j \in \{1, 2, \dots m'\}$ and $k \in \{0, 1, \dots 2^{j-1}-1\}$, provided that the subsequences $S_{v,w}(\mathbf{x})$, for all $v \in \{1, 2, \dots j-1\}$ and all $w \in \{0, 1, \dots 2^{v-1}-1\}$, are known.*

Proof. Theorem 6.2.1 on page 179 states that the weight–and–index source code for sequences from binary memoryless sources is uniquely decodable if the source decoder is aware of the length of each sequence. Lemma 6.2.1 states that the length of the subsequence $S_{j,k}(\mathbf{x})$ may be determined if the subsequences $S_{v,w}(\mathbf{x})$, for all $v \in \{1, 2, \dots j-1\}$ and all $w \in \{0, 1, \dots 2^{v-1}-1\}$, are known. As this condition is satisfied, it follows that the weight–and–index source code for sequences from binary memoryless sources is uniquely decodable when used to encode the subsequences $S_{j,k}(\mathbf{x})$, where $j \in \{1, 2, \dots m'\}$ and $k \in \{0, 1, \dots 2^{j-1}-1\}$. □

Lemma 6.2.4. *The subsequences $S_{j,k}(\mathbf{x})$, for a fixed $j \in \{1, 2, \ldots m'\}$ and all $k \in \{0, 1, \ldots 2^{j-1} - 1\}$, may be uniquely decoded from the sequence $\mathbf{c}_j \triangleq \Lambda_{i=0}^{2^{j-1}-1} g_3(S_{j,i}(\mathbf{x}))$, provided that the subsequences $S_{v,w}(\mathbf{x})$, for all $v \in \{1, 2, \ldots j-1\}$ and all $w \in \{0, 1, \ldots 2^{v-1} - 1\}$, are known.*

Proof. The lemma is proved using mathematical induction. It is first proved that the subsequence $S_{j,0}(\mathbf{x})$ may be uniquely decoded from the sequence $\mathbf{c}_j$. The codeword length $|g_3(S_{j,0}(\mathbf{x}))|$ is obtained by calculating the length and the weight of subsequence $S_{j,0}(\mathbf{x})$, and evaluating equation 6.205. The length of the subsequence may be calculated using equation 6.178, as the subsequences $S_{v,w}(\mathbf{x})$, for all $v \in \{1, 2, \ldots j-1\}$ and all $w \in \{0, 1, \ldots 2^{v-1}-1\}$, are known (refer to lemma 6.2.1). The weight of the subsequence is calculated using equation 6.209, where $\mathbf{c}_{1,j,0}$ is obtained by using the equation

$$\mathbf{c}_{1,j,0} = \mathcal{H}_{l_{1,j,0}(\mathbf{x})}(\mathbf{c}_j), \tag{6.210}$$

and where $l_{1,j,0}(\mathbf{x})$ is calculated using equation 6.206. The codeword $\mathbf{c}_{j,0} = g_3(S_{j,0}(\mathbf{x}))$ is obtained by using the equation

$$\mathbf{c}_{j,0} = \mathcal{H}_{|g_3(S_{j,0}(\mathbf{x}))|}(\mathbf{c}_j). \tag{6.211}$$

The codeword $\mathbf{c}_{j,0}$ may be correctly decoded, as the length of the subsequence that is associated with the codeword is known (refer to lemma 6.2.3). The subsequence $S_{j,0}(\mathbf{x})$ is therefore obtained by decoding the codeword $\mathbf{c}_{j,0}$.

It is subsequently assumed that the subsequences $S_{j,u}(\mathbf{x})$, for all $u \in \{0, 1, \ldots k-1\}$ and for a certain integer $k \in \{1, 2, \ldots 2^{j-1} - 1\}$, may be recovered from the sequence $\mathbf{c}_j$, and proved that the subsequence $S_{j,k}(\mathbf{x})$ may be obtained from the sequence $\mathbf{c}_j$. As it is assumed that the subsequences $S_{j,u}(\mathbf{x})$ (for all $u \in \{0, 1, \ldots k-1\}$) are known, the sum

$$l'_{j,k}(\mathbf{x}) = \sum_{i=0}^{k-1} |g_3(S_{j,i}(\mathbf{x}))| \tag{6.212}$$

may be calculated. Let the function $\mathcal{H}'_k : \mathbb{B}^{k+r} \mapsto \mathbb{B}^r$ map the $(k+r)$–bit sequences in its domain to their r–bit suffixes, where $k \geq 0$ and $r > 0$. The sequence $\mathbf{c}'_{j,k} \triangleq \mathcal{H}'_{l'_{j,k}(\mathbf{x})}(\mathbf{c}_j)$ may be expressed as

$$\mathbf{c}'_{j,k} = \Lambda_{i=k}^{2^{j-1}-1} g_3(S_{j,i}(\mathbf{x})). \tag{6.213}$$

It is subsequently proved that the subsequence $S_{j,k}(\mathbf{x})$ may be obtained from the sequence $\mathbf{c}'_{j,k}$. The codeword length $|g_3(S_{j,k}(\mathbf{x}))|$ is calculated by determining the length and the weight of the subsequence $S_{j,k}(\mathbf{x})$, and evaluating equation 6.205. The length of the subsequence may be calculated using equation 6.178, as the subsequences $S_{v,w}(\mathbf{x})$, for all $v \in \{1, 2, \ldots j-1\}$ and all $w \in \{0, 1, \ldots 2^{v-1} - 1\}$, are known (refer to lemma 6.2.1). The weight of the subsequence is calculated using equation 6.209, where $\mathbf{c}_{1,j,k}$ is obtained by evaluating the equation

$$\mathbf{c}_{1,j,k} = \mathcal{H}_{l_{1,j,k}(\mathbf{x})}(\mathbf{c}'_{j,k}), \tag{6.214}$$

and where $l_{1,j,k}(\mathbf{x})$ is determined by evaluating equation 6.206. The codeword $\mathbf{c}_{j,k} = g_3(S_{j,k}(\mathbf{x}))$ is obtained using the equation

$$\mathbf{c}_{j,k} = \mathcal{H}_{|g_3(S_{j,k}(\mathbf{x}))|}(\mathbf{c}'_{j,k}). \tag{6.215}$$

The codeword $\mathbf{c}_{j,k}$ may be correctly decoded as the length of the subsequence associated with the codeword is known (refer to lemma 6.2.3). The subsequence $S_{j,k}(\mathbf{x})$ is therefore obtained by decoding the codeword $\mathbf{c}_{j,k}$. The proof that the subsequences $S_{j,k}(\mathbf{x})$ (for a fixed $j \in \{1, 2, \ldots m'\}$ and all $k \in \{0, 1, \ldots 2^{j-1} - 1\}$) may be uniquely recovered from the sequence $\mathbf{c}_j$ is therefore complete. □

Theorem 6.2.2. *The universal weight–and–index source code for q–ary memoryless sources is uniquely decodable.*

Proof. It is proved that a source sequence $\mathbf{x}^n$ may be recovered from its codeword, which is expressed as

$$\begin{aligned} \mathbf{c}_{\mathbf{x}} &= \Lambda_{j=1}^{m'} \Lambda_{k=0}^{2^{j-1}-1} g_3(S_{j,k}(\mathbf{x})) \\ &= \Lambda_{j=1}^{m'} \mathbf{c}_j. \end{aligned} \tag{6.216}$$

The proof is carried out using mathematical induction. It is first proved that the subsequence $S_{1,0}(\mathbf{x})$ may be recovered from the codeword $\mathbf{c}_{\mathbf{x}}$. The length of the codeword assigned to the subsequence $S_{1,0}(\mathbf{x})$ is obtained from the codeword $\mathbf{c}_{\mathbf{x}}$ by calculating the length and the weight of the subsequence $S_{1,0}(\mathbf{x})$, and evaluating equation 6.205. The subsequence length $|S_{1,0}(\mathbf{x})|$ equals n bits by definition. The subsequence weight $w_{1,0}(\mathbf{x})$ is calculated using equation 6.209, where $\mathbf{c}_{1,1,0}$ is obtained by evaluating the expression

$$\mathbf{c}_{1,1,0} = \mathcal{H}_{l_{1,1,0}(\mathbf{x})}(\mathbf{c}_{\mathbf{x}}), \tag{6.217}$$

and where $l_{1,1,0}(\mathbf{x})$ is calculated using equation 6.206. The length of the codeword $g_3(S_{1,0}(\mathbf{x}))$ may therefore be calculated.

After calculating the length of the codeword $g_3(S_{1,0}(\mathbf{x}))$, the codeword $\mathbf{c}_1$ is obtained by evaluating the expression

$$\mathbf{c}_1 = \mathcal{H}_{|g_3(S_{1,0}(\mathbf{x}))|}(\mathbf{c}_{\mathbf{x}}). \tag{6.218}$$

It follows that the subsequence $S_{1,0}(\mathbf{x})$ may be recovered from the codeword $\mathbf{c}_1$ (refer to lemma 6.2.4).

It is subsequently assumed that the subsequences $S_{i,u}(\mathbf{x})$ (for all $i \in \{1, 2, \ldots j-1\}$ and all $u \in \{0, 1, \ldots 2^{i-1} - 1\}$) are known, and proved that the subsequences $S_{j,v}(\mathbf{x})$ (for all $v \in \{0, 1, \ldots 2^{j-1} - 1\}$ and any $j \in \{2, 3, \ldots m'\}$) may be recovered from the codeword $\mathbf{c}_{\mathbf{x}}$. The sum

$$l'_j(\mathbf{x}) = \sum_{i=1}^{j-1} \sum_{u=0}^{2^{i-1}-1} |g_3(S_{i,u}(\mathbf{x}))| \tag{6.219}$$

may be calculated, as the relevant subsequences are known. The sequence $\mathbf{c}'_j$, which is defined as

$$\mathbf{c}'_j \triangleq \mathcal{H}'_{l'_j(\mathbf{x})}(\mathbf{c}_{\mathbf{x}}), \tag{6.220}$$

may be expressed as

$$\mathbf{c}'_j = \Lambda_{i=j}^{m'} \mathbf{c}_i. \tag{6.221}$$

It follows that the subsequences $S_{j,v}(\mathbf{x})$ (for all $v \in \{0, 1, \ldots 2^{j-1} - 1\}$) may be recovered from the codeword $\mathbf{c}'_j$, as the subsequences $S_{i,u}(\mathbf{x})$ (for all $i \in \{1, 2, \ldots j-1\}$ and all $u \in \{0, 1, \ldots 2^{i-1} - 1\}$) are known — refer to lemma 6.2.4. It follows from mathematical induction that the subsequences $S_{j,k}(\mathbf{x})$, for all $j \in \{1, 2, \ldots m'\}$ and all $k \in \{0, 1, \ldots 2^{j-1} - 1\}$, may be recovered from the codeword $\mathbf{c}_\mathbf{x}$. As all the subsequences associated with the codeword $\mathbf{c}_\mathbf{x}$ may be recovered, the entire source sequence $\mathbf{x}^n$ may be recovered. The universal weight–and–index source code for q–ary memoryless sources is therefore uniquely decodable. □

6.2.2.4 Theoretical performance

The average per–codeword redundancies of two universal weight–and–index source codes for q–ary sources are derived in this section. The universal source code for q–ary memoryless sources, as well as the universal source code for q–ary context–tree sources are considered. The source coding of finite–length source sequences and source sequences with lengths that tend to infinity are investigated.

Performance w.r.t. q–ary memoryless sources Suppose that the source encoder of the universal weight–and–index source code for q–ary memoryless sources is used to encode a finite–length sequence $\mathbf{x}^n$ of symbols from a q–ary memoryless source, where $n > 1$ and $m' = \log_2(q)$. The source encoder assigns a codeword of

$$l(\mathbf{x}) = \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} (l_{1,j,k}(\mathbf{x}) + l_{2,j,k}(\mathbf{x})) \tag{6.222}$$

bits to the source sequence, with

$$l_{1,j,k}(\mathbf{x}) = \lceil \log_2(|S_{j,k}(\mathbf{x})| + 1) \rceil \tag{6.223}$$

and

$$l_{2,j,k}(\mathbf{x}) = \left\lceil \log_2 \binom{|S_{j,k}(\mathbf{x})|}{w_{j,k}(\mathbf{x})} \right\rceil, \tag{6.224}$$

and where $w_{j,k}(\mathbf{x})$ denotes the number of nonzero–valued bits in the subsequence[22] $S_{j,k}(\mathbf{x})$. Let the subsequence length $|S_{j,k}(\mathbf{x})|$ be denoted by

$$n_{j,k} \triangleq |S_{j,k}(\mathbf{x})|, \tag{6.225}$$

and let the subsequence weight be denoted by

$$w_{j,k} \triangleq w_{j,k}(\mathbf{x}). \tag{6.226}$$

The bound of equation 6.130 is used to derive the upper bound

$$l(\mathbf{x}) < \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left[\log_2 \left(\frac{n_{j,k}(n_{j,k}+1)}{\sqrt{2\pi n_{j,k}(n_{j,k}-1)}} \right) + \left(\frac{1}{12 n_{j,k}} + 1 \right) \log_2(e) + n_{j,k} h\left(\frac{w_{j,k}}{n_{j,k}} \right) + 2 \right] \tag{6.227}$$

[22] Refer to equation 6.178 for the definition of the subsequence $S_{j,k}(\mathbf{x})$.

on the length of the codeword that is assigned to the source sequence $\mathbf{x}^n$, under the assumption that $n_{j,k} > 1$ for all $j \in \{1, 2, \ldots m'\}$ and all $k \in \{0, 1, \ldots 2^{j-1} - 1\}$. The case where one or more subsequence lengths $n_{j,k}$ equals zero bits or one bit is considered towards the end of the derivation.

Due to the fact that $1 < n_{j,k} \leq n$, and that the function

$$v(n_{j,k}) = \log_2 \left(\frac{n_{j,k}(n_{j,k}+1)}{\sqrt{2\pi n_{j,k}(n_{j,k}-1)}} \right) \tag{6.228}$$

is a monotonically increasing function of $n_{j,k}$, the inequality

$$\log_2 \left(\frac{n_{j,k}(n_{j,k}+1)}{\sqrt{2\pi n_{j,k}(n_{j,k}-1)}} \right) \leq \log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) \tag{6.229}$$

holds. It follows that

$$l(\mathbf{x}) \quad < \quad \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left[\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \frac{13}{12} \log_2(e) + n_{j,k} h\left(\frac{w_{j,k}}{n_{j,k}} \right) + 2 \right]. \tag{6.230}$$

It is subsequently proved that the sum between the square brackets of equation 6.230 is a valid upper bound on the length of each codeword that is assigned to a subsequence of zero bits or one bit. The inequality

$$\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) > 0 \tag{6.231}$$

holds for all sequence lengths n longer than one bit. As

$$\frac{13}{12} \log_2(e) > 0 \tag{6.232}$$

and

$$n_{j,k} h\left(\frac{w_{j,k}}{n_{j,k}} \right) \geq 0 \tag{6.233}$$

for all relevant values of $n_{j,k}$ and $w_{j,k}$, it follows that

$$\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \frac{13}{12} \log_2(e) + n_{j,k} h\left(\frac{w_{j,k}}{n_{j,k}} \right) + 2 > 1. \tag{6.234}$$

The codewords assigned to subsequences of zero bits and one bit have lengths of zero bits and one bit, respectively. As the sum within the square brackets of equation 6.230 exceeds one bit, it follows that equation 6.230 is a valid upper bound on the length of the codeword that is assigned to a source sequence, regardless of the length of any of its subsequences. Equation 6.230 may be simplified as

$$l(\mathbf{x}) \quad < \quad \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} n_{j,k} h\left(\frac{w_{j,k}}{n_{j,k}} \right) + (2^{m'} - 1) \left(\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \frac{13}{12} \log_2(e) + 2 \right). \tag{6.235}$$

Several definitions that are used in the derivation of an upper bound on the average codeword length are subsequently presented. A random source sequence $\mathbf{X}^n$ has subsequences with random lengths $N_{j,k}$ and random weights $W_{j,k}$, where $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$. Let $n'_{j,k}$ denote an integer from the set $\{0, 1, \ldots n\}$, and let $w'_{j,k}$ denote an integer from the set $\{0, 1, \ldots n'_{j,k}\}$. Let the probability distribution $\Pr(n'_{j,k}) \triangleq \Pr(N_{j,k} = n'_{j,k})$ be defined as

$$\Pr(n'_{j,k}) = \sum_{\mathbf{x}} \Pr(\mathbf{x}) : \mathbf{x} \in \mathcal{A}^n \wedge |S_{j,k}(\mathbf{x})| = n'_{j,k}, \tag{6.236}$$

and let the probability distribution $\Pr(w'_{j,k}) \triangleq \Pr(W_{j,k} = w'_{j,k})$ be defined as

$$\Pr(w'_{j,k}) = \sum_{\mathbf{x}} \Pr(\mathbf{x}) : \mathbf{x} \in \mathcal{A}^n \wedge w_{j,k}(\mathbf{x}) = w'_{j,k}. \tag{6.237}$$

The ordered multisets $\mathbf{N}$, $\mathbf{W}$, $\mathbf{n}'$, and $\mathbf{w}'$ are defined as

$$\mathbf{N} \triangleq \{N_{j,k} : 1 \leq j \leq m' \wedge 0 \leq k \leq 2^{j-1} - 1\}, \tag{6.238}$$

$$\mathbf{W} \triangleq \{W_{j,k} : 1 \leq j \leq m' \wedge 0 \leq k \leq 2^{j-1} - 1\}, \tag{6.239}$$

$$\mathbf{n}' \triangleq \{n'_{j,k} : 1 \leq j \leq m' \wedge 0 \leq k \leq 2^{j-1} - 1\}, \tag{6.240}$$

and

$$\mathbf{w}' \triangleq \{w'_{j,k} : 1 \leq j \leq m' \wedge 0 \leq k \leq 2^{j-1} - 1\}. \tag{6.241}$$

The ordered multisets appear in the expressions for the joint probability distributions of the lengths and the weights of a random source sequence's subsequences. The joint probability distribution of the lengths of a random source sequence's subsequences is expressed as $\Pr(\mathbf{n}') \triangleq \Pr(\mathbf{N} = \mathbf{n}')$. The joint probability distribution of the weights of a random source sequence's subsequences is expressed as $\Pr(\mathbf{w}') \triangleq \Pr(\mathbf{W} = \mathbf{w}')$.

The average length of the codewords that are assigned to source sequences from a q–ary memoryless source is subsequently derived. The fact that the source sequences $\mathbf{x}^n$ and $\mathbf{y}^n$ are assigned codewords of equal length if $|S_{j,k}(\mathbf{x})| = |S_{j,k}(\mathbf{y})|$ and $w_{j,k}(\mathbf{x}) = w_{j,k}(\mathbf{y})$ for all $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$ is used in the derivation. The average codeword length $L(\mathbf{X}^n)$ may be expressed as

$$\begin{aligned} L(\mathbf{X}^n) &= \sum_{\mathbf{x} \in \mathcal{A}^n} \Pr(\mathbf{x}) l(\mathbf{x}) \\ &= \sum_{\mathbf{n}' \in \mathbf{q}} \sum_{\mathbf{w}' \in \mathbf{q}} \Pr(\mathbf{n}', \mathbf{w}') l(\mathbf{n}', \mathbf{w}'), \end{aligned} \tag{6.242}$$

where $l(\mathbf{n}', \mathbf{w}')$ denotes the length of the codeword that is assigned to a source sequence with subsequence lengths $\mathbf{n}'$ and subsequence weights $\mathbf{w}'$, and where $\mathbf{q} \triangleq \{0, 1, \ldots n\}^{2^{m'}-1}$.

Equation 6.235 is substituted into equation 6.242 to obtain the upper bound

$$\begin{aligned} L(\mathbf{X}^n) &< \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{\mathbf{n}' \in \mathbf{q}} \sum_{\mathbf{w}' \in \mathbf{q}} \Pr(\mathbf{n}', \mathbf{w}') n'_{j,k} h\left(\frac{w'_{j,k}}{n'_{j,k}}\right) \\ &\quad + (2^{m'} - 1)\left(\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \frac{13}{12} \log_2(e) + 2 \right) \\ &< \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{n'_{j,k}=0}^{n} \sum_{w'_{j,k}=0}^{n'_{j,k}} \Pr(n'_{j,k}, w'_{j,k}) n'_{j,k} h\left(\frac{w'_{j,k}}{n'_{j,k}}\right) \\ &\quad + (2^{m'} - 1)\left(\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \frac{13}{12} \log_2(e) + 2 \right) \\ &< \left[\sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{n'_{j,k}=0}^{n} \sum_{w'_{j,k}=0}^{n'_{j,k}} \Pr(n'_{j,k}) \Pr(w'_{j,k}|n'_{j,k}) n'_{j,k} h\left(\frac{w'_{j,k}}{n'_{j,k}}\right) \right] \\ &\quad + (2^{m'} - 1)\left(\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \frac{13}{12} \log_2(e) + 2 \right). \end{aligned} \tag{6.243}$$

The function $\Pr(w'_{j,k}|n'_{j,k})$ is the probability mass function of a binomial random variable with parameters $n'_{j,k}$ and $p_{j,k}$, where

$$p_{j,k} \triangleq \Pr(B_j = 1 | \mathbf{B}_1^{j-1} = \beta_{j-1}(k)). \tag{6.244}$$

It is defined for all $w'_{j,k} \in \{0, 1, \ldots n'_{j,k}\}$.

Jensen's inequality is used to simplify the term within the square brackets of equation 6.243, as the binary entropy function $h(\cdot)$ is a concave function. The term is simplified as

$$\begin{aligned} &\sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{n'_{j,k}=0}^{n} \sum_{w'_{j,k}=0}^{n'_{j,k}} \Pr(n'_{j,k}) \Pr(w'_{j,k}|n'_{j,k}) n'_{j,k} h\left(\frac{w'_{j,k}}{n'_{j,k}}\right) \\ \leq &\sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{n'_{j,k}=0}^{n} \Pr(n'_{j,k}) n'_{j,k} h\left(\sum_{w'_{j,k}=0}^{n'_{j,k}} \Pr(w'_{j,k}|n'_{j,k}) \frac{w'_{j,k}}{n'_{j,k}} \right) \\ \leq &\sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{n'_{j,k}=0}^{n} \Pr(n'_{j,k}) n'_{j,k} h(p_{j,k}). \end{aligned} \tag{6.245}$$

The final step of equation 6.245 follows from the fact that the expected value of the binomial random variable equals $n'_{j,k} p_{j,k}$. The term on the right–hand side of equation 6.245 may be rewritten as

$$\begin{aligned} &\sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{n'_{j,k}=0}^{n} \Pr(n'_{j,k}) n'_{j,k} h(p_{j,k}) \\ = &\sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{n'_{j,k}=0}^{n} \Pr(n'_{j,k}) n'_{j,k} H(B_j | \mathbf{B}_1^{j-1} = \beta_{j-1}(k)). \end{aligned} \tag{6.246}$$

The function $\Pr(n'_{j,k})$ is the probability mass function of a binomial random variable with parameters n and $p''_{j,k}$, where

$$p''_{j,k} \triangleq \Pr(\mathbf{B}_1^{j-1} = \beta_{j-1}(k)). \tag{6.247}$$

It is defined for all $n'_{j,k} \in \{0, 1, \ldots n\}$.

Equation 6.246 is simplified as

$$\begin{aligned}
& \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{n'_{j,k}=0}^{n} \Pr(n'_{j,k}) n'_{j,k} H(B_j|\mathbf{B}_1^{j-1} = \beta_{j-1}(k)) \\
= \; & n \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} p''_{j,k} H(B_j|\mathbf{B}_1^{j-1} = \beta_{j-1}(k)) \\
= \; & n\left[H(B_1) + \sum_{j=2}^{m'} H(B_j|B_1, B_2, \ldots B_{j-1})\right] \\
= \; & nH(B_1, B_2, \ldots B_{m'}) \\
= \; & nH(X).
\end{aligned} \tag{6.248}$$

The first step of equation 6.248 follows from the fact that the expected value of the binomial random variable equals $np''_{j,k}$.

A bound on the average length $L(\mathbf{X}^n)$ of a codeword assigned to an n–symbol source sequence by the encoder of the universal weight–and–index source code for q–ary memoryless sources is derived using equations 6.243 to 6.248 as

$$L(\mathbf{X}^n) < (2^{m'} - 1)\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right) + nH(X). \tag{6.249}$$

An upper bound on the average per–codeword redundancy of the universal weight–and–index source code for q–ary memoryless sources is derived as

$$\begin{aligned}
R(\mathbf{X}^n) &= L(\mathbf{X}^n) - H(\mathbf{X}^n) \\
&= L(\mathbf{X}^n) - nH(X) \\
&< (2^{m'} - 1)\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right).
\end{aligned} \tag{6.250}$$

The limit of the normalized average per–codeword redundancy, as the source sequence length n tends to infinity, is derived as follows. The average per–codeword redundancy satisfies the inequality

$$0 \leq R(\mathbf{X}^n) < (2^{m'} - 1)\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right) \tag{6.251}$$

for all $n > 1$. The average per–codeword redundancy is therefore expressed as

$$R(\mathbf{X}^n) = K_n q(n), \tag{6.252}$$

where

$$q(n) = (2^{m'} - 1)\left(\log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) + \frac{13}{12} \log_2(e) + 2 \right) \tag{6.253}$$

and $0 \leq K_n < 1$ for all $n > 1$. The limit[23]

$$\begin{aligned} \lim_{n\to\infty} \frac{1}{n} q(n) &= \lim_{n\to\infty} \frac{(2^{m'} - 1)}{n} \log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) \\ &= \lim_{n\to\infty} \frac{1}{n} \log_2 \left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}} \right) \\ &= 0 \end{aligned} \tag{6.254}$$

is used to prove that

$$\begin{aligned} \lim_{n\to\infty} \frac{1}{n} R(\mathbf{X}^n) &= \lim_{n\to\infty} \frac{1}{n} K_n q(n) \\ &= 0. \end{aligned} \tag{6.255}$$

The limit of the normalized average per–codeword redundancy of the universal weight–and–index source code for q–ary memoryless sources, as the source sequence length tends to infinity, equals zero. It is concluded that the source code is universal.

Performance w.r.t. q–ary context–tree sources Suppose that the source encoder of the universal weight–and–index source code for q–ary context–tree sources is used to encode a finite–length symbol sequence $\mathbf{x}^n$ from a q–ary context–tree source, where $n > 1$ and $m' = \log_2(q)$. Let $|\mathcal{S}|$ denote the number of states of the context–tree source, m the length of the longest source context, and C the actual number of i.i.d. symbol segments in the BWT output sequence. Also, let $n_{i,j,k}$ and $w_{i,j,k}$ denote the length and weight of the subsequence $S_{j,k}(\mathbf{y}_i)$, where $\mathbf{y}_i$ denotes segment i of the BWT output sequence, and where $i \in \{1, 2, \ldots C\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

The source sequence $\mathbf{x}^n$ is assigned a codeword of

$$l(\mathbf{x}) = \sum_{i=1}^{C} \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left(l_{1,i,j,k}(\mathbf{x}) + l_{2,i,j,k}(\mathbf{x}) \right) + \sum_{v=2}^{4} l_v(\mathbf{x}) \tag{6.256}$$

bits. The field lengths of equation 6.256 were defined in section 6.2.2.1 on page 210 as

$$l_{1,i,j,k}(\mathbf{x}) = \lceil \log_2(n_{i,j,k} + 1) \rceil , \tag{6.257}$$

$$l_{2,i,j,k}(\mathbf{x}) = \left\lceil \log_2 \binom{n_{i,j,k}}{w_{i,j,k}} \right\rceil , \tag{6.258}$$

$$l_2(\mathbf{x}) = \lceil \log_2(n + 1) \rceil , \tag{6.259}$$

$$l_3(\mathbf{x}) = (C - 1) \lceil \log_2(n) \rceil \tag{6.260}$$

[23] The second step of equation 6.254 follows from equation 6.138.

and

$$l_4(\mathbf{x}) = \lceil \log_2(n) \rceil . \tag{6.261}$$

An upper bound on the codeword length of equation 6.256 is derived as

$$\begin{aligned} l(\mathbf{x}) &= \sum_{i=1}^{C} \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left(l_{1,i,j,k}(\mathbf{x}) + l_{2,i,j,k}(\mathbf{x}) \right) + \sum_{v=2}^{4} l_v(\mathbf{x}) \\ &\leq \sum_{i=1}^{|\mathcal{S}|} \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left(l_{1,i,j,k}(\mathbf{x}) + l_{2,i,j,k}(\mathbf{x}) \right) + \sum_{v=2}^{4} l_v(\mathbf{x}) + mm', \end{aligned} \tag{6.262}$$

where it is assumed that the BWT output sequence segments $\mathbf{y}_1, \mathbf{y}_2, \ldots \mathbf{y}_{|\mathcal{S}|}$ correspond to the $|\mathcal{S}|$ states of the context–tree source. The derivation of the upper bound makes use of the fact that subsequences of one bit are assigned codewords of one bit, and that each segment of the BWT output sequence that does not correspond to a state of the context–tree source consists of one symbol. The derivation also uses the upper bound $C \leq |\mathcal{S}| + m$.

Let $\hat{n}_{i,j,k}$ and $\hat{w}_{i,j,k}$ denote the length and weight of the subsequence $S_{j,k}(\mathbf{v}_i)$, where $\mathbf{v}_i$ denotes the sequence of symbols produced by the source while in state i, with $i \in \{1, 2, \ldots |\mathcal{S}|\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$. It follows that $n_{i,j,k} \geq \hat{n}_{i,j,k}$ and $w_{i,j,k} \geq \hat{w}_{i,j,k}$ for all $i \in \{1, 2, \ldots |\mathcal{S}|\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$. The fact that $log_2(v)$ is an increasing function of v and that

$$\binom{n_{i,j,k}}{w_{i,j,k}} \leq \binom{n_{i,j,k}+1}{w_{i,j,k}+\alpha} \tag{6.263}$$

for $\alpha = 0$ and $\alpha = 1$ is used to simplify equation 6.262 as

$$l(\mathbf{x}) \leq \sum_{i=1}^{|\mathcal{S}|} \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left(\hat{l}_{1,i,j,k}(\mathbf{x}) + \hat{l}_{2,i,j,k}(\mathbf{x}) \right) + \sum_{v=2}^{4} l_v(\mathbf{x}) + mm', \tag{6.264}$$

where

$$\hat{l}_{1,i,j,k}(\mathbf{x}) = \lceil \log_2(\hat{n}_{i,j,k} + 1) \rceil \tag{6.265}$$

and

$$\hat{l}_{2,i,j,k}(\mathbf{x}) = \left\lceil \log_2 \binom{\hat{n}_{i,j,k}}{\hat{w}_{i,j,k}} \right\rceil . \tag{6.266}$$

Equations 6.259 to 6.261, 6.265 and 6.266 are substituted into equation 6.264 to obtain the upper bound

$$\begin{aligned} l(\mathbf{x}) < & \sum_{i=1}^{|\mathcal{S}|} \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \left(\lceil \log_2(\hat{n}_{i,j,k} + 1) \rceil + \left\lceil \log_2 \binom{\hat{n}_{i,j,k}}{\hat{w}_{i,j,k}} \right\rceil \right) \\ & + (C+1) \log_2(n+1) + C + 1 + mm'. \end{aligned} \tag{6.267}$$

The upper bound of equation 6.130 is used to simplify equation 6.267 as

$$l(\mathbf{x}) < \sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{k=0}^{2^{j-1}-1}\left[\log_2\left(\frac{\hat{n}_{i,j,k}(\hat{n}_{i,j,k}+1)}{\sqrt{2\pi\hat{n}_{i,j,k}(\hat{n}_{i,j,k}-1)}}\right)+\left(\frac{1}{12\hat{n}_{i,j,k}}+1\right)\log_2(e) + \hat{n}_{i,j,k}h\left(\frac{\hat{w}_{i,j,k}}{\hat{n}_{i,j,k}}\right)+2\right]+(C+1)\log_2(n+1)+C+1+mm', \quad (6.268)$$

where it is assumed that $\hat{n}_{i,j,k}$ exceeds one bit for all $i \in \{1,2,\ldots|\mathcal{S}|\}$, $j \in \{1,2,\ldots m'\}$ and $k \in \{0,1,\ldots 2^{j-1}-1\}$. The simplified bound of equation 6.268 is generalized to the case where $\hat{n}_{i,j,k} \geq 0$ towards the end of the derivation.

The function

$$v(\hat{n}_{i,j,k}) \triangleq \log_2\left(\frac{\hat{n}_{i,j,k}(\hat{n}_{i,j,k}+1)}{\sqrt{2\pi\hat{n}_{i,j,k}(\hat{n}_{i,j,k}-1)}}\right) \quad (6.269)$$

is a monotonically increasing function of $\hat{n}_{i,j,k}$. This property of the function $v(\hat{n}_{i,j,k})$, as well as the fact that $\hat{n}_{i,j,k} \leq n$, are used to derive the upper bound

$$\log_2\left(\frac{\hat{n}_{i,j,k}(\hat{n}_{i,j,k}+1)}{\sqrt{2\pi\hat{n}_{i,j,k}(\hat{n}_{i,j,k}-1)}}\right) \leq \log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right), \quad (6.270)$$

where $n > 1$. It follows that

$$l(\mathbf{x}) < \sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{k=0}^{2^{j-1}-1}\left[\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right)+\frac{13}{12}\log_2(e)+\hat{n}_{i,j,k}h\left(\frac{\hat{w}_{i,j,k}}{\hat{n}_{i,j,k}}\right)+2\right] + (C+1)\log_2(n+1)+C+1+mm', \quad (6.271)$$

where it is assumed that $\hat{n}_{i,j,k} > 1$ for all $i \in \{1,2,\ldots|\mathcal{S}|\}$, $j \in \{1,2,\ldots m'\}$ and $k \in \{0,1,\ldots 2^{j-1}-1\}$.

It is subsequently proved that the sum within the square brackets of equation 6.271 is a valid upper bound on the lengths of the codewords assigned to subsequences with lengths of zero bits and one bit. The term on the right–hand side of equation 6.270 exceeds zero for all values of n larger than one. As

$$\frac{13}{12}\log_2(e) > 0 \quad (6.272)$$

and

$$\hat{n}_{i,j,k}h\left(\frac{\hat{w}_{i,j,k}}{\hat{n}_{i,j,k}}\right) \geq 0 \quad (6.273)$$

for all[24] $\hat{n}_{i,j,k} \in \{0,1,\ldots n\}$ and $\hat{w}_{i,j,k} \in \{0,1,\ldots \hat{n}_{i,j,k}\}$, it follows that

$$\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right)+\frac{13}{12}\log_2(e)+\hat{n}_{i,j,k}h\left(\frac{\hat{w}_{i,j,k}}{\hat{n}_{i,j,k}}\right)+2 > 2. \quad (6.274)$$

The codewords assigned to subsequences with lengths of zero bits and one bit have lengths of zero bits and one bit, respectively. The sum within the square brackets of

[24]The term $\hat{n}_{i,j,k}h(\hat{w}_{i,j,k}/\hat{n}_{i,j,k})$ is evaluated as zero if $\hat{n}_{i,j,k}$ equals zero.

equation 6.271 exceeds two bits — it follows that equation 6.271 is a valid upper bound on the length of the codeword that is assigned to a source sequence with subsequences $\hat{n}_{i,j,k} \in \{0, 1, \ldots n\}$, where $i \in \{1, 2, \ldots |\mathcal{S}|\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$. Equation 6.271 is simplified as

$$\begin{aligned} l(\mathbf{x}) &< |\mathcal{S}|(2^{m'} - 1)\left[\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right] + (C+1)\log_2(n+1) \\ &\quad + C + 1 + mm' + \sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{k=0}^{2^{j-1}-1} \hat{n}_{i,j,k} h\left(\frac{\hat{w}_{i,j,k}}{\hat{n}_{i,j,k}}\right). \end{aligned} \tag{6.275}$$

Several definitions that are used in the derivation of an upper bound on the average codeword length are subsequently presented. A random n–symbol source sequence $\mathbf{X}^n$ has a random total of N_i'' symbols that were produced by the source while it was in state i, where $i \in \{1, 2, \ldots |\mathcal{S}|\}$. Let $\mathbf{V}_i(\mathbf{X})$ denote the sequence that consists of the symbols produced by the source while it was in state i (as the source sequence $\mathbf{X}$ was produced). The symbol sequence $\mathbf{V}_i(\mathbf{X})$ has subsequences with random lengths $N_{i,j,k}$ and random weights $W_{i,j,k}$, where $i \in \{1, 2, \ldots |\mathcal{S}|\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

Let n_i'' denote an integer from the set $\{0, 1, \ldots n\}$, $n'_{i,j,k}$ denote an integer from the set $\{0, 1, \ldots n_i''\}$ and $w'_{i,j,k}$ denote an integer from the set $\{0, 1, \ldots n'_{i,j,k}\}$. Let the probability distribution $\Pr(n_i'') \triangleq \Pr(N_i'' = n_i'')$ be defined as

$$\Pr(n_i'') = \sum_{\mathbf{x}} \Pr(\mathbf{x}) : \mathbf{x} \in \mathcal{A}^n \wedge |\mathbf{v}_i(\mathbf{x})| = n_i'', \tag{6.276}$$

the probability distribution $\Pr(n'_{i,j,k}) \triangleq \Pr(N_{i,j,k} = n'_{i,j,k})$ be defined as

$$\Pr(n'_{i,j,k}) = \sum_{\mathbf{x}} \Pr(\mathbf{x}) : \mathbf{x} \in \mathcal{A}^n \wedge |S_{j,k}(\mathbf{v}_i(\mathbf{x}))| = n'_{i,j,k} \tag{6.277}$$

and the probability distribution $\Pr(w'_{i,j,k}) \triangleq \Pr(W_{i,j,k} = w'_{i,j,k})$ be defined as

$$\Pr(w'_{i,j,k}) = \sum_{\mathbf{x}} \Pr(\mathbf{x}) : \mathbf{x} \in \mathcal{A}^n \wedge w_{j,k}(\mathbf{v}_i(\mathbf{x})) = w'_{i,j,k}. \tag{6.278}$$

The ordered multisets $\mathbf{N}, \mathbf{W}, \mathbf{N}'', \mathbf{n}', \mathbf{w}'$ and $\mathbf{n}''$ are defined as

$$\mathbf{N} \triangleq \{N_{i,j,k} : 1 \leq i \leq |\mathcal{S}| \wedge 1 \leq j \leq m' \wedge 0 \leq k \leq 2^{j-1} - 1\}, \tag{6.279}$$

$$\mathbf{W} \triangleq \{W_{i,j,k} : 1 \leq i \leq |\mathcal{S}| \wedge 1 \leq j \leq m' \wedge 0 \leq k \leq 2^{j-1} - 1\}, \tag{6.280}$$

$$\mathbf{N}'' \triangleq \{N_i'' : 1 \leq i \leq |\mathcal{S}|\}, \tag{6.281}$$

$$\mathbf{n}' \triangleq \{n'_{i,j,k} : 1 \leq i \leq |\mathcal{S}| \wedge 1 \leq j \leq m' \wedge 0 \leq k \leq 2^{j-1} - 1\}, \tag{6.282}$$

$$\mathbf{w}' \triangleq \{w'_{i,j,k} : 1 \leq i \leq |\mathcal{S}| \wedge 1 \leq j \leq m' \wedge 0 \leq k \leq 2^{j-1} - 1\} \tag{6.283}$$

and

$$\mathbf{n}'' = \{n_i'' : 1 \leq i \leq |\mathcal{S}|\}. \tag{6.284}$$

The ordered multisets appear in the expressions for the joint probability distributions that involve the subsequences of a random source sequence. The joint probability distribution of the sequence lengths $|\mathbf{V}_i(\mathbf{X})|$ is expressed as $\Pr(\mathbf{n}'') \triangleq \Pr(\mathbf{N}'' = \mathbf{n}'')$. The joint probability distribution of the lengths of a random source sequence's subsequences is expressed as $\Pr(\mathbf{n}') \triangleq \Pr(\mathbf{N} = \mathbf{n}')$. The joint probability distribution of the weights of a random source sequence's subsequences is expressed as $\Pr(\mathbf{w}') \triangleq \Pr(\mathbf{W} = \mathbf{w}')$.

The average length of the codewords that are assigned to source sequences from a q–ary context–tree source is subsequently derived. The average codeword length $L(\mathbf{X}^n)$ may be expressed as

$$\begin{aligned} L(\mathbf{X}^n) &= \sum_{\mathbf{x}\in\mathcal{A}^n} \Pr(\mathbf{x}) l(\mathbf{x}) \\ &\leq \sum_{\mathbf{n}''\in\mathbf{q}_1} \sum_{\mathbf{n}'\in\mathbf{q}_2} \sum_{\mathbf{w}'\in\mathbf{q}_2} \Pr(\mathbf{n}'', \mathbf{n}', \mathbf{w}') l(\mathbf{n}'', \mathbf{n}', \mathbf{w}') \\ &\leq \sum_{\mathbf{n}''\in\mathbf{q}_1} \sum_{\mathbf{n}'\in\mathbf{q}_2} \sum_{\mathbf{w}'\in\mathbf{q}_2} \Pr(\mathbf{n}'', \mathbf{n}') \Pr(\mathbf{w}'|\mathbf{n}'', \mathbf{n}') l(\mathbf{n}'', \mathbf{n}', \mathbf{w}'), \end{aligned} \tag{6.285}$$

where $\mathbf{q}_1 \triangleq \{0, 1, \ldots n\}^{|\mathcal{S}|}$ and $\mathbf{q}_2 \triangleq \{0, 1, \ldots n\}^{|\mathcal{S}|(2^{m'}-1)}$. The function $l(\mathbf{n}'', \mathbf{n}', \mathbf{w}')$ of equation 6.285 denotes an upper bound on the length of each codeword that is assigned to a source sequence $\mathbf{x}^n$ that satisfies $\mathbf{v}_i(\mathbf{x}) = n_i''$ for all $i \in \{1, 2, \ldots |\mathcal{S}|\}$, and with subsequences $S_{j,k}(\mathbf{v}_i(\mathbf{x}))$ having lengths equal to $n'_{i,j,k}$ and weights equal to $w'_{i,j,k}$ for all $i \in \{1, 2, \ldots |\mathcal{S}|\}$, $j \in \{1, 2, \ldots m'\}$ and $k \in \{0, 1, \ldots 2^{j-1} - 1\}$.

The upper bound of equation 6.275 is substituted into equation 6.285 to obtain the upper bound

$$\begin{aligned} L(\mathbf{X}^n) &< \sum_{i=1}^{|\mathcal{S}|} \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{\mathbf{n}''\in\mathbf{q}_1} \sum_{\mathbf{n}'\in\mathbf{q}_2} \sum_{\mathbf{w}'\in\mathbf{q}_2} \Pr(\mathbf{n}'', \mathbf{n}') \Pr(\mathbf{w}'|\mathbf{n}'', \mathbf{n}') n'_{i,j,k} h\left(\frac{w'_{i,j,k}}{n'_{i,j,k}}\right) \\ &\quad + |\mathcal{S}|(2^{m'} - 1)\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right) \\ &\quad + (C+1)\log_2(n+1) + C + 1 + mm' \\ &< \sum_{i=1}^{|\mathcal{S}|} \sum_{j=1}^{m'} \sum_{k=0}^{2^{j-1}-1} \sum_{n_i''=1}^{n} \sum_{n'_{i,j,k}=0}^{n_i''} \Pr(n_i'', n'_{i,j,k}) n'_{i,j,k} \left[\sum_{w'_{i,j,k}=0}^{n'_{i,j,k}} \Pr(w'_{i,j,k}|n_i'', n'_{i,j,k}) h\left(\frac{w'_{i,j,k}}{n'_{i,j,k}}\right)\right] \\ &\quad + |\mathcal{S}|(2^{m'} - 1)\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right) \\ &\quad + (C+1)\log_2(n+1) + C + 1 + mm'. \end{aligned} \tag{6.286}$$

The function $\Pr(w'_{i,j,k}|n_i'', n'_{i,j,k})$ is the probability mass function of a binomial random variable with parameters $n'_{i,j,k}$ and $p_{i,j,k}$. The parameter $p_{i,j,k}$ is defined as

$$p_{i,j,k} \triangleq \Pr(B_j^{(i)} = 1 | \mathbf{B}_1^{j-1,(i)} = \beta_{j-1}(k)), \tag{6.287}$$

where $B_j^{(i)}$ denotes bit j of the binary word that is assigned to a random symbol that the source produces in state i, and where $\mathbf{B}_1^{j,(i)}$ denotes bits 1 to j of the same binary word. The probability mass function is defined for all $w'_{i,j,k} \in \{0, 1, \ldots n'_{i,j,k}\}$.

Jensen's inequality is used to simplify the term within the square brackets of equation 6.286, as the binary entropy function $h(\cdot)$ is a concave function. The term is simplified as

$$\begin{aligned}\sum_{w'_{i,j,k}=0}^{n'_{i,j,k}} \Pr(w'_{i,j,k}|n''_i, n'_{i,j,k}) h\left(\frac{w'_{i,j,k}}{n'_{i,j,k}}\right) &\leq h\left(\sum_{w'_{i,j,k}=0}^{n'_{i,j,k}} \Pr(w'_{i,j,k}|n''_i, n'_{i,j,k}) \frac{w'_{i,j,k}}{n'_{i,j,k}}\right) \\ &\leq h(p_{i,j,k}). \end{aligned} \tag{6.288}$$

The final step of equation 6.288 follows from the fact that the expected value of the binomial random variable equals $n'_{i,j,k}p_{i,j,k}$.

The first term on the right–hand side of equation 6.286 is simplified as

$$\begin{aligned}&\sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{k=0}^{2^{j-1}-1}\sum_{n''_i=0}^{n}\sum_{n'_{i,j,k}=0}^{n''_i} \Pr(n''_i, n'_{i,j,k}) n'_{i,j,k} \left[\sum_{w'_{i,j,k}=0}^{n'_{i,j,k}} \Pr(w'_{i,j,k}|n''_i, n'_{i,j,k}) h\left(\frac{w'_{i,j,k}}{n'_{i,j,k}}\right)\right] \\ \leq &\sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{k=0}^{2^{j-1}-1}\sum_{n''_i=0}^{n}\sum_{n'_{i,j,k}=0}^{n''_i} \Pr(n''_i, n'_{i,j,k}) n'_{i,j,k} h(p_{i,j,k}) \\ \leq &\sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{k=0}^{2^{j-1}-1}\sum_{n''_i=0}^{n} \Pr(n''_i) \sum_{n'_{i,j,k}=0}^{n''_i} \Pr(n'_{i,j,k}|n''_i) n'_{i,j,k} h(p_{i,j,k}). \end{aligned} \tag{6.289}$$

The function $\Pr(n'_{i,j,k}|n''_i)$ is the probability mass function of a binomial random variable with parameters n''_i and $p''_{i,j,k}$, where

$$p''_{i,j,k} \triangleq \Pr(\mathbf{B}_1^{j-1,(i)} = \beta_{j-1}(k)). \tag{6.290}$$

It is defined for all $n'_{i,j,k} \in \{0, 1, \ldots n''_i\}$.

Equation 6.289 is simplified as

$$\begin{aligned}&\sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{k=0}^{2^{j-1}-1}\sum_{n''_i=0}^{n} \Pr(n''_i) \sum_{n'_{i,j,k}=0}^{n''_i} \Pr(n'_{i,j,k}|n''_i) n'_{i,j,k} h(p_{i,j,k}) \\ = &\sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{k=0}^{2^{j-1}-1}\sum_{n''_i=0}^{n} \Pr(n''_i) n''_i p''_{i,j,k} h(p_{i,j,k}). \end{aligned} \tag{6.291}$$

The first step of equation 6.291 follows from the fact that the expected value of the binomial random variable equals $n''_i p''_{i,j,k}$.

The term on the right–hand side of equation 6.288 may be rewritten as

$$h(p_{i,j,k}) = H(B_j^{(i)}|\mathbf{B}_1^{j-1,(i)} = \beta_{j-1}(k)) \tag{6.292}$$

and substituted into equation 6.291 to obtain the expression

$$\begin{aligned}
&\sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{k=0}^{2^{j-1}-1}\sum_{n_i''=0}^{n}\Pr(n_i'')n_i''p_{i,j,k}''h(p_{i,j,k}) \\
&= \sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{n_i''=0}^{n}\Pr(n_i'')n_i''\sum_{k=0}^{2^{j-1}-1}\Pr(\mathbf{B}_1^{j-1,(i)}=\beta_{j-1}(k))H(B_j^{(i)}|\mathbf{B}_1^{j-1,(i)}=\beta_{j-1}(k)) \\
&= \sum_{i=1}^{|\mathcal{S}|}\sum_{j=1}^{m'}\sum_{n_i''=0}^{n}\Pr(n_i'')n_i''H(B_j^{(i)}|\mathbf{B}_1^{j-1,(i)}) \\
&= \sum_{i=1}^{|\mathcal{S}|}\sum_{n_i''=0}^{n}\Pr(n_i'')n_i''\sum_{j=1}^{m'}H(B_j^{(i)}|\mathbf{B}_1^{j-1,(i)}) \\
&= \sum_{i=1}^{|\mathcal{S}|}\sum_{n_i''=0}^{n}\Pr(n_i'')n_i''H(\mathbf{B}_1^{m',(i)}) \\
&= \sum_{i=1}^{|\mathcal{S}|}\sum_{n_i''=0}^{n}\Pr(n_i'')n_i''H(X^{(i)}), \qquad (6.293)
\end{aligned}$$

where $H(X^{(i)})$ denotes the entropy associated with each symbol that the context–tree source produces in state i.

The function $\Pr(n_i'')$ is the probability mass function of a binomial random variable with parameters n and q_i, where q_i denotes the probability of the context–tree source being in state i. It is defined for all $n_i'' \in \{0, 1, \ldots n\}$.

Equation 6.293 is simplified as

$$\begin{aligned}
&\sum_{i=1}^{|\mathcal{S}|}\sum_{n_i''=0}^{n}\Pr(n_i'')n_i''H(X^{(i)}) \\
&= n\sum_{i=1}^{|\mathcal{S}|}q_iH(X^{(i)}) \\
&= nH(\mathcal{X}), \qquad (6.294)
\end{aligned}$$

where $H(\mathcal{X})$ denotes the entropy rate of the context–tree source. The final step of equation 6.294 follows from the fact that the expected value of the binomial random variable equals nq_i.

A bound on the average length $L(\mathbf{X}^n)$ of a codeword assigned to an n–symbol source sequence by the encoder of the universal weight–and–index source code for q–ary context–tree sources is derived using equations 6.285 to 6.294 as

$$\begin{aligned}
L(\mathbf{X}^n) \;<\; & nH(\mathcal{X}) + |\mathcal{S}|(2^{m'}-1)\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right)+\frac{13}{12}\log_2(e)+2\right) \\
& + (C+1)\log_2(n+1) + C + 1 + mm'. \qquad (6.295)
\end{aligned}$$

The normalized average per–codeword redundancy of the universal weight–and–index source code for q–ary context–tree sources, with respect to the entropy rate of the source[25], is bounded as

$$\begin{aligned} R'(\mathbf{X}^n) &= \frac{1}{n}L(\mathbf{X}^n) - H(\mathcal{X}) \\ &< \frac{|\mathcal{S}|(2^{m'}-1)}{n}\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right) \\ &\quad + \frac{(C+1)\log_2(n+1)}{n} + \frac{C+1+mm'}{n}. \end{aligned} \tag{6.296}$$

The limit of the normalized average per–codeword redundancy, as the source sequence length n tends to infinity, is derived as follows. The normalized average per–codeword redundancy satisfies the inequality

$$\begin{aligned} 0 \leq R'(\mathbf{X}^n) < \frac{|\mathcal{S}|(2^{m'}-1)}{n}\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right) \\ + \frac{(C+1)\log_2(n+1)}{n} + \frac{C+1+mm'}{n}. \end{aligned} \tag{6.297}$$

for all $n > 1$. The normalized average per–codeword redundancy is therefore expressed as

$$R'(\mathbf{X}^n) = \frac{1}{n}K_n q(n), \tag{6.298}$$

where

$$\begin{aligned} q(n) &= |\mathcal{S}|(2^{m'}-1)\left(\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) + \frac{13}{12}\log_2(e) + 2\right) \\ &\quad + (C+1)\log_2(n+1) + C + 1 + mm' \end{aligned} \tag{6.299}$$

and $0 \leq K_n < 1$ for all $n > 1$. The limit[26]

$$\begin{aligned} \lim_{n\to\infty}\frac{1}{n}q(n) &= \lim_{n\to\infty}\frac{|\mathcal{S}|(2^{m'}-1)}{n}\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) \\ &= \lim_{n\to\infty}\frac{1}{n}\log_2\left(\frac{n(n+1)}{\sqrt{2\pi n(n-1)}}\right) \\ &= 0 \end{aligned} \tag{6.300}$$

is used to prove that

$$\begin{aligned} \lim_{n\to\infty} R'(\mathbf{X}^n) &= \lim_{n\to\infty}\frac{1}{n}K_n q(n) \\ &= 0. \end{aligned} \tag{6.301}$$

The limit of the normalized average per–codeword redundancy of the universal weight–and–index source code for q–ary context–tree sources, as the source sequence length tends to infinity, equals zero. It is concluded that the source code is universal.

[25] The approach of deriving the redundancy with respect to the entropy rate of the source was used by Effros et. al. [10].

[26] The second step of equation 6.300 follows from equation 6.138.

6.2.2.5 Practical results

Practical implementations of the weight–and–index source codes for q–ary memoryless sources and q–ary context–tree sources were used to encode sequences from their respective sources. The source coding results for each source type are presented in this section.

Q–ary memoryless sources The universal weight–and–index source code for q–ary memoryless sources, which is referred to as the proposed source code for the remainder of this section, was used to encode sequences from octonary memoryless sources. The symbol alphabet of these sources contains eight symbols, and the symbols are numbered from one to eight. A maximum of

$$\begin{aligned} m' &= \log_2(q) \\ &= 3 \end{aligned} \tag{6.302}$$

bits are required to represent each alphabet symbol.

Source sequences from two octonary memoryless sources with stationary symbol distributions were encoded in order to investigate the normalized average per–codeword redundancy of the proposed source code. The two octonary memoryless sources are referred to as source one and source two. The length of the source sequences were varied between sets of simulation trials in order to determine the rate at which the normalized redundancy of the code decreases w.r.t. the source sequence length. The practical implementation of the proposed source code had no apriori knowledge of each source's symbol distribution. The parameters and quantities associated with the simulation that involves the proposed source code are presented in table 6.12 on page 229.

The source coding results associated with the proposed source code are presented in figure 6.18 on page 230. This figure contains a plot of the normalized average per–codeword redundancy of the proposed source code, as a function of the source sequence length n. A curve that represents an upper bound on the normalized average per–codeword redundancy of the proposed source code is included in the figure. The upper bound was derived by normalizing equation 6.250 on page 219 with the source sequence length n.

Figure 6.18 reveals that the upper bound on the redundancy of the proposed source code is not exceeded by the redundancy that is associated with the practical implementation of the proposed source code. Both of the curves that represent the redundancy of the practical implementation decrease w.r.t. the source sequence length, as expected. The redundancy of the practical implementation appears to decrease at the same rate w.r.t. the source sequence length as the upper bound on the redundancy of the source code. These observations imply that the source code was correctly implemented.

The curves that are associated with the source coding of sequences from source one and source two nearly overlap. As source one has a symbol distribution that is significantly less uniform than source two, this observation suggests that the effectiveness of the proposed source code is not overly sensitive to the symbol distribution of the source.

Table 6.12: Parameters and quantities of the simulation that involves the weight–and–index source code for q–ary memoryless sources.

Parameter / Quantity	Symbol	Value, src. 1	Value, src. 2	Unit
Src. sequence length	n	256 to 8192	256 to 8192	symbols
Prob. alphabet symbol 1	$\Pr(X = a_1)$	0.0060	0.1984	—
Prob. alphabet symbol 2	$\Pr(X = a_2)$	0.1080	0.1081	—
Prob. alphabet symbol 3	$\Pr(X = a_3)$	0.0480	0.1247	—
Prob. alphabet symbol 4	$\Pr(X = a_4)$	0.3240	0.0482	—
Prob. alphabet symbol 5	$\Pr(X = a_5)$	0.0140	0.1542	—
Prob. alphabet symbol 6	$\Pr(X = a_6)$	0.4320	0.1293	—
Prob. alphabet symbol 7	$\Pr(X = a_7)$	0.0320	0.1238	—
Prob. alphabet symbol 8	$\Pr(X = a_8)$	0.0360	0.1132	—
Per–symbol src. entropy	$H(X)$	2.0690	2.9219	bits

Q–ary context–tree sources The three weight–and–index source codes for q–ary context–tree sources were implemented and used to encode sequences from quaternary context–tree sources. The symbol alphabet $\mathcal{A}$ of these sources contains four symbols, and is defined as

$$\mathcal{A} = \{0, 1, 2, 3\}. \tag{6.303}$$

A maximum of

$$\begin{aligned} m' &= \log_2(q) \\ &= 2 \end{aligned} \tag{6.304}$$

bits are required to represent each symbol of the alphabet.

Sequences from two quaternary context–tree sources were encoded using the weight–and–index source codes for context–tree sources. The two sources are referred to as source one and source two. Each of the two quaternary context–tree sources is defined in what follows.

Source 1 The tree of the first quaternary context–tree source is presented in figure 6.19 on page 231. The first context–tree source has a total of 13 states; its state set $\mathcal{S}$ is defined as

$$\mathcal{S} = \{00, 01, 02, 03, 1, 200, 201, 202, 203, 21, 22, 23, 3\}. \tag{6.305}$$

The states of the first context–tree source are numbered from one to thirteen as indicated in table 6.13 on page 232. Table 6.13 includes the distribution of the symbols that the first context–tree source produces in each of its states.

Let m denote the length of the longest context that is associated with the first context–tree source. It follows that

$$m = 3, \tag{6.306}$$

as the source has no context longer than three symbols.

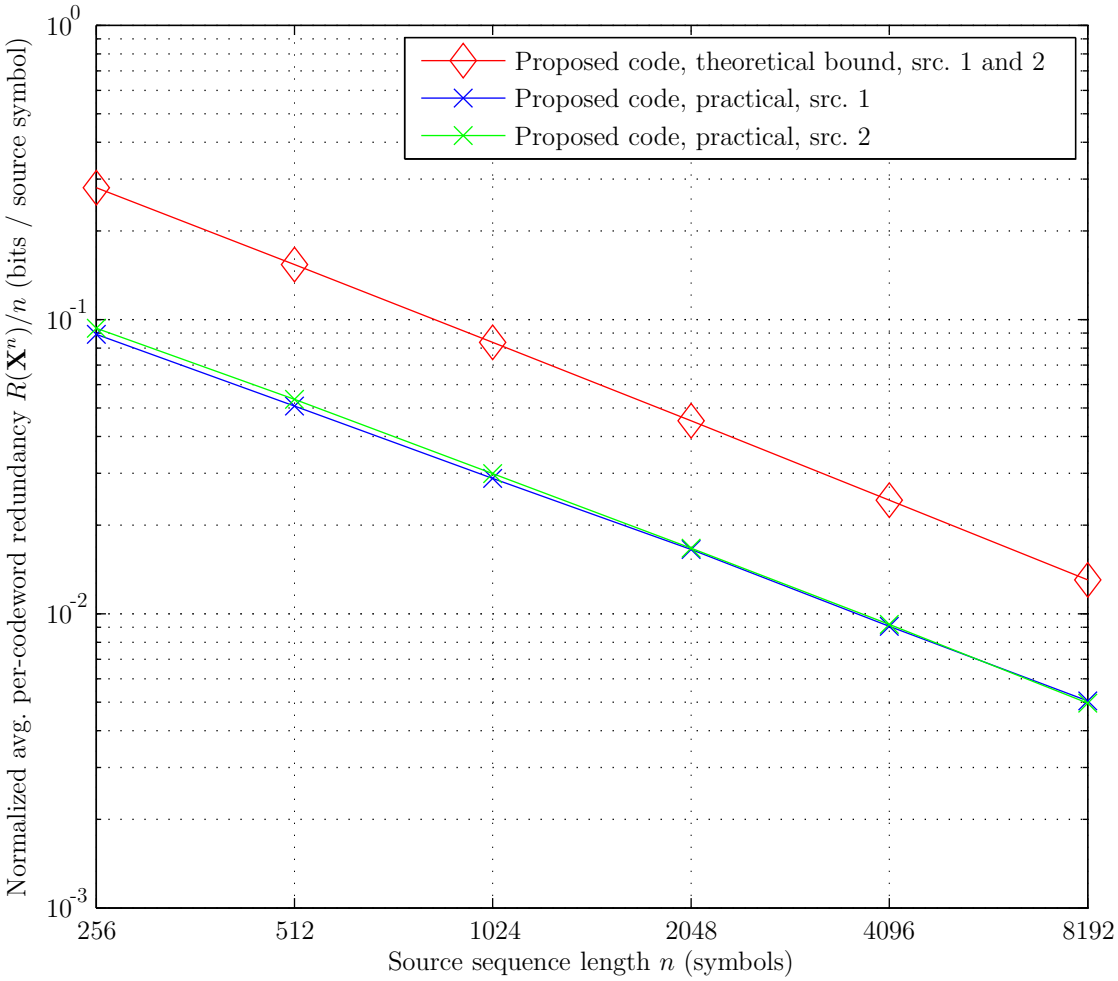


Figure 6.18: Normalized average per–codeword redundancy of the weight–and–index source code for q–ary memoryless sources, as a function of the source sequence length n. The figure includes a curve that represents an upper bound on the redundancy of the source code.

The FSM closure of the first quaternary context–tree source is subsequently derived. Each state of the first context–tree source's FSM closure is the reverse of the corresponding state of the context–tree source. The states of the FSM closure of the first context–tree source are provided in table 6.13 on page 232.

The state–transition diagram of the first context–tree source's FSM closure is not provided, due to the excessive number of transitions between its states. A more compact representation of the source states and the state transitions is provided instead. The

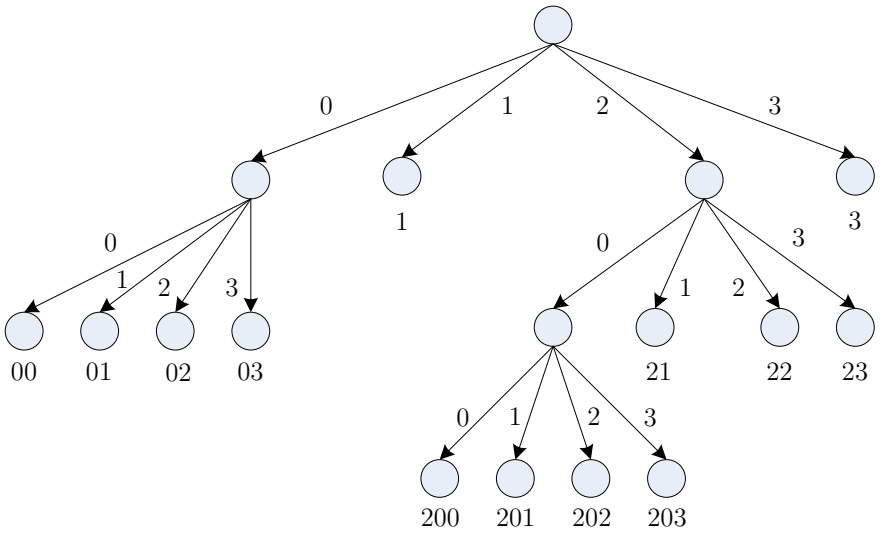


Figure 6.19: The tree of the first quaternary context–tree source.

compact transition matrix of the first context–tree source's FSM closure is obtained as

$$\mathbf{S}_{\text{tr,c}} \triangleq \begin{pmatrix} t_{1,1} & t_{1,2} & \dots & t_{1,|\mathcal{S}'|} \\ t_{2,1} & t_{2,2} & \dots & t_{2,|\mathcal{S}'|} \\ \vdots & \vdots & \ddots & \vdots \\ t_{q,1} & t_{q,2} & \dots & t_{q,|\mathcal{S}'|} \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 & 11 & 11 & 11 & 11 & 11 & 11 & 12 \\ 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 \end{pmatrix}, \quad (6.307)$$

where $t_{x,y}$ denotes the destination state of the transition that occurs when the symbol x is produced in the state $\mathbf{s}'_y$.

The probability of the first context–tree source's FSM closure residing in each of its states is calculated using the equation

$$\mathbf{S}^T\mathbf{P} = \mathbf{P}, \quad (6.308)$$

where $\mathbf{S}$ denotes the state–transition probability matrix of the FSM closure (refer to equation 6.90 on page 173), and $\mathbf{P}$ denotes the state probability vector of the FSM closure (refer to equation 6.91 on page 173). The elements of the state probability vector are obtained as

$$\begin{array}{lll} \Pr(\mathbf{s}'_1) = 0.0660 & \Pr(\mathbf{s}'_2) = 0.0659 & \Pr(\mathbf{s}'_3) = 0.0668 \\ \Pr(\mathbf{s}'_4) = 0.0495 & \Pr(\mathbf{s}'_5) = 0.2544 & \Pr(\mathbf{s}'_6) = 0.0133 \\ \Pr(\mathbf{s}'_7) = 0.0267 & \Pr(\mathbf{s}'_8) = 0.0095 & \Pr(\mathbf{s}'_9) = 0.0101 \\ \Pr(\mathbf{s}'_{10}) = 0.0988 & \Pr(\mathbf{s}'_{11}) = 0.0731 & \Pr(\mathbf{s}'_{12}) = 0.0561 \\ \Pr(\mathbf{s}'_{13}) = 0.2097. & & \end{array} \quad (6.309)$$

Table 6.13: The symbol distributions that are associated with the states of the first context–tree source. The table includes the states of the FSM closure of the context–tree source.

CT state	FSM state	$\Pr(X=0\|\mathbf{s}_i)$	$\Pr(X=1\|\mathbf{s}_i)$	$\Pr(X=2\|\mathbf{s}_i)$	$\Pr(X=3\|\mathbf{s}_i)$
$\mathbf{s}_1 = 00$	$\mathbf{s}'_1 = 00$	0.2884	0.3092	0.2012	0.2012
$\mathbf{s}_2 = 01$	$\mathbf{s}'_2 = 10$	0.3916	0.0160	0.4056	0.1868
$\mathbf{s}_3 = 02$	$\mathbf{s}'_3 = 20$	0.0336	0.6901	0.1432	0.1331
$\mathbf{s}_4 = 03$	$\mathbf{s}'_4 = 30$	0.3820	0.1604	0.2043	0.2533
$\mathbf{s}_5 = 1$	$\mathbf{s}'_5 = 1$	0.2590	0.2461	0.3886	0.1064
$\mathbf{s}_6 = 200$	$\mathbf{s}'_6 = 002$	0.1367	0.0688	0.2350	0.5595
$\mathbf{s}_7 = 201$	$\mathbf{s}'_7 = 102$	0.0940	0.2972	0.2781	0.3308
$\mathbf{s}_8 = 202$	$\mathbf{s}'_8 = 202$	0.4238	0.1502	0.1583	0.2677
$\mathbf{s}_9 = 203$	$\mathbf{s}'_9 = 302$	0.1289	0.3295	0.3210	0.2206
$\mathbf{s}_{10} = 21$	$\mathbf{s}'_{10} = 12$	0.1610	0.4166	0.1430	0.2794
$\mathbf{s}_{11} = 22$	$\mathbf{s}'_{11} = 22$	0.1440	0.2017	0.3714	0.2828
$\mathbf{s}_{12} = 23$	$\mathbf{s}'_{12} = 32$	0.5461	0.1308	0.2944	0.0287
$\mathbf{s}_{13} = 3$	$\mathbf{s}'_{13} = 3$	0.2362	0.1879	0.2675	0.3084

The entropy $H_y(X)$ that is associated with the distribution of the symbols produced in state $\mathbf{s}'_y$ of the first context–tree source's FSM closure is calculated using the equation

$$H_y(X) = -\sum_{z=1}^{|\mathcal{S}'|} p_{y,z} \log_2(p_{y,z}), \tag{6.310}$$

where $p_{y,z}$ denotes the probability of a transition from state $\mathbf{s}'_y$ to state $\mathbf{s}'_z$ of the FSM closure, conditioned on the FSM closure being in state s'_y. The entropy rate $H(\mathcal{X})$ of the FSM closure of the first quaternary context–tree source is subsequently calculated as

$$\begin{aligned} H(\mathcal{X}) &= \sum_{y=1}^{|\mathcal{S}'|} p_y H_y(X) \\ &= 1.8305 \end{aligned} \tag{6.311}$$

bits per source symbol, where $p_y \triangleq \Pr(\mathbf{s}'_y)$. The entropy rate of the first quaternary context–tree source equals the entropy rate of its FSM closure.

Source 2 The second quaternary context–tree source has the same tree as the first quaternary context–tree source (figure 6.19 on page 231), and therefore the same state set as the first quaternary context–tree source (equation 6.305). The states of the second quaternary context–tree source are numbered in the same manner as the states of the first quaternary context–tree source (refer to table 6.14 on page 233).

The distribution of the symbols that the second context–tree source produces in each of its states differs from the corresponding distribution of the first context–tree

Table 6.14: The symbol distributions that are associated with the states of the second context–tree source. The table includes the states of the FSM closure of the context–tree source.

CT state	FSM state	$\Pr(X = 0\|\mathbf{s}_i)$	$\Pr(X = 1\|\mathbf{s}_i)$	$\Pr(X = 2\|\mathbf{s}_i)$	$\Pr(X = 3\|\mathbf{s}_i)$
$\mathbf{s}_1 = 00$	$\mathbf{s}'_1 = 00$	0.0151	0.6013	0.2052	0.1783
$\mathbf{s}_2 = 01$	$\mathbf{s}'_2 = 10$	0.2985	0.4455	0.1238	0.1322
$\mathbf{s}_3 = 02$	$\mathbf{s}'_3 = 20$	0.3436	0.2088	0.1071	0.3405
$\mathbf{s}_4 = 03$	$\mathbf{s}'_4 = 30$	0.4979	0.1056	0.0010	0.3955
$\mathbf{s}_5 = 1$	$\mathbf{s}'_5 = 1$	0.5346	0.0729	0.0240	0.3685
$\mathbf{s}_6 = 200$	$\mathbf{s}'_6 = 002$	0.3326	0.2842	0.0041	0.3791
$\mathbf{s}_7 = 201$	$\mathbf{s}'_7 = 102$	0.4136	0.3819	0.0495	0.1550
$\mathbf{s}_8 = 202$	$\mathbf{s}'_8 = 202$	0.1410	0.3266	0.3571	0.1753
$\mathbf{s}_9 = 203$	$\mathbf{s}'_9 = 302$	0.2928	0.2919	0.2865	0.1288
$\mathbf{s}_{10} = 21$	$\mathbf{s}'_{10} = 12$	0.0283	0.3694	0.5176	0.0847
$\mathbf{s}_{11} = 22$	$\mathbf{s}'_{11} = 22$	0.0481	0.3868	0.2041	0.3609
$\mathbf{s}_{12} = 23$	$\mathbf{s}'_{12} = 32$	0.6239	0.1439	0.1904	0.0417
$\mathbf{s}_{13} = 3$	$\mathbf{s}'_{13} = 3$	0.3163	0.4038	0.2655	0.0144

source. The distribution of the symbols that the second context–tree source produces in each state is presented in table 6.14 on page 233.

The FSM closure of the second quaternary context–tree source has the same state set as the FSM closure of the first quaternary context–tree source, as well as the same compact transition matrix as the FSM closure of the first quaternary context–tree source (equation 6.307). The states of the FSM closure of the second context–tree source are numbered in the same manner as the states of the FSM closure of the first context–tree source (refer to table 6.14 on page 233).

The elements of the state probability vector $\mathbf{P}$ of the second context–tree source's FSM closure are obtained as

$$\begin{array}{lll} \Pr(\mathbf{s}'_1) = 0.0985 & \Pr(\mathbf{s}'_2) = 0.1530 & \Pr(\mathbf{s}'_3) = 0.0519 \\ \Pr(\mathbf{s}'_4) = 0.0674 & \Pr(\mathbf{s}'_5) = 0.2861 & \Pr(\mathbf{s}'_6) = 0.0202 \\ \Pr(\mathbf{s}'_7) = 0.0189 & \Pr(\mathbf{s}'_8) = 0.0056 & \Pr(\mathbf{s}'_9) = 0.0001 \\ \Pr(\mathbf{s}'_{10}) = 0.0069 & \Pr(\mathbf{s}'_{11}) = 0.0218 & \Pr(\mathbf{s}'_{12}) = 0.0566 \\ \Pr(\mathbf{s}'_{13}) = 0.2130 & & \end{array} \tag{6.312}$$

using equation 6.308. The entropy rate $H(\mathcal{X})$ of the FSM closure of the second context–tree source is subsequently calculated as

$$H(\mathcal{X}) = 1.5720 \tag{6.313}$$

bits per source symbol using equations 6.310 and 6.311. The entropy rate of the second quaternary context–tree source equals the entropy rate of its FSM closure.

Source coding results The three weight–and–index source codes for q–ary context–tree sources were implemented and used to encode sequences from the first and second

Table 6.15: Parameters and quantities associated with the first and second quaternary context–tree sources.

Parameter / Quantity	Symbol	Value	Unit
Source sequence length	n	256 – 8192	symbols
Entropy rate, src. 1	$H(\mathcal{X})$	1.8305	bits per source symbol
Entropy rate, src. 2	$H(\mathcal{X})$	1.5720	bits per source symbol

Table 6.16: Block lengths and parameters associated with the segmentation algorithm of the universal weight–and–index source code for q–ary context–tree sources.

Source sequence length n (symbols)	Level–one block length $k_1(n)$ (symbols)	Level–two block length $k_2(n)$ (symbols)	Parameter μ	Parameter γ
256	75	15	1.0763	1.4650
512	100	20	1.0959	1.5966
1024	126	21	1.1004	1.5359
2048	150	25	1.0896	1.5936
4096	180	30	1.0898	1.6640
8192	210	30	1.0847	1.5994

quaternary context–tree sources. These source codes are the type–one and type–two nonuniversal weight–and–index source codes for q–ary context–tree sources (introduced in section 6.2.2.1 on pages 204 and 207), as well as the universal weight–and–index source code for q–ary context–tree sources (introduced in section 6.2.2.1 on page 208). The length of the source sequences that were encoded was varied between sets of simulation trials in order to investigate the rate at which the normalized average per–codeword redundancy of each source code decreases w.r.t. the source sequence length.

The parameters and quantities associated with the first and second quaternary context–tree sources are presented in table 6.15 on page 234. Table 6.16 on page 234 presents the block lengths and parameters that are associated with the practical implementation of the segmentation algorithm, which is part of the universal weight–and–index source code for q–ary context–tree sources. The parameters μ and γ, as defined in equations 6.99 and 6.100, were assigned values by trial and error.

The source coding results are presented in figure 6.20 on page 235. This figure contains a plot of the normalized average per–codeword redundancy of each source code's practical implementation, as a function of the source sequence length n. The figure includes a curve that represents the analytically–derived upper bound on the redundancy of the universal weight–and–index source code. This bound was derived under the assumption that the segmentation algorithm of the universal source code is perfectly accurate, and is expressed in equation 6.296 on page 227.

Figure 6.20 reveals that the normalized redundancy of each practical source code implementation does not exceed the upper bound on the normalized redundancy of the universal source code (with perfect segmentation). The normalized redundancy of

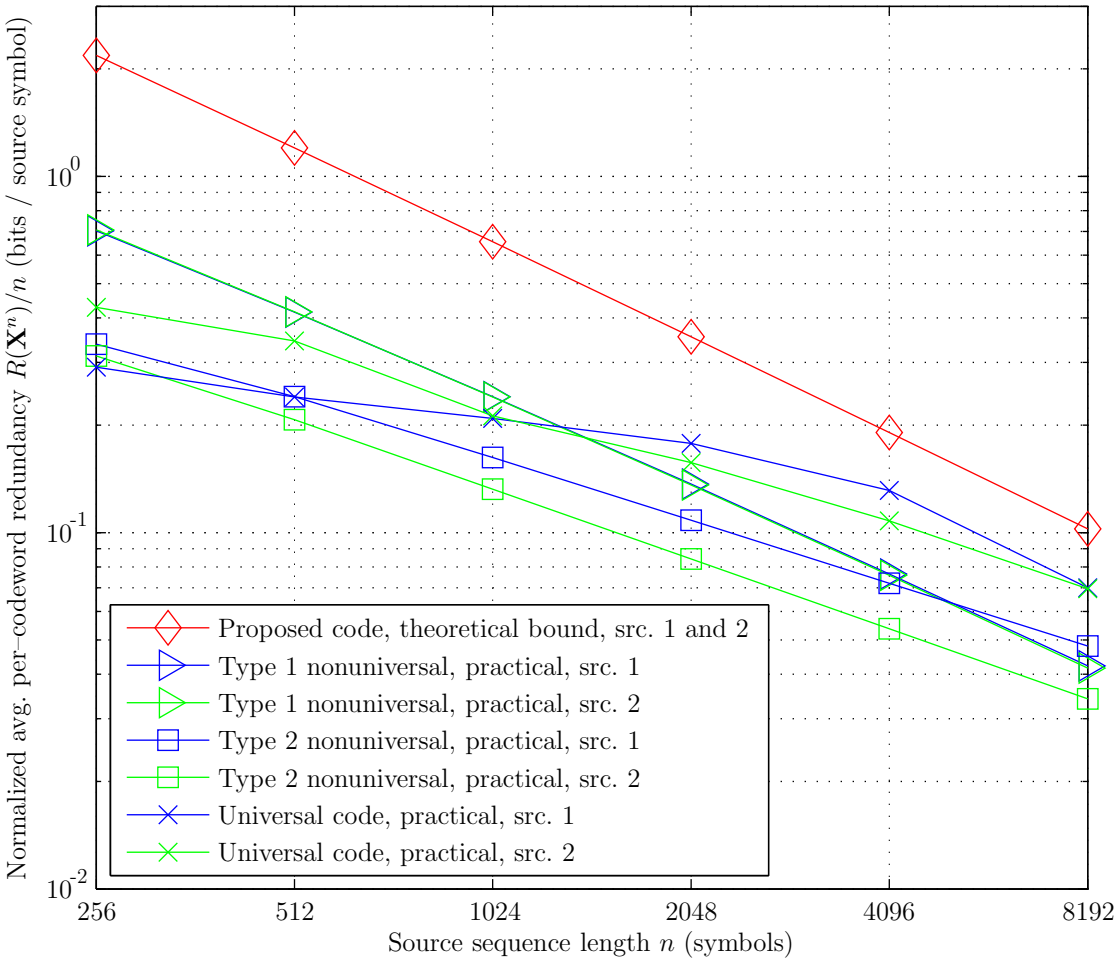


Figure 6.20: The normalized average per–codeword redundancy of the nonuniversal and universal weight–and–index source codes for q–ary context–tree sources, as a function of the source sequence length n. The figure includes a curve that represents a bound on the redundancy of the universal source code (where it is assumed that the segmentation algorithm is perfectly accurate).

the type–one nonuniversal weight–and–index source code decreases at approximately the same rate w.r.t. the source sequence length as the upper bound. The encoder of the type–one nonuniversal weight–and–index source code produces codewords that are of approximately[27] the same length as the codewords that would be produced by the encoder of the universal weight–and–index source code (assuming perfectly accurate segmentation). It is therefore concluded that the type–one nonuniversal weight–and–index source code was correctly implemented.

The curves associated with the redundancy of the type–one nonuniversal source code, when used to encode sequences from the first context–tree source and the second

[27]The difference between the lengths of the codewords arises from the fact that the nonuniversal source code encodes $|\mathcal{S}|$ symbol segments that correspond to the states of the context–tree source, while the universal source code encodes up to C symbol segments that correspond to the i.i.d. symbol segments of the BWT output sequence.

context–tree source, overlap. It can not be concluded that the redundancy of the source code is independent of the entropy of the source, as there is insufficient evidence to motivate the conclusion.

A comparison between the curves of the type–one nonuniversal weight–and–index source code and the universal weight–and–index source code reveals that the universal source code is significantly less redundant when both source codes are used to encode shorter source sequences. This observation is motivated by considering the overhead involved in encoding the subsequences of short i.i.d. symbol segments.

The source coding of an i.i.d. symbol segment involves the source coding of several subsequences that correspond to the segment. The weight of each subsequence is encoded in a total of $\lceil \log_2(l_{ss}+1) \rceil$ bits using the conventional binary–coded representation of an integer, where l_{ss} denotes the length of the subsequence. The average redundancy of the codewords that are assigned to the subsequence weights is significant, as the probability distribution of the subsequence weights are not taken into account as they are assigned codewords.

A source code that divides short BWT output sequences into fewer segments may have a smaller normalized average per–codeword redundancy than a source code which divides the BWT output sequences into the actual i.i.d. symbol segments (i.e. a greater number of segments). For short source sequences, the redundancy of the codewords assigned to an additional set of subsequence weights may exceed the additional redundancy of the codewords that are assigned to segments which contain symbols from more than one probability distribution. As the length of the source sequences increases, more symbols from distinct distributions appear in each segment. The additional redundancy of the codewords that are assigned to these segments may in this case exceed the redundancy of the codewords that are assigned to an additional set of subsequence weights. The improved performance of the universal source code, when used to encode shorter source sequences, is therefore due to the segmentation algorithm — it was found that the segmentation algorithm divides certain BWT output sequences into fewer segments than the actual number of i.i.d. symbol segments.

Both curves associated with the universal weight–and–index source code do not (in general) decrease at the same rate w.r.t. the source sequence length as the upper bound on the redundancy of the code. It was found that the accuracy of the segmentation algorithm is extremely sensitive to changes in the block lengths $k_1(n)$ and $k_2(n)$. It is likely that a search for more suitable block lengths would result in a source code with a normalized redundancy that decreases more rapidly w.r.t. the sequence length. It is not clear whether the selection of a single set of block lengths would result in a source code with suitable performance when used to encode sequences from sources with different parameters, however. It appears that this is not the case, as the redundancy of the universal source code changes significantly when used to encode sequences from the second context–tree source, instead of the first context–tree source.

The final observation regarding figure 6.20 concerns the type–two nonuniversal weight–and–index source code for q–ary context–tree sources. The codewords of this source code are less redundant on average than the codewords of the type–one nonuniversal source code if both source codes are used to encode shorter sequences. The normalized redundancy of the type–two nonuniversal source code does not decrease at

as great a rate w.r.t. the source sequence length as the normalized redundancy of the type–one nonuniversal source code, however. This observation implies that the strategy of encoding the segments between the expected positions of the transition points in the BWT output sequence is not optimal in terms of maximizing the rate at which the normalized redundancy of the source code decreases (w.r.t. the source sequence length).

CHAPTER 7

Conclusions

This chapter presents several conclusions regarding enumerative source codes and their use in BWT–based source code implementations. The first two sections of this chapter address the practicality and effectiveness of the enumerative source codes that were investigated in this thesis. The final conclusions regarding enumerative source codes are presented in the third section of this chapter.

7.1 Practicality

The practicality of the enumerative source codes that were investigated in chapter 6 depends on the computational complexity and the memory requirements of their implementations. Both the computational complexity and the memory requirements that are associated with the computation of large binomial coefficients were investigated in chapter 5, as this is the most computationally–intensive task that typical enumerative source code implementations perform. A novel approach to computing the large binomial coefficients that are required by enumerative source code implementations was proposed in section 5.3. This approach involves the decomposition of large numbers into their prime factors.

Several observations regarding the memory requirements of the novel approach to computing large binomial coefficients appear in section 5.3 on page 138. It is concluded from these observations that the memory requirements of enumerative source code implementations for source sequences of up to 10^4 symbols are not excessive, considering the amount of memory that is available on modern personal computers. All enumerative source codes that were investigated in chapter 6 are therefore considered practical in terms of their memory requirements. This conclusion does not imply that the proposed enumerative source code implementations require less memory than other source code implementations, or that enumerative source codes should be favoured above other source codes.

The mean and the standard deviation of the computation time that is required to encode and decode a single source sequence of 10^4 bits using the variable weight, fixed–to–fixed length source code were presented in section 6.1.2.6 on page 165. The encoder and decoder required an average computation time on the order of one or more seconds to encode a random source sequence and decode it from its codeword. The standard deviation of the computation time was negligible. It is concluded that the proposed implementation of the enumerative source code does not compare favourably against implementations of the Lempel–Ziv source code or the `bzip2` implementation (in terms of computation time) — these implementations typically require an equivalent amount of computation time to encode files of several megabytes, instead of 10^4 bits. The

computation time of the enumerative source code implementation is manageable for the purposes of research and simulation, however.

The enumerative source codes that were investigated in this thesis are interesting and useful from a research point of view, even though their implementations are not competitive with those of conventional source codes in terms of their computational complexity. This conclusion may be justified by considering a statement of Caire et. al. [161] regarding optimal fixed–to–fixed length source codes for binary memoryless sources. Caire et. al. stated that optimal fixed–to–fixed length source codes for long source sequences (i.e. 500 bits or longer) from binary memoryless sources are 'non-constructable' if the source codes are to have a reasonable degree of computational complexity. The word 'optimal' refers to the minimum block–error probability within the context of this statement.

It was proved in this thesis that the statement of Caire et. al. [161] is incorrect — the variable weight, fixed–to–fixed length enumerative source code of section 6.1.2 on page 153 was proved as being optimal in theorem 6.1.4 on page 158, and was empirically demonstrated as being optimal (refer to figure 6.5 on page 165). The implementation of this source code may be used to encode and decode source sequences of up to 10^4 bits in seconds, and requires less than 150 kilobytes of memory — this implementation clearly has a reasonable degree of computational complexity.

It is likely that the computational complexity of the enumerative source code implementations that were proposed in this thesis may be reduced. The computational complexity of enumerative source codes was not the topic addressed by this thesis. The original enumerative source codes' high degree of computational complexity was considered an obstacle that needed to be overcome in order to study their effectiveness. This obstacle led to the development of the prime factor decomposition approach to computing large binomial coefficients, which reduces the computational complexity of the original enumerative source codes significantly. A small fraction of the overall time and effort that was spent on this thesis went to reducing the computational complexity of enumerative source code implementations. It is very likely that the efficiency of the proposed enumerative source code implementations may be improved, which implies that future implementations of enumerative source codes may be more competitive with conventional source codes in this regard.

7.2 Effectiveness

The conclusions regarding the effectiveness of the enumerative source codes that were investigated in this thesis are presented in this section.

7.2.1 Single–field enumerative source codes

The conclusions regarding the effectiveness of the single–field enumerative source codes are presented in what follows.

7.2.1.1 The constant weight, fixed–to–fixed length code

The constant weight, fixed–to–fixed length source code was introduced in section 6.1.1 on page 139. It was demonstrated that the block–error probability of this source code did not exceed the block–error probability of several variable–length source codes that were implemented as fixed–to–fixed length source codes, where the source codes were used to encode sequences from the fixed–weight binary source. It is therefore concluded that the constant weight, fixed–to–fixed length source code is more effective in terms of its block–error probability than fixed–to–fixed length implementations of conventional variable–length source codes, if sequences from a fixed–weight binary source are encoded.

A fixed–to–variable length implementation of the constant weight, fixed–to–fixed length source code was proposed in section 6.1.1.5 on page 147. It was demonstrated that a conventional variable–length Huffman code outperforms the fixed–to–variable length implementation of the constant weight, fixed–to–fixed length source code, when compared in terms of their average codeword redundancy (the source codes were used to encode sequences from the fixed–weight binary source). The Huffman code that was compared to the constant weight, fixed–to–fixed length source code was extended to sequences of 500 bits, however — the efficient construction of this extended Huffman code is not transparent (its construction may involve an unmanageable degree of computational complexity). It is concluded that the fixed–to–variable length implementation of the constant weight, fixed–to–fixed length source code is suboptimal in terms of its effectiveness, but that its computational complexity is manageable (unlike the extended Huffman code).

7.2.1.2 The variable weight, fixed–to–fixed length code

The variable weight, fixed–to–fixed length source code was introduced in section 6.1.2 on page 153. It was both proved and demonstrated that the variable weight, fixed–to–fixed length source code is optimal in terms of its block–error probability when used to encode sequences from binary memoryless sources. It was also demonstrated that conventional variable–length source codes, when implemented as fixed–to–fixed length source codes and used to encode source sequences from binary memoryless sources, are suboptimal in terms of their block–error probability. It is therefore concluded that the variable weight, fixed–to–fixed length source code is more effective than fixed–to–fixed length implementations of conventional variable–length source codes, when used to encode sequences from binary memoryless sources.

7.2.2 Multi–field enumerative source codes

The conclusions regarding the effectiveness of multi–field enumerative source codes are presented in what follows.

7.2.2.1 The weight–and–index variable–length code for binary sources

The universal weight–and–index variable–length source code for binary memoryless sources was introduced in section 6.2.1.1 on page 167. It was demonstrated that this

source code is less effective than extended Huffman codes, as well as arithmetic codes, when used to encode sequences from a binary memoryless source with an arbitrary source parameter p. It was also demonstrated that the normalized redundancy of the universal weight–and–index source code for binary memoryless sources decreases at a slower rate w.r.t. the source sequence length than the normalized redundancy of extended Huffman codes and arithmetic codes.

Two important observations regarding the comparison between the universal weight–and–index variable–length source code for binary memoryless sources and the conventional variable–length source codes (i.e. the extended Huffman and arithmetic source codes) were made in section 6.2.1.5 on page 189. The first observation concerns the fact that the weight–and–index source code for binary memoryless sources is a universal[1] source code — its encoder requires no apriori knowledge of the source parameter p in order to encode source sequences and produce codewords with a normalized redundancy that tends to zero asymptotically. The encoders of the extended Huffman codes and the arithmetic codes that were considered in section 6.2.1.5 require apriori knowledge of the source parameter p in order to effectively encode source sequences.

The second observation concerns the computational complexity that is associated with the extended Huffman code. The construction of the extended Huffman code for sequences of 500 bits, as considered in section 6.2.1.5, is computationally infeasible. The computational complexity of the weight–and–index source code is manageable when it is used to encode longer source sequences from binary memoryless sources — this source code is computationally feasible.

Three weight–and–index variable–length source codes for binary context–tree sources were introduced in section 6.2.1.1 on page 168. Two of the three source codes are nonuniversal, while the remaining source code is universal. Each of the proposed weight–and–index source codes for binary context–tree sources transforms source sequences using the BWT, and encodes the transformed sequences using an enumerative source code.

It was demonstrated that the normalized redundancy of the codewords produced by the two nonuniversal weight–and–index source codes decreases w.r.t. the source sequence length if the source codes are used to encode sequences from a context–tree source with known statistics. It was proved analytically that the remaining weight–and–index source code is universal. The proof was carried out under the assumption that the segmentation algorithm of the source code is perfectly accurate, which does not hold true in practice. The segmentation algorithm's degree of accuracy strongly influences the performance of the universal source code. A highly accurate segmentation algorithm relies on the selection of appropriate lengths for its level–one and level–two blocks.

It is concluded that the universal weight–and–index variable–length source code for binary context–tree sources may be used to effectively encode source sequences from binary context–tree sources without apriori knowledge of the source statistics. The practical results presented in section 6.2.1.5 on page 197 support this conclusion. The dependency of the source code's effectiveness on the lengths of the segmentation

[1]It was proved analytically that the weight–and–index source code for binary memoryless sources is universal.

algorithm's blocks detracts from the universality of the code, however.

7.2.2.2 The weight–and–index variable–length code for q–ary sources

The universal weight–and–index variable–length source code for q–ary memoryless sources was introduced in section 6.2.2.1 on page 200. This source code uses the weight–and–index variable–length source code for binary memoryless sources to effectively encode source symbols from q–ary memoryless sources, where $q \geq 2$.

The weight–and–index source code for q–ary memoryless sources was proved analytically as being universal. It was demonstrated in section 6.2.2.5 on page 228 that this source code may be used to encode sequences from q–ary memoryless sources with a normalized redundancy that decreases w.r.t. the source sequence length, and without apriori knowledge of the symbol distribution of the source. It is concluded that the weight–and–index variable–length source code for q–ary memoryless sources is a conceptually simple, universal source code for sequences of q–ary symbols from memoryless sources. The computational complexity of this source code is manageable when used to encode sequences of up to 10^4 symbols.

Three weight–and–index variable–length source codes for q–ary context–tree sources were introduced in section 6.2.2.1 on page 203. Two of the three source codes are nonuniversal, while the remaining source code is universal. Each of the proposed weight–and–index source codes for q–ary context–tree sources transforms source sequences using the BWT, and encodes the transformed sequences using the universal weight–and–index source code for binary memoryless sources.

It was demonstrated that the normalized redundancy of the codewords produced by the two nonuniversal weight–and–index source codes decreases w.r.t. the source sequence length if the source codes are used to encode sequences from two quaternary context–tree sources with known statistics. It was proved analytically that the remaining weight–and–index source code is universal. The proof was carried out under the assumption that the segmentation algorithm of the source code is perfectly accurate, which does not hold true in practice. The segmentation algorithm's degree of accuracy strongly influences the performance of the universal weight–and–index source code for q–ary context–tree sources, as was the case with the universal weight–and–index source code for binary context–tree sources. A highly accurate segmentation algorithm relies on the selection of appropriate lengths for its level–one and level–two blocks.

It is concluded that the universal weight–and–index variable–length source code for q–ary context–tree sources may be used to effectively encode source sequences from q–ary context–tree sources without apriori knowledge of the source statistics. The practical results presented in section 6.2.2.5 on page 229 support this conclusion. The dependency of the source code's effectiveness on the lengths of the segmentation algorithm's blocks detracts from the universality of the code, however.

7.3 Final conclusions

Enumerative source code implementations with manageable computational complexity may be realized. These implementations are not, at present, competitive with con-

ventional source code implementations in terms of their efficiency. The computational complexity of the enumerative source code implementations presented in this thesis is manageable for the purposes of research and simulation, however. It likely that further research would lead to additional reductions in the computational complexity of these implementations.

The fixed–to–fixed length enumerative source code proposed in this thesis was proved as being optimal in terms of its block–error probability when used to encode sequences of up to 10^4 bits from binary memoryless sources. Fixed–to–fixed length implementations of conventional variable–length source codes were demonstrated as being suboptimal when used to encode the same sequences as the proposed source code. It is concluded that enumerative source codes may be used to realize optimal fixed–to–fixed length source code implementations with manageable computational complexity.

Enumerative source codes may be used to universally encode sequences from q–ary memoryless sources and q–ary context–tree sources, where $q \geq 2$. The universal source coding of sequences from context–tree sources is achieved by combining enumerative source codes, the Burrows–Wheeler transform, and a segmentation algorithm. The performance of the universal source code for context–tree sources depends on the lengths that are selected for the level–one and level–two blocks of the segmentation algorithm.

REFERENCES

[1] International FFederation of the Phonographic Industry. (2007) Recorded music sales. [Online]. Available: http://www.ifpi.com/content/library/Recorded-music-sales-2007.pdf

[2] Cisco Systems Inc. (2008) Cisco annual report. [Online]. Available: http://www.cisco.com/web/about/ac49/ac20/ac19/ar2008/index.html

[3] The GSM association. (2008) The GSM world webpage. [Online]. Available: http://www.gsmworld.com/

[4] K. Sayood, *Introduction to Data Compression*, 3rd ed. San Francisco, CA: Morgan Kaufmann, 2005.

[5] T. J. Lynch, "Sequence time coding for data compression," *Proc. IEEE*, vol. 54, pp. 1490–1491, Oct. 1966.

[6] L. D. Davisson, "Comments on 'Sequence time coding for data compression'," *Proc. IEEE*, vol. 54, p. 2010, Dec. 1966.

[7] T. Cover, "Enumerative source coding," *IEEE Trans. Inf. Theory*, vol. 19, pp. 73–77, Jan. 1973.

[8] J. G. Cleary and I. H. Witten, "A comparison of enumerative and adaptive codes," *IEEE Trans. Inf. Theory*, vol. 30, pp. 306–315, Mar. 1984.

[9] M. Burrows and D. J. Wheeler, "A block–sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, CA, Research Report SRC–124, May 1994.

[10] M. Effros, K. Visweswariah, S. R. Kulkarni, and S. Verdu, "Universal lossless source coding with the Burrows Wheeler transform," *IEEE Trans. Inf. Theory*, vol. 48, pp. 1061–1081, May 2002.

[11] S. Verdu, "Fifty years of Shannon theory," *IEEE Trans. Inf. Theory*, vol. 44, pp. 2057–2078, Oct. 1998.

[12] C. E. Shannon, "A mathematical theory of communication," *Bell Systems technical journal*, vol. 27, pp. 379–423, 623–656, July–Oct. 1948.

[13] B. McMillan, "The basic theorems of information theory," *Ann. Math. Statist.*, vol. 24, pp. 196–219, Jun. 1953.

[14] A. I. Khinchin, "The entropy concept in probability theory," *Usp. Mat. Nauk.*, vol. 8, pp. 3–20, 1953, English translation in Mathematical Foundations of Information Theory. New York: Dover, 1957.

[15] R. M. Fano, "The transmission of information," Research laboratory of electronics, MIT, Cambridge, MA, Tech. Rep. 65, 1944.

[16] K. Sayood, Ed., *Lossless compression handbook*, 1st ed. San Diego, CA: Academic press, 2003.

[17] D. Huffman, "A method for the construction of minimum redundancy codes," in *Proceedings of the IRE*, vol. 48, Sep. 1952, pp. 1098–1101.

[18] R. G. Gallager, "Variations on a theme by Huffman," *IEEE Trans. Inf. Theory*, vol. 24, pp. 668–674, Nov. 1978.

[19] (2007) The International Telecommunications Union website. [Online]. Available: http://www.itu.org/

[20] *JPEG still image data compression standard*, ISO Std. DIS 10 918-1/2.

[21] A. S. Fraenkel and S. T. Klein, "Bounding the depth of search trees," *The computer journal*, vol. 36, pp. 668–678, 1978.

[22] R. L. Milidiu, A. A. Pessoa, and E. S. Laber, "In–place length–restricted prefix coding," in *Proc. String processing and information retrieval*, Sep. 1998, pp. 50–59.

[23] A. Moffat and J. Katajainen, "In–place calculation of minimum redundancy codes," in *Proc. of the workshop on algorithms and data structures*, Aug. 1995, pp. 303–402.

[24] L. L. Larmore and D. S. Hirschberg, "A fast algorithm for optimal length–limited Huffman codes," *Journal of the ACM*, vol. 37, pp. 464–473, Jul. 1990.

[25] N. Faller, "An adaptive system for data compression," in *Proc. 7th Asilomar Conference on Circuits, Systems and Computers*, 1973, pp. 393–397.

[26] D. R. McIntyre and M. A. Pechura, "Data compression using static Huffman code–decode tables," *Communications of the ACM*, vol. 28, pp. 612–616, Jun. 1985.

[27] A. Moffat and A. Turpin, "On the implementation of minimum redudancy prefix codes," *IEEE Trans. Commun.*, vol. 45, pp. 1200–1207, Oct. 1997.

[28] D. S. Hirschberg and D. A. Lelewer, "Efficient decoding of prefix codes," *Communications of the ACM*, vol. 33, pp. 449–459, Apr. 1990.

[29] Y. Choueka, S. T. Klein, and Y. Perl, "Efficient variants of Huffman codes in high–level languages," in *Proc. of the 8th ACM–SIGIR Conference*, 1985, pp. 122–130.

[30] A. Sieminski, "Fast decoding of the Huffman codes," *Information processing letters*, vol. 26, pp. 237–241, Jan. 1988.

[31] H. Tanaka, "Data structure of Huffman codes and its application to efficient coding and decoding," *IEEE Trans. Inf. Theory*, vol. 33, pp. 154–156, Jan. 1987.

[32] M. A. Bassiouni and A. Mukherjee, "Efficient decoding of compressed data," *Journal of the American society of Information science*, vol. 46, pp. 1–8, Jan. 1995.

[33] L. Kraft, "A device for quantizing, grouping and coding amplitude modulated pulses," Master's thesis, Dept. Elec. Eng., MIT, Cambridge, MA, 1949.

[34] B. McMillan, "Two inequalities implied by unique decipherability," *IRE trans. inform. theory*, vol. 2, pp. 115–116, Dec. 1956.

[35] N. Abramson, *Information theory and coding*. New York: McGraw–Hill, 1963.

[36] F. Jelinek, *Probabilistic information theory*. New York: McGraw–Hill, 1968.

[37] J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM journal of research and development*, vol. 20, pp. 198–203, May 1976.

[38] R. Pasco, "Source coding algorithms for fast data compression," Ph.D. dissertation, Stanford University, Stanford, CA, 1976.

[39] J. Rissanen and G. Langdon, Jr., "Arithmetic coding," *IBM journal of research and development*, vol. 23, pp. 149–162, Mar. 1979.

[40] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, pp. 520–540, Jun. 1987.

[41] (2007) A non-commercial resource regarding djvu. [Online]. Available: http://www.djvu.org/

[42] (2007) A non-commercial resource regarding bzip. [Online]. Available: http://www.bzip.org/

[43] R. Aravind, G. Cash, D. Duttweiler, H. Hang, B. Haskell, and A. Puri, "Image and video coding standards," *ATT Tech. J.*, vol. 72, pp. 67–89, Jan.–Feb. 1993.

[44] D. J. C. MacKay, *Information theory, inference, and learning algorithms*. Cambridge, UK: Cambridge University Press, 2003.

[45] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun.*, vol. 32, pp. 396–402, Apr. 1984.

[46] J. Rissanen and K. M. Mohiuddin, "A multiplication–free, multialphabet arithmetic code," *IEEE Trans. Commun.*, vol. 37, pp. 93–98, Feb. 1989.

[47] M. Effros, "PPM performance with BWT complexity: A fast and effective data compression algorithm," *Proc. IEEE*, vol. 88, pp. 1703–1712, Nov. 2000.

[48] A. I. Wirth, "Symbol–driven compression of Burrows Wheeler transformed text," Master's thesis, The University of Melbourne, Melbourne, Australia, 2001.

[49] I. H. Witten and T. C. Bell, "The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression," *IEEE Trans. Inf. Theory*, vol. 37, pp. 1085–1094, Jul. 1991.

[50] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Trans. Commun.*, vol. 38, pp. 1917–1921, Nov. 1990.

[51] J. G. Cleary and W. J. Teahan, "Unbounded length contexts for PPM," *The computer journal*, vol. 40, pp. 67–75, 1997.

[52] S. Bunton, "On–line stochastic processes in data compression," Ph.D. dissertation, The University of Washington, Seattle, Washington, 1996.

[53] ——, "Semantically motivated improvements for PPM variants," *The computer journal*, vol. 40, pp. 76–93, 1997.

[54] E. M. McCreight, "A space–economical suffix tree construction algorithm," *Journal of the ACM*, vol. 23(12), pp. 262–272, Apr. 1976.

[55] E. Ukkonen, "On–line construction of suffix trees," *Algorithmica*, vol. 14, pp. 249–260, Sep. 1995.

[56] N. J. Larsson, "Structures of string matching and data compression," Ph.D. dissertation, Lund University, Sweden, 1999.

[57] A. N. Kolmogorov, "Three approaches to the quantitive definition of information," *Prob. Inf. Trans.*, vol. 1, pp. 1–7, 1965.

[58] (2007) A non-commercial resource regarding gzip. [Online]. Available: http://www.gzip.org/

[59] L. Davisson, "Universal noiseless coding," *IEEE Trans. Inf. Theory*, vol. 19, pp. 783–795, Nov. 1973.

[60] J. P. M. Schalkwijk, "An algorithm for source coding," *IEEE Trans. Inf. Theory*, vol. 18, pp. 395–399, May 1972.

[61] Y. M. Shtarkov and V. F. Babkin, "Combinatorial method of universal coding for discrete stationary sources," in *Proc. Second International Symposium on Information Theory (Tsahkadsor, Armenian S.S.R, 1971)*, Budapest, Hungary: Publishing House of the Hungarian Acad. Sci., 1973, pp. 249–256.

[62] J. Ziv, "Coding of sources with unknown statistics — part I: Probability of encoding error," *IEEE Trans. Inf. Theory*, vol. 18, pp. 384–389, May 1972.

[63] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, pp. 337–343, May 1977.

[64] ——, "Compression of individual sequences via variable–rate coding," *IEEE Trans. Inf. Theory*, vol. 24, pp. 530–536, Sep. 1978.

[65] (2009) The Winzip homepage. [Online]. Available: http://www.winzip.com/

[66] M. Rodeh, V. R. Pratt, and S. Even, "Linear algorithm for data compression via string matching," *Journal of the ACM*, vol. 28, pp. 16–24, Jan. 1981.

[67] G. Louchard and W. Szpankowski, "Average profile and limiting distribution for a phrase size in the Lempel–Ziv parsing algorithm," *IEEE Trans. Inf. Theory*, vol. 41, pp. 478–488, Mar. 1995.

[68] *Data compression procedures for data circuit terminating equipment (DCE) using error correcting procedures*, ITU-T Std. V.42 bis, 1990.

[69] *Data compression procedures*, ITU-T Std. V.44, 2000.

[70] (2009) Wikipedia web page for the compress utility. [Online]. Available: http://en.wikipedia.org/wiki/Compress

[71] (2009) Wikipedia web page for the gif image file format. [Online]. Available: http://en.wikipedia.org/wiki/Gif

[72] K. Sadakane, "A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation," in *Proc. 1998 Data Compression Conf.*, Mar.–Apr. 1998, pp. 129–138.

[73] C. A. R. Hoare, "Quicksort," *The computer journal*, vol. 5, pp. 10–16, 1962.

[74] J. L. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings," in *Proc. 8th ann. ACM–SIAM Symp. on Discrete Alg.*, Jan. 1997, pp. 360–369.

[75] A. Andersson and S. Nilsson, "A new efficient radix sort," in *Proc. 35th ann. IEEE Symp. on Foundations of Computer Science*, Nov. 1994, pp. 714–721.

[76] D. Baron and Y. Bresler, "Antisequential suffix sorting for Bwt–based data compression," *IEEE Trans. Comput.*, vol. 54, pp. 385–397, Apr. 2005.

[77] R. M. Karp, R. E. Miller, and A. L. Rosenberg, "Rapid identifcation of repeated patterns in strings, trees and arrays," in *Proc. 4th ann. ACM Symp. on Theory of Computing*, May 1972, pp. 125–136.

[78] U. Manber and G. Myers, "Suffix arrays: A new method for on–line string searches," in *Proc. 1st ann. ACM–SIAM Symp. on Discrete Algorithms*, Jan. 1990, pp. 319–327.

[79] M. Weinberger, J. Rissanen, and M. Feder, "A universal finite memory source," *IEEE Trans. Inf. Theory*, vol. 41, pp. 643–652, May 1995.

[80] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens, "The context–tree weighting method: Basic properties," *IEEE Trans. Inf. Theory*, vol. 41, pp. 653–664, May 1995.

[81] A. Martin, G. Seroussi, and M. Weinberger, "Enumerative coding for tree sources," in *Proc. 2008 IEEE int. Symposium on Information Theory*, Jul. 2008, pp. 2459–2463.

[82] M. Effros, "Universal lossless source coding with the Burrows Wheeler transform," in *Proc. 1999 IEEE Data Compression Conference*, Mar. 1999, pp. 178–187.

[83] K. Viswesvariah, S. Kulkarni, and S. Verdu, "Output distribution of the Burrows–Wheeler transform," in *Proc. 2000 IEEE int. Symposium on Information Theory*, Jun. 2000, p. 53.

[84] P. Elias, "Interval and recency rank source coding: two on–line adaptive variable–length schemes," *IEEE Trans. Inf. Theory*, vol. 33, pp. 3–10, Jan. 1987.

[85] G. Caire, S. Shamai, and S. Verdu, "Noiseless data compression with low density parity check codes," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society*, P. Gupta and G. Kramer, Eds., 2004.

[86] J. Abel, "Improvements to the Burrows–Wheeler compression algorithm: After BWT stages," University of Duisburg–Essen, Duisburg, Germany, Tech. Rep., 2003.

[87] (2009) Wikipedia webpage regarding bzip. [Online]. Available: http://en.wikipedia.org/wiki/Bzip2

[88] P. M. Fenwick, "Improvements to the block sorting text compression algorithm," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep. 120, Aug. 1995.

[89] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, pp. 520–540, Jun. 1987.

[90] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM transactions on information systems*, vol. 16, pp. 256–294, Jul. 1998.

[91] P. M. Fenwick, "Experiments with a block sorting text compression algorithm," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep. 111, May 1995.

[92] ——, "The Burrows–Wheeler transform for block sorting text compression — principles and improvements," *The computer journal*, vol. 39, pp. 731–740.

[93] ——, "Block sorting text compression — final report," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep. 130, Apr. 1996.

[94] B. Balkenhol, S. Kurtz, and Y. M. Shtarkov, "Modifications of the Burrows and Wheeler data compression algorithm," in *Proc. 1999 IEEE Data compression conference*, Mar. 1999, pp. 188–197.

[95] B. Balkenhol and Y. M. Shtarkov, "One attempt of a compression algorithm using the BWT," Faculty of Mathematics, University of Bielefeld, Germany, Tech. Rep. 99-133 (SFB343), 1999.

[96] B. Chapin and S. R. Tate, "Higher compression from the Burrows–Wheeler transform by modified sorting," in *Proc. 1998 IEEE Data compression conference*, Mar.–Apr. 1998, p. 532.

[97] H. Kruse and A. Mukherjee, "Improving text compression ratios with the Burrows-Wheeler transform," in *Proc. 1999 IEEE Data compression conference (DCC99)*, Mar. 1999, p. 536.

[98] S. Deorowicz, "Universal lossless data compression algorithms," Ph.D. dissertation, Silesian University of Technology, Gliwice, Poland, 2003. [Online]. Available: http://sun.aei.polsl.pl/ sdeor

[99] S. Grabowski, "Text preprocessing for Burrows-Wheeler block–sorting compression," in *Proc. VII Konferencja Sieci i Systemy Informatyczne – Teoria, Projekty, Wdrozenia*, Lodz, Poland, Oct. 1999, pp. 229–239.

[100] T. Bell, I. H. Witten, and J. G. Cleary, "Modelling for text compression," *ACM computing surveys*, vol. 21, pp. 557–591, Dec. 1989.

[101] F. S. Awan, N. Zhang, N. Motgi, R. T. Iqbal, and A. Mukherjee, "LIPT: A reversible lossless text transform to improve compression performance," in *Proc. 2001 IEEE Data compression conference*, Mar. 2001, p. 481.

[102] F. S. Awan and A. Mukherjee, "LIPT: A lossless text transform to improve compression," in *Proc. 2001 Int. conference on information technology: Coding and computing*, Apr. 2001, pp. 452–460.

[103] L. Butterman and N. Memon, "An error–resilient blocksorting compression algorithm," in *Proc. 2003 IEEE Data compression conference*, Mar. 2003, p. 417.

[104] H. A. Elsayed and M. Alghoniemy, "Lossless audio coding using Burrows–Wheeler transform and move–to–front coding," in *Proc. 2007 int. Conference on Computer Engineering and Systems*, Nov. 2007, pp. 209–212.

[105] D. Adjeroh, Y. Zhang, A. Mukherjee, M. Powell, and T. Bell, "DNA sequence compression using the Burrows–Wheeler transform," in *Proc. 2002 IEEE Computer Society Bioinformatics Conference*, 2002, pp. 303–313.

[106] R. Y. K. Isal and A. Moffat, "Word–based block–sorting text compression," in *Proc. 24th Australasian Conference on Computer Science*, Jan.–Feb. 2001, pp. 92–99.

[107] J. Lansky, K. Chernik, and Z. Vlickova, "Comparison of text models for BWT," in *Proc. 2007 IEEE Data compression conference*, Mar. 2007, p. 389.

[108] S. Mantaci, A. Restivo, and M. Sciortino, "An extension of the Burrows Wheeler transform to k words," in *Proc. 2005 IEEE Data compression conference*, Mar. 2005, p. 469.

[109] Z. Arnavut and S. S. Magliveras, "Lexical permutation sorting algorithm," *The computer journal*, vol. 40, pp. 292–295, 1997.

[110] Z. Arnavut, "Generalization of the BWT transformation and inversion ranks," in *Proc. 2002 IEEE Data compression conference*, Apr. 2002, p. 447.

[111] C. E. Shannon, "Prediction and entropy of printed English," in *Bell Systems Technical Journal*, no. 30, pp. 50–64.

[112] (2008) The Canterbury corpus. [Online]. Available: http://corpus.canterbury.ac.nz/

[113] Z. Arnavut, D. Leavitt, and M. Abdulazizoglu, "Block sorting transformations," in *Proc. 1998 IEEE Data compression conference*, Mar.–Apr. 1998, p. 524.

[114] Z. Arnavut and E. Plaku, "Lossless compression of ECG signals," in *Proc. of the first joint BMES/EMBS conference*, Oct. 1999, p. 274.

[115] J. McCabe, "On serial files with relocatable records," *Operations research*, vol. 13, pp. 609–618, Jul.–Aug. 1965.

[116] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, pp. 320–330, Apr. 1986.

[117] M. Schindler, "A fast block–sorting algorithm for lossless data compression," in *Proc. 1997 IEEE Data compression conference*, Mar. 1997, p. 469.

[118] R. Dorrigiv, A. Lopez-Ortiz, and J. I. Munro, "Experimental evaluation of list update algorithms for data compression," School of Computer Science, University of Waterloo, Ontario, Canada, Tech. Rep. CS–2007–38, 2007.

[119] R. Bachrach, R. El-Yaniv, and M. Reinstaedtler, "On the competitive theory and practice of online list accessing algorithms," *Algorithmica*, vol. 32, pp. 201–246, 2002.

[120] R. Bachrach and R. El-Yaniv, "Online list accessing algorithms and their applications: Recent empirical evidence," in *Proc. 8th Annual ACM–SIAM Symposium on discrete algorithms,*, Jan. 1997, pp. 53–62.

[121] B. Chapin, "Switching between two on–line list update algorithms for higher compression of Burrows–Wheeler transformed data," in *Proc. 2000 IEEE Data compression conference*, Mar. 2000, pp. 183–192.

[122] S. Albers, "Improved randomized on–line algorithms for the list update problem," *SIAM journal on computing*, vol. 27, pp. 682–693, Jun. 1998.

[123] S. Albers and M. Mitzenmacher, "Average case analyses of list update algorithms, with applications to data compression," *Algorithmica*, vol. 21, pp. 312–329, 1998.

[124] F. Schulz, "Two new families of list update algorithms," in *Proc. 9th Int. symposium on algorithms and computation*, Dec. 1998, pp. 99–108.

[125] R. Dorrigiv, A. Lopez-Ortiz, and J. I. Munro, "List update algorithms for data compression," in *Proc. 2008 IEEE Data compression conference*, Mar. 2008, p. 512.

[126] S. Deorowicz, "Second step algorithms in the Burrows–Wheeler compression algorithm," Silesian University of Technology, Gliwice, Poland, Tech. Rep. 1–13, 2002.

[127] Z. Arnavut and S. S. Magliveras, "Block sorting and compression," in *Proc. 1997 Data compression conference*, Mar. 1997, pp. 181–190.

[128] A. Kadach, "Effective algorithms for lossless compression of textual data," Ph.D. dissertation, Russian science society, Novosibirsk, Russia, 1997.

[129] E. Binder. (2000) Distance coder – usenet group comp.compression. [Online]. Available: http://groups.google.com/group/comp.compression/msg/27d46abca0799d12

[130] P. A. J. Volf and F. M. J. Willems, "Switching between two universal source coding algorithms," in *Proc. 1998 IEEE Data compression conference*, 1998, pp. 491–500.

[131] (2008) A resource regarding bred3. [Online]. Available: ftp://ftp.cl.cam.ac.uk/users/djw3/

[132] G. K. Zipf, *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology.*, 1st ed. Cambridge, MA: Addison–Wesley, 1949.

[133] B. Balkenhol and S. Kurtz, "Universal data compression based on the Burrows and Wheeler transformation: theory and practice," Faculty of Mathematics, The University of Bielefeld, Bielefeld, Germany, Technical Report 98–069, Sonderforschungsbereich: Diskrete Strukturen in der Mathematik, 1998.

[134] R. Krichevsky and V. Trofimov, "The performance of universal encoding," *IEEE Trans. Inf. Theory*, vol. 27, pp. 199–207, Mar. 1981.

[135] J. Rissanen, "Complexity of strings in the class of Markov sources," *IEEE Trans. Inf. Theory*, vol. 32, pp. 526–532, Jul. 1986.

[136] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Trans. Inf. Theory*, vol. 21, pp. 194–203, Mar. 1975.

[137] M. A. Maniscalco. (2000) A run length encoding scheme for block sort transformed data. [Online]. Available: http://www.geocities.com/m99datacompression/papers/rle/rle.html

[138] ——. (2001) A second modified run length encoding scheme for block sort transformed data. [Online]. Available: http://www.geocities.com/m99datacompression/papers/rle2.html

[139] N. J. Larsson, "The context trees of block sorting compression," in *Proc. 1998 IEEE Data compression conference*, Mar.–Qpr. 1998, pp. 189–198.

[140] H. Yokoo, "Data compression using a sort–based context similarity measure," *The computer journal*, vol. 40, pp. 94–102, 1997.

[141] Z. Arnavut, "Move–to–front and inversion coding," in *Proc. 2000 IEEE Data compression conference*, Mar. 2000, pp. 193–202.

[142] R. Sedgewick, "Permutation generation methods," *ACM Computing Surveys*, vol. 9, pp. 137–164, Jun. 1977.

[143] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, "Fast compression with a static model in high–order entropy," in *Proc. 2004 IEEE Data compression conference*, Mar. 2004, pp. 62–71.

[144] R. Grossi, A. Gupta, and J. S. Vitter, "High–order entropy–compressed text indexes," in *Proc. 14th ann. ACM–SIAM symp. on discrete algorithms*, Jan. 2003, pp. 841–850.

[145] L. Oktem and J. Astola, "Hierarchical enumerative coding of locally stationary binary data," *IET Electronics Letters*, vol. 35, pp. 1428–1429, Aug. 1999.

[146] ——, "Hierarchical enumerative coding of first–order Markovian binary sources," *IET Electronics Letters*, vol. 35, pp. 2003–2005, Nov. 1999.

[147] W. van der Walt and H. C. Ferreira, "Enumerative Huffman coding," in *Proc. 1997 IEEE int. symp. on information theory*, Jun.–Jul. 1997, p. 143.

[148] K. A. S. Immink and A. J. E. M. Janssen, "Error propagation assessment of enumerative coding schemes," *IEEE Trans. Inf. Theory*, vol. 45, pp. 2591–2594, Nov. 1999.

[149] V. Dai and A. Zakhor, "Binary combinatorial coding," in *Proc. 2003 IEEE Data compression conference*, Mar. 2003, p. 420.

[150] H. Tanaka, M. Ando, and A. Leon-Garcia, "A unique ranking of multilevel sequences and its application to source coding," *IEEE Trans. Inf. Theory*, vol. 31, pp. 530–537, Jul. 1985.

[151] D. Hertz and Y. Azenkot, "On a class of multilevel universal source coding schemes," *IEEE Trans. Inf. Theory*, vol. 36, pp. 1442–1446, Nov. 1990.

[152] B. Ryabko, "Fast enumerative source coding," in *Proc. 1995 IEEE Int. symp. on information theory*, Sep. 1995, p. 395.

[153] G. A. Gordon, "An algorithmic code for monotonic sources," NASA, Greenbelt, Md., Final report of GSFC Summer Workshop, GSFC X–100–65–407, 1965.

[154] F. Jelinek, "Buffer overflow in variable length coding of fixed rate sources," *IEEE Trans. Inf. Theory*, vol. 14, pp. 490–501, May 1968.

[155] F. M. J. Willems, private communication, 2008.

[156] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*. Addison–Wesley, 1976.

[157] J. Rissanen and G. Langdon, Jr., "Universal modeling and coding," *IEEE Trans. Inf. Theory*, vol. 27, pp. 12–23, Jan. 1981.

[158] J. Rissanen, "Arithmetic codings as number representations," in *Acta Polytech, Scandanavica*, vol. Math 31, 1979, pp. 44–51.

[159] G. I. Shamir and D. J. Costello, Jr., "Asymptotically optimal low–complexity sequential lossless coding for piecewise–stationary memoryless sources — part I: The regular case," *IEEE Trans. Inf. Theory*, vol. 46, pp. 2444–2467, Nov. 2000.

[160] J. G. Proakis, *Digital communications*, 4th ed. New York, NY: McGraw-Hill, 2001.

[161] G. Caire, S. Shamai, and S. Verdu, "A new data compression algorithm for sources with memory based on error correcting codes," in *Proc. IEEE Information Theory Workshop (ITW'03)*, Apr. 2003, pp. 291–295.

ADDENDUM A

The first enumerative source code

Lynch [5] proposed the first enumerative source code, but did not derive any bounds on its effectiveness. Davisson [6] subsequently proposed a source code implementation for removing redundant data samples from a sample sequence. This implementation is similar to the first enumerative source code. Davisson investigated the performance of the source code implementation, and provided an analytical definition of its source encoder and decoder.

Addendum A is divided into two sections. The first section is a description of the enumerative source code implementation that was proposed by Davisson [6]. The second section contains an expression for the asymptotic code rate of the source code implementation, as well as its derivation.

A.1 Description

Davisson [6] proposed a blockwise enumerative source code implementation. The implementation encodes each sequence of n source samples independently from other sequences, thereby producing a variable–length codeword for each source sequence. The structure of the variable–length codeword that is assigned to a sequence of n samples is presented in figure A.1 on page 256. The variable–length codeword consists of three fields.

The source encoder encodes each sequence of n source samples by first determining which of its samples are to be removed (the redundant samples) and which are to be retained (the nonredundant samples). The number of nonredundant source samples may differ from source sequence to source sequence — in order to successfully decode each codeword, the source decoder requires knowledge of the exact number of nonredundant source samples within each source sequence. The first field of each codeword represents the number of nonredundant samples within the source sequence associated with the codeword.

Consider a source sequence $\mathbf{x}^n$ with m nonredundant samples and $n-m$ redundant samples. The first field of the codeword assigned to the source sequence is the conventional binary–coded representation of the integer m. As the integer m may assume one of $n+1$ distinct values (any integer from zero to n), the first field consists of

$$l_1(\mathbf{x}) = \lceil \log_2(n+1) \rceil \tag{A.1}$$

bits.

The indices of the m nonredundant samples within the source sequence may be expressed as a monotonically increasing integer sequence. The source encoder encodes

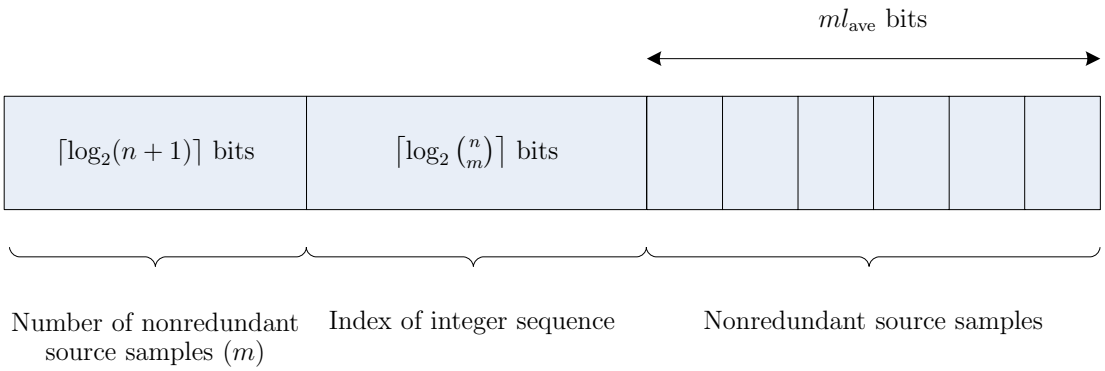


Figure A.1: The codeword assigned to a source sequence of n samples, m of which are nonredundant. A description of each codeword field, as well as the length of each field, are provided in the figure.

this sequence using the first enumerative source code (i.e. the source code proposed by Lynch [5] — refer to section 4.1.1 on page 101). The codeword of the first enumerative source code represents the index of the monotonically increasing integer sequence in the ordered set of all monotonically increasing integer sequences of the same length (and with integers limited to the set $\{1, 2, \ldots n\}$). The codeword of the first enumerative source code is the second field of the codeword assigned to the sequence $\mathbf{x}^n$ (refer to figure A.1). The second field has a length of

$$l_2(\mathbf{x}) = \left\lceil \log_2 \binom{n}{m} \right\rceil \tag{A.2}$$

bits, as a total of $\binom{n}{m}$ distinct, monotonically increasing integer sequences of length m exist (each integer assumes a distinct value from the set $\{1, 2, \ldots n\}$).

The final field of the codeword represents the m nonredundant samples of the source sequence associated with the codeword. The implementation of Davisson [6] accommodates the case where the number of bits required to represent each nonredundant sample varies from sample to sample. In order to derive an expression for the asymptotic code rate of the source code implementation, it is sufficient to consider only the average number of bits that are required to represent a nonredundant sample. Let l_{ave} denote the average number of bits that are required to represent a nonredundant sample. The final field of the codeword has a length of

$$l_3(\mathbf{x}) = ml_{\text{ave}} \tag{A.3}$$

bits.

The source decoder is aware of the source sequence length n. It is able to calculate the length of the first field of each codeword, and decode the field to obtain the value of m (i.e. the number of nonredundant samples in the source sequence associated with the codeword). It subsequently calculates the length of the second field of the codeword, and decodes the second field using the first enumerative source code. It finally reconstructs the source sequence using the recovered integer sequence and the nonredundant samples of the third codeword field.

A.2 Performance

An expression for the average number of codeword bits that the source code implementation of Davisson [6] assigns to each sample from a particular source is derived in this section. A source that produces statistically independent samples is considered. The expression holds for source sequences with lengths that tend to infinity.

Consider a source that produces nonredundant samples with a constant probability of p, and redundant samples with a probability of $1-p$, where $0 < p < 1$. The three fields of the codeword assigned to a source sequence $\mathbf{x}^n$ with m nonredundant samples require a total of

$$l(\mathbf{x}) = \lceil \log_2(n+1) \rceil + \left\lceil \log_2 \binom{n}{m} \right\rceil + m l_{\text{ave}} \tag{A.4}$$

bits to represent (refer to figure A.1). The binomial coefficient $\binom{n}{m}$ may be approximated as

$$\binom{n}{m} \approx \sqrt{\frac{n}{(n-m)m2\pi}} \left(\frac{n}{n-m}\right)^n \left(\frac{n-m}{m}\right)^m \tag{A.5}$$

for large values of n. The average number of codeword bits assigned to each source sample, as the source sequence length n tends to infinity, is approximated as

$$\begin{aligned} l_a(\mathbf{x}) &= \lim_{n\to\infty} \frac{l(\mathbf{x})}{n} \\ &= \lim_{n\to\infty} \frac{\log_2(n+1)}{n} + \lim_{n\to\infty} \frac{m l_{\text{ave}}}{n} \\ &\quad \lim_{n\to\infty} \frac{1}{n} \left[\log_2 \left(\sqrt{\frac{n}{(n-m)m2\pi}} \right) + n \log_2 \left(\frac{n}{n-m} \right) + m \log_2 \left(\frac{n-m}{m} \right) \right]. \end{aligned} \tag{A.6}$$

The weak law of large numbers is used to simplify the expression for the average number of codeword bits assigned to each source sample. According to the weak law of large numbers, a positive integer n_0 and positive real numbers δ and ϵ (where $0 < \delta, \epsilon < 1$) exist such that

$$\Pr[|(m/n - p)| < \epsilon] > 1 - \delta \tag{A.7}$$

for any $n > n_0$. It follows that the expression

$$\lim_{n\to\infty} \frac{m}{n} = p \tag{A.8}$$

holds.

Equation A.6 is simplified in what follows. The first term within the square brackets on the right–hand side of equation A.6 is simplified as

$$\begin{aligned} &\lim_{n\to\infty} \frac{1}{n} \log_2 \left(\sqrt{\frac{n}{(n-m)m2\pi}} \right) \\ &= \lim_{n\to\infty} \frac{1}{2n} \log_2 \left(\frac{1}{n(1-p)p2\pi} \right) \\ &= \lim_{n\to\infty} \frac{1}{2n} \log_2 \left(\frac{1}{n} \right) + \lim_{n\to\infty} \frac{1}{2n} \log_2 \left(\frac{1}{(1-p)p2\pi} \right). \end{aligned} \tag{A.9}$$

The second term on the right–hand side of equation A.9 equals zero. The first term on the right–hand side of equation A.9 is simplified using l'Hopital's rule as

$$\begin{aligned}\lim_{n\to\infty}\frac{1}{2n}\log_2\left(\frac{1}{n}\right) &= -\frac{1}{2}\lim_{n\to\infty}\frac{\log_2(n)}{n}\\ &= -\frac{1}{2}\lim_{n\to\infty}\frac{1}{n\ln(2)}\\ &= 0.\end{aligned} \tag{A.10}$$

Equation A.6 may therefore be simplified as

$$\begin{aligned}l_a(\mathbf{x}) &= \lim_{n\to\infty}\frac{ml_{\text{ave}}}{n}+\lim_{n\to\infty}\frac{1}{n}\left[n\log_2\left(\frac{n}{n-m}\right)+m\log_2\left(\frac{n-m}{m}\right)\right]\\ &= pl_{\text{ave}}-\log_2(1-p)+p\log_2\left(\frac{1-p}{p}\right)\\ &= pl_{\text{ave}}-p\log_2(p)-(1-p)\log_2(1-p)\\ &= pl_{\text{ave}}+h(p).\end{aligned} \tag{A.11}$$

The minimum number of codeword bits that have to be assigned (on average) to each source sample in order to guarantee a uniquely decodable code is subsequently derived. Each source sequence may be interpreted as a sequence of n statistically independent and identically distributed bits. Each bit of this sequence equals zero if the corresponding sample is redundant, and one if the sample is nonredundant. To uniquely represent each source bit, a minimum of

$$h(p) = -p\log_2(p)-(1-p)\log_2(1-p) \tag{A.12}$$

codeword bits have to be assigned to each source bit on average.

The nonredundant samples of each source sequence are encoded in addition to its bit sequence. An average of l_{ave} bits are required to represent each nonredundant sample; the m nonredundant samples of a source sequence are therefore encoded in an average of ml_{ave} bits. An average of ml_{ave}/n bits per source sample are required to represent the nonredundant source samples of the source sequence. The optimal asymptotic code rate is therefore derived as

$$l_{a,\text{opt}}(\mathbf{x}) = pl_{\text{ave}}+h(p) \tag{A.13}$$

bits per source sample. This expression is identical to the expression for the asymptotic code rate of the source code implementation proposed by Davisson (refer to equation A.11). The source code implementation is therefore optimal in terms of its asymptotic code rate.

ADDENDUM B

The enumerative source code proposed by Cover

This addendum is a summary of the enumerative source code for bit sequences from stationary first–order Markov sources, as proposed by Cover [7]. The first part of the summary is a description of the source code, and the second part contains the derivation of a bound on the performance of the source code.

B.1 Description

Cover [7] considered the source coding of bit sequences from stationary first–order Markov sources. The state–transition diagram of a first–order binary Markov source is presented in figure B.1 on page 260.

Consider an n–bit source sequence $\mathbf{x}^n$ that was produced by the Markov source of figure B.1. Let the number of occurrences of the two–bit pair ab (where $a, b \in \{0, 1\}$) in the sequence $\mathbf{x}^n$ be denoted by m_{ab}, and let the vector $\mathbf{m}$, where

$$\mathbf{m} \triangleq [m_{01}, m_{10}, m_{00}, m_{11}], \tag{B.1}$$

be referred to as the count profile of the sequence. All n–bit source sequences with the same initial bit as the sequence $\mathbf{x}^n$ and with the same count profile have the same probability of occurrence as the source sequence $\mathbf{x}^n$. Cover [7] defined an ordered subset S_b that consists of all sequences with the same count profile. The value that the initial bit of a source sequence requires in order to belong to the subset S_b was not specified (i.e. both sequences with zero–valued and nonzero–valued initial bits belong to the same subset S_b, provided that they share the same count profile $\mathbf{m}$).

The source encoder of the enumerative source code that was proposed by Cover [7] encodes each n–bit source sequence as follows (refer to figure B.2 on page 260 for the structure of the codeword). It first encodes the count profile of the source sequence. Each element m_{ab} of the count profile $\mathbf{m}$ is encoded in a total of $\lceil \log_2(n) \rceil$ bits using the conventional binary–coded representation of an integer. It is sufficient to assign $\lceil \log_2(n) \rceil$ bits to the codeword of each element of the count profile, as each element may assume an integer value from zero to $n - 1$. It follows that the count profile of the source sequence is encoded in a total of $4\lceil \log_2(n) \rceil$ bits.

After encoding the count profile $\mathbf{m}$ of the source sequence $\mathbf{x}^n$, the source encoder encodes the index of the source sequence in the ordered subset S_b that consists of all sequences with the count profile $\mathbf{m}$. To calculate the index of the sequence $\mathbf{x}^n$ in the ordered subset S_b, it is necessary to calculate the number of n–bit source sequences that

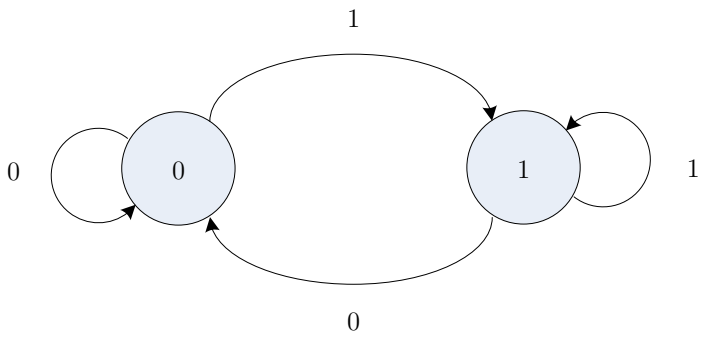


Figure B.1: The state–transition diagram of a first–order binary Markov source.

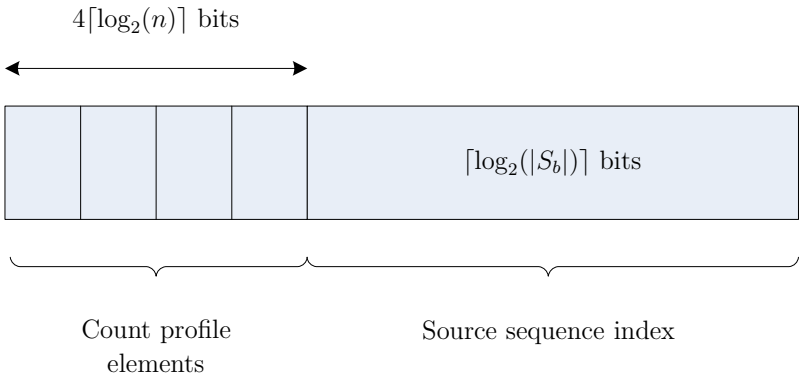


Figure B.2: The codeword assigned to a source sequence of n bits. A description of each codeword field, as well as the length of each field, are provided in the figure.

share the count profile $\mathbf{m}$ and start with a zero–valued bit. Let this total be denoted by $g(\mathbf{m})$. An expression for $g(\mathbf{m})$ is subsequently derived.

Cover [7] proved that the count profile $\mathbf{m}$ of any source sequence that starts with a zero–valued bit satisfies either $m_{01} = m_{10}$ or $m_{01} = m_{10}+1$. The proof of this statement is based on the fact that a sequence's runs of zero–valued bits and nonzero–valued bits alternate. A source sequence with a count profile $\mathbf{m}$ has $m_{10}+1$ runs of zero–valued bits and m_{01} runs of nonzero–valued bits. The same sequence has $m_{10}+m_{00}+1$ zero–valued bits, and $m_{01}+m_{11}$ nonzero–valued bits.

An expression for $g(\mathbf{m})$ may be found by deriving an expression for the number of distinct ways of choosing valid lengths for the zero–valued bit runs in a source sequence. The latter expression is derived by finding the number of distinct ways that positive integer values can be assigned to the variables $r_1, r_2, \ldots r_k$ in order to satisfy the equation

$$r_1 + r_2 + \ldots + r_k = r, \tag{B.2}$$

where r_i denotes the number of bits in the ith zero–valued bit run, k denotes the number of zero–valued bit runs, and r denotes the number of zero–valued bits in the source sequence. Cover [7] stated that there are a total of $\binom{r-1}{k-1}$ positive integer solutions of

equation B.2; this statement implies that there are

$$q_0 = \binom{m_{10} + m_{00}}{m_{10}} \tag{B.3}$$

different ways of choosing valid lengths for the zero–valued bit runs. The same statement implies that there are a total of

$$q_1 = \binom{m_{01} + m_{11} - 1}{m_{01} - 1} \tag{B.4}$$

different ways of choosing valid lengths for the nonzero–valued bit runs. The lengths of the zero–valued bit runs may be chosen independently from the lengths of the nonzero–valued bit runs — it follows that the number of source sequences that share the count profile $\mathbf{m}$ and start with a zero–valued bit equals

$$g(\mathbf{m}) = \begin{cases} \binom{m_{10}+m_{00}}{m_{10}}\binom{m_{01}+m_{11}-1}{m_{01}-1} & \text{if } m_{01} = m_{10} \text{ or } m_{01} = m_{10}+1, \\ 0 & \text{otherwise.} \end{cases}$$

Cover [7] derived an expression for the index of a source sequence $\mathbf{x}^n$ with a count profile $\mathbf{m}$ in the ordered subset S_b of all sequences that share the count profile $\mathbf{m}$. This expression was derived using equation 4.24 as

$$i_{S_b}(\mathbf{x}) = \sum_{k=1}^{n} x_k g(\mathbf{m} - \mathbf{m}(x_1, x_2, x_3 \ldots x_{k-1}, 1 - x_k)), \tag{B.5}$$

where $\mathbf{m}(x_1, x_2, \ldots x_j)$ denotes the count profile of the sequence $\{x_1, x_2, \ldots x_j\}$.

Cover [7] derived an expression for the total number of source sequences that share the count profile $\mathbf{m}$ as

$$|S_b| = g(m_{01}, m_{10}, m_{00}, m_{11}) + g(m_{10}, m_{01}, m_{11}, m_{00}). \tag{B.6}$$

This expression was derived by using the fact that the number of sequences that share the count profile $\mathbf{m}$ and start with a nonzero–valued bit equals the number of sequences that share the count profile $\overline{\mathbf{m}} = (m_{10}, m_{01}, m_{11}, m_{00})$ and start with a zero–valued bit.

B.2 Performance

An expression for the asymptotic code rate of the enumerative source code that was proposed by Cover [7] is derived in what follows. The expression is derived under the assumption that source sequences with lengths that tend to infinity are encoded.

The first step of the derivation is to simplify the expression for the length of the codeword that is assigned to the source sequence $\mathbf{x}^n$. The first field of the codeword has a length of

$$l_1(\mathbf{x}) = 4 \lceil \log_2(n) \rceil \tag{B.7}$$

bits, and the second field of the codeword has a length of

$$l_2(\mathbf{x}) = \lceil \log_2(|S_b|) \rceil \tag{B.8}$$

bits. It follows that a total of

$$l(\mathbf{x}) = 4\lceil \log_2(n) \rceil + \lceil \log_2(|S_b|) \rceil \tag{B.9}$$

bits are required to encode an n–bit source sequence $\mathbf{x}^n$ with a count profile $\mathbf{m}$.

The length of the second field of the codeword may be simplified by first substituting the equation

$$\begin{aligned}
|S_b| &= g(m_{01}, m_{10}, m_{00}, m_{11}) + g(m_{10}, m_{01}, m_{11}, m_{00}) \\
&= \binom{m_{10}+m_{00}}{m_{10}}\binom{m_{01}+m_{11}-1}{m_{01}-1} + \binom{m_{01}+m_{11}}{m_{01}}\binom{m_{10}+m_{00}-1}{m_{10}-1} \\
&= \binom{m_{10}+m_{00}}{m_{10}}\binom{m_{01}+m_{11}}{m_{01}}\left(\frac{m_{01}}{m_{01}+m_{11}}\right) \\
&+ \binom{m_{01}+m_{11}}{m_{01}}\binom{m_{10}+m_{00}}{m_{10}}\left(\frac{m_{10}}{m_{10}+m_{00}}\right) \\
&= \binom{m_{10}+m_{00}}{m_{10}}\binom{m_{01}+m_{11}}{m_{01}}\left[\frac{m_{01}}{m_{01}+m_{11}} + \frac{m_{10}}{m_{10}+m_{00}}\right].
\end{aligned} \tag{B.10}$$

into equation B.8 to obtain the approximation

$$\begin{aligned}
l_2(\mathbf{x}) &= \lceil \log_2(|S_b|) \rceil \\
&\approx \log_2\left[\sqrt{\frac{m_{10}+m_{00}}{m_{00}m_{10}2\pi}}\left(\frac{m_{10}+m_{00}}{m_{00}}\right)^{m_{10}+m_{00}}\left(\frac{m_{00}}{m_{10}}\right)^{m_{10}}\right] && \text{(B.11)} \\
&+ \log_2\left[\sqrt{\frac{m_{01}+m_{11}}{m_{11}m_{01}2\pi}}\left(\frac{m_{01}+m_{11}}{m_{11}}\right)^{m_{01}+m_{11}}\left(\frac{m_{11}}{m_{01}}\right)^{m_{01}}\right] && \text{(B.12)} \\
&+ \log_2\left[\frac{m_{01}}{m_{01}+m_{11}} + \frac{m_{10}}{m_{10}+m_{00}}\right]. && \text{(B.13)}
\end{aligned}$$

The approximation of equations B.11 to B.13 becomes more accurate as n is increased.

Cover [7] derived the expressions

$$E[m_{01}] = \frac{p_{01}p_{10}}{p_{01}+p_{10}}(n-1) = k_{01}(n-1), \tag{B.14}$$

$$E[m_{10}] = \frac{p_{10}p_{01}}{p_{01}+p_{10}}(n-1) = k_{10}(n-1), \tag{B.15}$$

$$E[m_{00}] = \frac{p_{00}p_{10}}{p_{01}+p_{10}}(n-1) = k_{00}(n-1), \tag{B.16}$$

and

$$E[m_{11}] = \frac{p_{11}p_{01}}{p_{01}+p_{10}}(n-1) = k_{11}(n-1), \tag{B.17}$$

where p_{ab} denotes the probability of a state transition from state a to state b. Cover [7] also stated that

$$\left[m_{01}, m_{10}, m_{00}, m_{11}\right] \rightarrow \left[E[m_{01}], E[m_{10}], E[m_{00}], E[m_{11}]\right] \tag{B.18}$$

for ergodic Markov sources (i.e. sources with state transition probabilities that satisfy $p_{00} \neq 1$ and $p_{11} \neq 1$) as n tends to infinity. This statement, together with equations B.11 to B.17, are used to obtain an expression for the normalized asymptotic length $l'_2(\mathbf{x})$ of the second codeword field, where

$$l'_2(\mathbf{x}) = \lim_{n\to\infty} \frac{1}{n} l_2(\mathbf{x}). \tag{B.19}$$

Equation B.11 is simplified as

$$\begin{aligned}
l'_{2,1}(\mathbf{x}) &= \lim_{n\to\infty} \frac{1}{n} \log_2 \left[\sqrt{\frac{m_{10}+m_{00}}{m_{00}m_{10}2\pi}} \left(\frac{m_{10}+m_{00}}{m_{00}} \right)^{m_{10}+m_{00}} \left(\frac{m_{00}}{m_{10}} \right)^{m_{10}} \right] \qquad \text{(B.20)} \\
&= \lim_{n\to\infty} \frac{1}{2n} \log_2 \left[\frac{k_{10}+k_{00}}{k_{00}k_{10}2\pi(n-1)} \right] \\
&+ \lim_{n\to\infty} \frac{(n-1)(k_{10}+k_{00})}{n} \log_2 \left(\frac{k_{10}+k_{00}}{k_{00}} \right) \\
&+ \lim_{n\to\infty} \frac{(n-1)k_{10}}{n} \log_2 \left(\frac{k_{00}}{k_{10}} \right) \\
&= \lim_{n\to\infty} \frac{1}{2n} \log_2 \left[\frac{k_{10}+k_{00}}{k_{00}k_{10}2\pi} \right] + \lim_{n\to\infty} \frac{1}{2n} \log_2 \left[\frac{1}{n-1} \right] \\
&+ (k_{10}+k_{00}) \log_2 \left(\frac{k_{10}+k_{00}}{k_{00}} \right) + k_{10} \log_2 \left(\frac{k_{00}}{k_{10}} \right) \\
&= -\lim_{n\to\infty} \frac{1}{2(n-1)\ln(2)} + (k_{10}+k_{00}) \log_2 \left(\frac{k_{10}+k_{00}}{k_{00}} \right) + k_{10} \log_2 \left(\frac{k_{00}}{k_{10}} \right) \\
&= (k_{10}+k_{00}) \log_2 \left(\frac{k_{10}+k_{00}}{k_{00}} \right) + k_{10} \log_2 \left(\frac{k_{00}}{k_{10}} \right). \qquad \text{(B.21)}
\end{aligned}$$

Equation B.12 is simplified as

$$\begin{aligned}
l'_{2,2}(\mathbf{x}) &= \lim_{n\to\infty} \frac{1}{n} \log_2 \left[\sqrt{\frac{m_{01}+m_{11}}{m_{11}m_{01}2\pi}} \left(\frac{m_{01}+m_{11}}{m_{11}} \right)^{m_{01}+m_{11}} \left(\frac{m_{11}}{m_{01}} \right)^{m_{01}} \right] \qquad \text{(B.22)} \\
&= (k_{01}+k_{11}) \log_2 \left(\frac{k_{01}+k_{11}}{k_{11}} \right) + k_{01} \log_2 \left(\frac{k_{11}}{k_{01}} \right). \qquad \text{(B.23)}
\end{aligned}$$

Equation B.13 is simplified as

$$\begin{aligned}
l'_{2,3}(\mathbf{x}) &= \lim_{n\to\infty} \frac{1}{n} \log_2 \left[\frac{m_{01}}{m_{01}+m_{11}} + \frac{m_{10}}{m_{10}+m_{00}} \right] \\
&= \lim_{n\to\infty} \frac{1}{n} \log_2 \left[\frac{(n-1)^2}{(n-1)^2} \left[\frac{k_{01}(k_{10}+k_{00}) + k_{10}(k_{01}+k_{11})}{k_{01}(k_{10}+k_{00}) + k_{11}(k_{10}+k_{00})} \right] \right] \\
&= 0. \qquad \text{(B.24)}
\end{aligned}$$

It follows that

$$\begin{aligned}
l'(\mathbf{x}) &= \lim_{n\to\infty} \frac{1}{n} l_1(\mathbf{x}) + l'_2(\mathbf{x}) \\
&= \lim_{n\to\infty} \frac{1}{n} l_1(\mathbf{x}) + l'_{2,1}(\mathbf{x}) + l'_{2,2}(\mathbf{x}) + l'_{2,3}(\mathbf{x}) \\
&= \lim_{n\to\infty} \frac{4\log_2(n)}{n} + (k_{10}+k_{00})\log_2\left(\frac{k_{10}+k_{00}}{k_{00}}\right) + k_{10}\log_2\left(\frac{k_{00}}{k_{10}}\right) \\
&+ (k_{01}+k_{11})\log_2\left(\frac{k_{01}+k_{11}}{k_{11}}\right) + k_{01}\log_2\left(\frac{k_{11}}{k_{01}}\right) \\
&= \lim_{n\to\infty} \frac{4}{n\ln(2)} + (k_{10}+k_{00})\log_2\left(\frac{k_{10}+k_{00}}{k_{00}}\right) + k_{10}\log_2\left(\frac{k_{00}}{k_{10}}\right) \\
&+ (k_{01}+k_{11})\log_2\left(\frac{k_{01}+k_{11}}{k_{11}}\right) + k_{01}\log_2\left(\frac{k_{11}}{k_{01}}\right) \\
&= (k_{10}+k_{00})\log_2\left(\frac{k_{10}+k_{00}}{k_{00}}\right) + k_{10}\log_2\left(\frac{k_{00}}{k_{10}}\right) \\
&+ (k_{01}+k_{11})\log_2\left(\frac{k_{01}+k_{11}}{k_{11}}\right) + k_{01}\log_2\left(\frac{k_{11}}{k_{01}}\right). \qquad \text{(B.25)}
\end{aligned}$$

Equations B.14 to B.17 are used to simplify equation B.25, thereby obtaining the expression

$$\begin{aligned}
l'(\mathbf{x}) &= p_0\log_2\left(\frac{1}{p_{00}}\right) + p_1\log_2\left(\frac{1}{p_{11}}\right) - p_0p_{01}\log_2\left(\frac{p_{01}}{p_{00}}\right) \\
&- p_1p_{10}\log_2\left(\frac{p_{10}}{p_{11}}\right) \\
&= p_0\left[p_{00}\log\left(\frac{1}{p_{00}}\right) + p_{01}\log\left(\frac{1}{p_{01}}\right)\right] \\
&+ p_1\left[p_{11}\log\left(\frac{1}{p_{11}}\right) + p_{10}\log\left(\frac{1}{p_{10}}\right)\right] \\
&= p_0h(p_{00}) + p_1h(p_{11}) \qquad \text{(B.26)}
\end{aligned}$$

for the asymptotic code rate of the source code, where

$$p_0 = \frac{p_{10}}{p_{01}+p_{10}} \qquad \text{(B.27)}$$

and

$$p_1 = \frac{p_{01}}{p_{01}+p_{10}} \qquad \text{(B.28)}$$

denote the probability of the source being in state zero and state one, respectively.

The asymptotic code rate of the source code may be compared to the entropy rate of the Markov source (figure B.1) to determine its degree of effectiveness. The entropy

rate of the stationary Markov source is derived as

$$\begin{aligned} H(\mathcal{X}) &= \lim_{n\to\infty}\frac{1}{n}H(X_1, X_2, X_3, \ldots X_n) \\ &= \lim_{n\to\infty}\frac{1}{n}\left[H(X_1) + \sum_{i=2}^{n} H(X_i|X_1, X_2, \ldots X_{i-1})\right] \\ &= \lim_{n\to\infty}\frac{1}{n}\left[H(X_1) + \sum_{i=2}^{n} H(X_i|X_{i-1})\right] \\ &= \lim_{n\to\infty}\frac{(n-1)H(X_2|X_1)}{n} \\ &= H(X_2|X_1), \end{aligned} \tag{B.29}$$

$$H(X_2|X_1) = \sum_{i=0}^{1}\sum_{j=0}^{1}\Pr(X_1 = i, X_2 = j)\log_2\left[\frac{1}{\Pr(X_2 = j|X_1 = i)}\right]. \tag{B.30}$$

The expression for the entropy rate of the Markov source (equation B.29) may be simplified as

$$\begin{aligned} H(\mathcal{X}) &= p_0 p_{00}\log_2\left(\frac{1}{p_{00}}\right) + p_0 p_{01}\log_2\left(\frac{1}{p_{01}}\right) \\ &= p_1 p_{10}\log_2\left(\frac{1}{p_{10}}\right) + p_1 p_{11}\log_2\left(\frac{1}{p_{11}}\right) \\ &= p_0 h(p_{00}) + p_1 h(p_{11}), \end{aligned} \tag{B.31}$$

where $1 - p_{00} = p_{01}$ and $1 - p_{11} = p_{10}$. Equation B.31 is identical to the expression for the asymptotic code rate of the source code (equation B.26). The source code is therefore optimal in terms of its asymptotic code rate.